

UNIVERSITÀ DI PISA

Corso di Laurea Triennale in Informatica

Realizzazione di un sistema di video Real-Time con canali multipath da drone a stazione di comando

Tutore Accademico:
Prof.ssa Federica Paganelli
Tutore Aziendale:
Dott. Alberto Gotta

Candidato:
Simone Iozia

Anno Accademico 2019/2020

Indice

1	Introduzione	2
2	Modello trasmittivo RTP/RTCP	4
2.1	Requisiti per protocolli in Real-time	6
2.2	Protocollo RTP	7
2.3	Qualità del servizio	11
2.4	Protocollo RTCP	12
2.5	Host multi-homing e Multipath-RTP	14
3	GStreamer	16
3.1	Architettura del framework	17
3.2	Librerie principali	19
4	Sviluppo delle pipelines	20
4.1	Sender	20
4.2	Scheduler	21
4.2.1	Implementazione	24
4.3	Receiver	31
4.4	Controller	33
5	Test e grafici	37
6	Conclusione e lavori futuri	40
A	Installazione del sistema	42
B	Codice Pipelines	44
B.1	sender.py	44
B.2	receiver.sh	53

Capitolo 1

Introduzione

L’obiettivo di questo tirocinio è l’implementazione di canali di comunicazione per la trasmissione in Real Time di un flusso video catturato da una telecamera posta su un drone verso una stazione di comando, a supporto di operazioni di Realtà Aumentata. A tal fine ho implementato un algoritmo di scheduling che gestisca la ripartizione del flusso multimediale su un canale di comunicazione *multipath*, ovvero un sistema che prevede la presenza di più canali fisici fra sorgente e destinazione. L’utilità di implementare un sistema multipath è legata alla scelta di poter adoperare in futuro anche contemporaneamente reti WiFi e/o connessioni multiple 5G per la ricezione del flusso multimediale e dei dati telemetrici sulla stazione di terra ad alta Qualità del Servizio (QoS) ed elevato bitrate.

Nella fase iniziale del tirocinio, ho approfondito la teoria alla base dei protocolli di rete in Real-Time RTP e RTCP, che definiscono il formato di un pacchetto standard per il transito in tempo reale di video (e audio) su Internet, e delle loro estensioni multipath. Ho compiuto anche uno studio sulla piattaforma software che mi è stata proposta, GStreamer, e sui molteplici plug-ins che lo compongono; la scelta di questa piattaforma è dovuto al suo utilizzo in applicazioni come QGroundControl¹, che facilita il controllo di droni in volo, offrendo anche uno streaming video dal veivolo. In figura 1.1 si può vedere una possibile schermata di QGroundControl.

¹<http://qgroundcontrol.com/>



Figura 1.1: Inferfaccia di QGroundControl[3]

Quindi, nella seconda fase, sulla base di tali conoscenze, ho sviluppato un plug-in di GStreamer che implementa il modello di scheduler conosciuto come *Interleaved Weighted Round-Robin*, in modo da smistare i pacchetti attraverso i canali multipli: i pesi associati ai rate di trasmissione su ciascuno dei sotto-flussi vengono aggiornati in maniera dinamica, in accordo con i feedback di QoS (Quality of Service) di ciascuno di essi, ricevuti attraverso i rispettivi canali di controllo.

I test sono stati realizzati mediante un Raspberry Pi (fornitomi dal tutore aziendale del CNR di Pisa), con il quale vengono catturati i pacchetti video dalla telecamera ad esso associata, smistati dallo scheduler e spediti verso la destinazione desiderata, nella fattispecie un portatile.

Questo tirocinio curriculare è stato svolto in modalità smart-working secondo i protocolli stabiliti dai vari enti, in seguito alla pandemia di Covid-19.

Capitolo 2

Modello trasmittivo RTP/RTCP

Dall'avvento di Internet, si è assistito a un'evoluzione e sviluppo dei servizi ad esso collegati. Oltre ai servizi tradizionali, quali posta elettronica, ricerche web, e via dicendo, gli utenti di tutto il mondo richiedono benefici più avanzati come piattaforme per l'erogazione di materiale multimediale oppure servizi di VoIP¹. Oggigiorno si è potuto notare come, con l'inizio della pandemia di Covid-19, sono diventate indispensabili, ad esempio, chiamate via Whatsapp o videoconferenze su Google Meet, rendendo fondamentale il progresso tecnologico riguardo alla comunicazione e alla gestione di flussi multimediali in Real-Time.

Prima di descrivere i protocolli di rete adatti a questo tipo di esigenza, è bene discutere di alcune caratteristiche generali del trasferimento audio/video in tempo reale, descritte in modo accurato in [4]:

- **Relazioni temporali:** su una rete a commutazione di pacchetto (come Internet), è importante preservare le relazioni temporali tra i vari pacchetti di una stessa sessione evitando, ad esempio, intervalli di tempo vuoti tra i diversi pacchetti, fenomeno che prende il nome di jitter, descritto meglio nei paragrafi successivi;
- **Timestamp:** per evitare fenomeni come il jitter, è necessario porre delle marcature temporali, dette timestamp. In questo modo ciascun pacchetto può mostrare il tempo in cui è stato prodotto rispetto al

¹Voice over IP, ovvero telefonia su IP

primo pacchetto o al pacchetto precedente, al fine di aiutare il ricevente nella riproduzione del flusso multimediale;

- **Buffer di playback:** il ricevente può ritardare la riproduzione dei dati in modo da memorizzarli in una coda, il buffer di playback appunto, così da iniziare l'estrazione dei dati quando viene raggiunta la soglia di riempimento della stessa. È bene ricordare che l'immissione dei dati nella coda avviene in base alla velocità di arrivo dei pacchetti, mentre l'estrazione avviene con una velocità costante: così facendo, se il ritardo della rete è minore del tempo necessario per riprodurre tutti i dati all'interno della soglia del buffer, non vi sarà presenza di jitter. La figura 2.1 mostra un grafico sullo stato dei pacchetti nel tempo e il ritardo introdotto dalla rete e dal buffer;

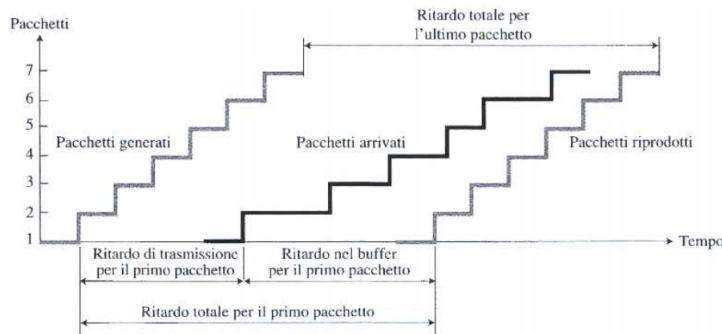


Figura 2.1: Linee temporali dei pacchetti[4]

- **Ordinamento:** il timestamp da solo non è sufficiente per poter rordinare il flusso. Per questo motivo, è necessario che i pacchetti siano numerati per mezzo di un numero di sequenza, così da ricavare informazioni riguardo la perdita di pacchetti e ordinare la sequenza in modo corretto per riprodurre materiale multimediale di alta qualità;
- **Traduzione:** la fruizione di un contenuto multimediale a un largo pubblico di utenti aggiunge il problema della risoluzione; un formato video ad alta risoluzione spesso deve essere decodificato e ricodificato in un video di qualità inferiore, affinché ne possano godere anche destinatari che hanno a disposizione minori ampiezze di banda o dispositivi con scarse prestazioni.

2.1 Requisiti per protocolli in Real-time

È facile capire come la natura dei dati video o audio sia molto diversa da quei dati di solito scambiati durante il trasferimento dei normali file o delle e-mail, di conseguenza quando si gestisce un flusso audio/video bisogna far fronte anche a requisiti differenti, che i comuni protocolli di rete come UDP e TCP non riescono a soddisfare in modo completo.

Largamente trattati in [4], alcuni di questi requisiti sono i seguenti:

- **Negoziazione tra mittente e ricevente:** dato che non esiste una codifica standard per i dati, risulta molto importante che i programmi di mittente e destinatario concordino la codifica prima di iniziare il trasferimento dei pacchetti, in caso contrario la comunicazione sarebbe impossibile. Inoltre vi dev'essere anche la possibilità di cambiare codifica in corso d'opera per poter ovviare a problemi di congestione di rete;
- **Creazione di un flusso di pacchetti:** la difficoltà di trovare un protocollo di rete per il trasferimento di video Real-Time sta nel soddisfare contemporaneamente le esigenze di mandare un flusso continuo di dati all'interno di messaggi con confini ben definiti e relazionare tra di loro i vari frame. TCP ha il "difetto" di trasferire byte, di dimensioni molto più piccoli dei frame, mentre UDP non fornisce relazioni fra i datagrammi;
- **Sincronizzazione delle sorgenti:** un'applicazione che utilizza diverse sorgenti deve poter sincronizzare i vari flussi di pacchetti che riceve, in modo da fornire una buona qualità del servizio;
- **Controllo degli errori e della congestione di rete:** nonostante TCP abbia una gestione ottimale della congestione di rete, nei casi in cui si perdessero o si danneggiassero pacchetti, non è possibile ritrasmettere i dati, poiché aggiungerebbero molto ritardo al video, motivo per cui TCP non può essere scelto come protocollo per applicazioni multimediali. Pertanto è necessario gestire tali problematiche in modo diverso;
- **Eliminazione del jitter:** necessità di gestire il timestamp e i numeri di sequenza dei pacchetti al fine di eliminare ritardi irregolari durante il trasferimento dei pacchetti;

- **Supporto del multicast:** in una comunicazione in tempo reale bisogna mettere in conto che ci possano essere più di 2 utenti che interagiscono tra di loro, per cui TCP, ancora una volta, non può essere preso come modello in quanto supporta solo una comunicazione unicast.

In questo progetto quindi, per soddisfare tali requisiti e far fronte alle varie esigenze, a supporto di UDP, è stato adottato il protocollo noto come RTP/RTCP definito dall'Audio-Video Transport Working Group, facente capo alla IETF (Internet Engineering Task Force) nel RFC 3550[7].

2.2 Protocollo RTP

RTP (Real-time Transport Protocol) fornisce funzionalità di trasporto end-to-end adatte per applicazioni che trasmettono dati in tempo reale. È gestito a livello applicazione ma, come illustrato in figura 2.2, nell'architettura ISO/OSI lo troviamo a livello di trasporto, nonostante non possegga la nozione di connessione: i servizi di frammentazione e assemblaggio dei pacchetti oppure la garanzia della consegna dei pacchetti a destinazione devono stare a carico degli strati inferiori. RTP quindi collabora congiuntamente con un altro protocollo di trasporto, solitamente UDP. Quest'ultimo, essendo privo di connessione, riduce la latenza ma ha il difetto di non garantire affidabilità né corretto ordinamento dei pacchetti, e potrebbe anche creare problemi di congestione della rete.

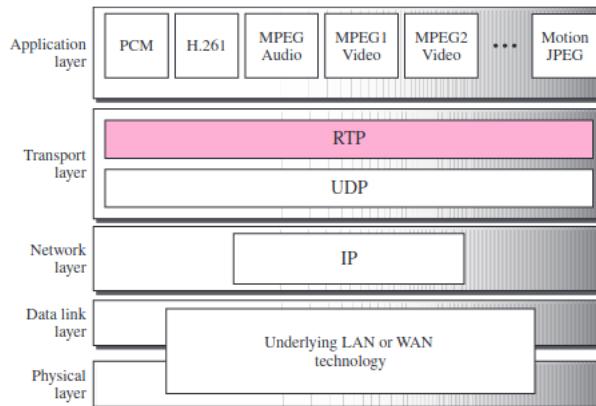


Figura 2.2: Posizione di RTP nella pila TCP/IP[4]

Per queste ragioni è stato creato RTP, ovvero un formato di pacchetti che permetta di dare all'applicazione delle informazioni, al fine di implementare trasporti influenzati dal tempo, come:

- identificazione della sorgente;
- tipo di dato trasportato;
- sequenza dei pacchetti, in modo da ricostruire l'ordine e controllare eventuali perdite;
- sincronizzazione dei molteplici flussi di dati, anche provenienti da mittenti differenti.

RTP viene trattato come un protocollo di livello applicazione, quindi encapsulato in datagrammi UDP, ma a differenza degli altri protocolli, non vi sono porte note assegnate ad esso. L'unico vincolo per quanto riguarda il numero di porta è che questa dev'essere di numero pari.

Header RTP

Un pacchetto RTP (figura 2.4) è formato da almeno 12 bytes (l'header) seguiti dal payload. Un header RTP (figura 2.3) è composto da diversi campi:

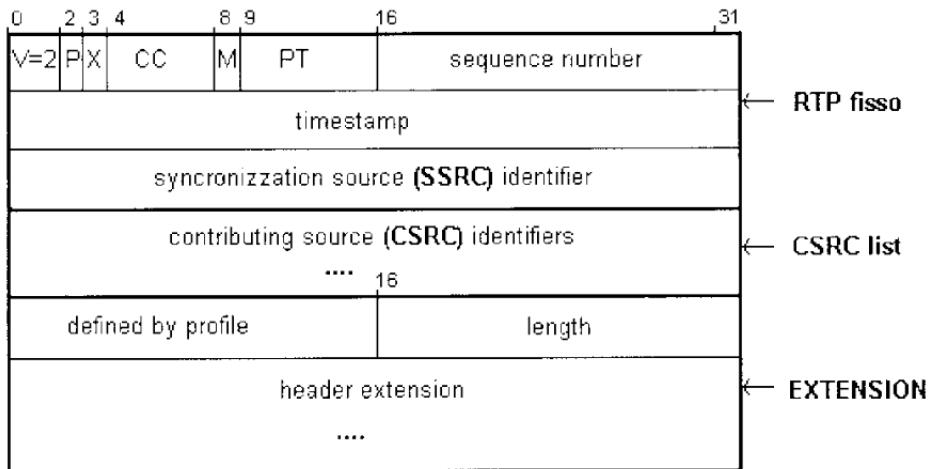


Figura 2.3: Header RTP[5]

- **Versione (V)**: indica la versione del protocollo (quella corrente è la 2);
- **Padding (P)**: bit che indica se il pacchetto contiene alla fine dei padding bytes, ovvero dei dati aggiuntivi che però non fanno parte del payload; di solito è usato per scopi crittanalitici;
- **Estensione (E)**: bit che indica se la parte fissa del header è seguita da un extension header, con un formato da definire.
- **Conteggio CSRC (CC)**: indica il numero di CSRC (Contributing Source, ovvero le diverse fonti che contribuiscono al payload), per un massimo di 16 possibili identificatori;
- **Marcatore (M)**: settato a livello applicativo , indica se il pacchetto ha qualche tipo di particolare rilevanza per l'applicazione;
- **Payload (PT)**: indica il formato del payload e quindi la sua interpretazione all'interno dell'applicazione;
- **Numero di sequenza**: numero che identifica la posizione del pacchetto all'interno dello stream; viene incrementato di uno ogni volta che un pacchetto RTP viene inoltrato da una stessa fonte, per questo è molto utile nella registrazione di eventuali perdite di pacchetti (anche se RTP non prende alcun tipo di provvedimento in questi casi, lasciando la decisione agli altri livelli) e nel riordinamento di quest'ultimi. Il suo valore iniziale dovrebbe essere casuale, diverso per ogni SSRC presente, in modo da evitare attacchi crittanalitici;
- **Timestamp**: indica l'istante di tempo preso nel momento dell'emissione del primo byte del pacchetto: il dato istante è generato da un clock (la cui frequenza dipende anche dal tipo di codifica del pacchetto) che si incrementa in modo monotono e lineare nel tempo. Usato per funzioni di sincronizzazione e calcolo del jitter;
- **SSRC (Synchronization Source Identifier)**: numero che identifica in modo univoco la fonte dello stream all'interno di una sessione RTP, evitando di dipendere dall'indirizzo di rete. Ad esempio, un'applicazione che cattura video da una telecamera e registra audio da un microfono, creerà 2 diversi SSRC in modo da identificare univocamente

(e quindi possibilmente dividere) video e audio. Scelto in modo randomico, ogni SSRC ha un proprio numero di sequenza iniziale e lo stesso clock;

- **Array di CSRC:** lista di SSRC che contribuiscono a formare uno stream;
- **Estensione Header:** prevista a fini sperimentali, sono bytes (e quindi informazioni aggiuntive) che possono essere aggiunti a piacere, a livello applicativo, dallo sviluppatore, alla fine dell'header. Ho sfruttato questo campo per aggiungere una funzione multipath al protocollo.

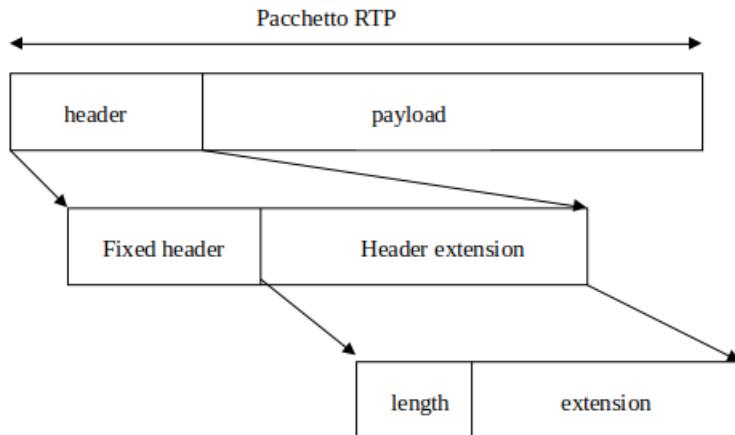


Figura 2.4: Struttura di un pacchetto RTP[6]

Payload

Descritto molto bene in [6], il payload RTP è la parte del pacchetto che è predisposta al trasporto dei dati forniti all'applicazione: tali dati possono essere campioni di un segnale audio oppure fotogrammi di un video opportunamente codificati. In genere il payload di un pacchetto RTP non corrisponde ad un solo campione o fotogramma, ma può contenere più campioni o essere, viceversa, una parte di un solo fotogramma. Il payload non ha una lunghezza definita, ma può dipendere dalla codifica usata e dal numero di campioni per pacchetto: in alcune codifiche la quantità dei dati può addirittura variare da

campione a campione ed è quindi possibile che pacchetti provenienti dalla stessa sorgente abbiano payload con differente lunghezza. L'unica limitazione alla lunghezza del payload è determinata dal protocollo sottostante: ad esempio, nel caso in cui RTP utilizzi il protocollo UDP, la lunghezza del pacchetto non può superare 65535 byte.

2.3 Qualità del servizio

Applicazioni multimediali richiedono prestazioni che siano garantite e prevedibili e sono anche molto sensibili al ritardo. Un approccio *best effort*, come quello usato da Internet, non è sufficiente per questo tipo di traffico dati. Al termine best effort, si contrappone il termine Qualità del Servizio (QoS) che indica tutta la serie di tecniche e meccanismi per poter garantire buone prestazioni.

Come analizzato in [4], per poter garantire una buona qualità del servizio, è necessario specificare quali caratteristiche bisogna migliorare o ridurre:

- **Affidabilità:** capacità di un flusso di rete di consegnare in modo corretto i dati a destinazione. Le conseguenze di una buona affidabilità sono pacchetti persi e/o danneggiati e perdita di riscontri.
- **Latenza:** ridurre il ritardo, introdotto dal mezzo di trasmissione, dei pacchetti, per il loro passaggio da sorgente a destinazione;
- **Jitter:** ridurre la variazione del ritardo dei pacchetti, se c'è ritardo è importante che sia uguale per tutti i pacchetti;
- **Aampiezza di banda:** avere a disposizione una banda necessaria per far fronte alle esigenze dell'applicazione che si sta adoperando.

2.4 Protocollo RTCP

La raccolta delle statistiche sulla qualità del servizio fornito dalle varie sessioni RTP è affidata invece dal protocollo RTCP (Real-Time Control Protocol). Definito come RTP nel RFC 3550[7], la funzione primaria di RTCP è quella di monitorare e trasmettere da un receiver a un sender, e viceversa, informazioni riguardanti il numero di pacchetti persi, valori di jitter o RTT²; è importante sottolineare che la gestione dei possibili fenomeni causati da statistiche non ottimali non è affidata a questo protocollo, bensì esso fornisce queste informazioni affinché è l'applicazione stessa a gestire in modo efficiente il flusso dei dati attraverso le sessioni o aggiustare il tipo di dati trasmessi in modo da migliorare le statistiche risultanti.

In secondo luogo, fornisce informazioni di controllo, per esempio, quando una sorgente del sender manda un segnale in modo da avvisare gli altri partecipanti che sta lasciando la sessione oppure quando viene richiesto uno scambio di informazioni quali nome, indirizzo, e-mail dei vari partecipanti alla sessione.

Al crescere del numero dei partecipanti e delle sessioni, incrementa in modo direttamente proporzionale anche il numero dei pacchetti RTCP inviati, con il rischio di creare congestione nella rete. Per questo motivo, RTCP non dovrebbe usare più del 5% della banda utilizzata per i pacchetti RTP. Ciò significa che bisognerebbe cambiare l'intervallo di tempo che intercorre tra 2 pacchetti RTCP consecutivi, in modo inversamente proporzionale al numero dei pacchetti RTP da trasmettere.

Infine se a una sessione RTP è assegnata una porta UDP pari a $2n$, il flusso RTCP associato a quella sessione sarà assegnata alla porta UDP $2n+1$.

Header RTCP

Un pacchetto RTCP è formato da un header (parte fissa) seguito dal tipo di report o informazione che trasporta e dai vari blocchi di report, ognuno per ogni partecipante alla sessione. L'header RTCP (figura 2.5) è composto da:

- **Versione (V)**: indica la versione del protocollo (come per RTP, quella corrente è la 2);

²Round Trip Time.

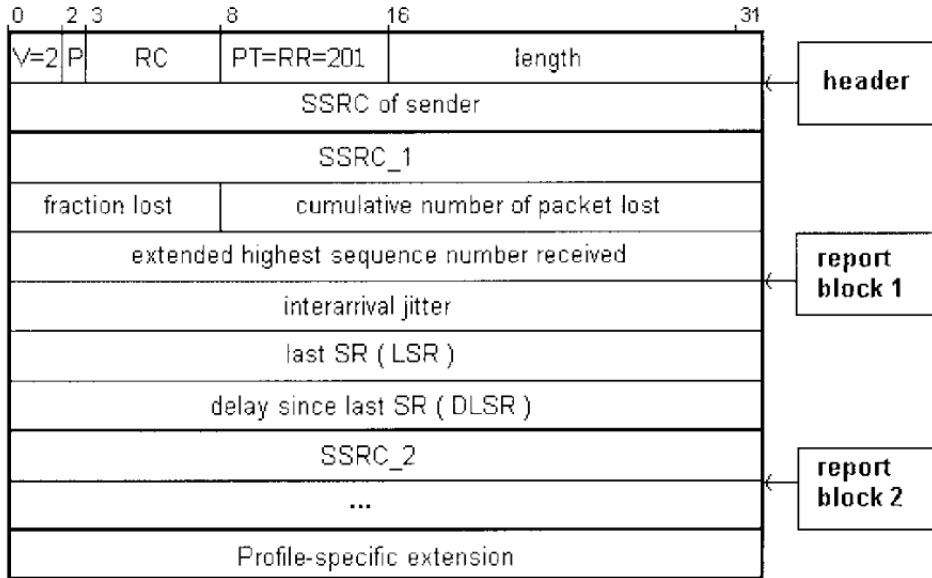


Figura 2.5: Header RTCP di tipo RR[5]

- **Padding (P)**: bit che indica se il pacchetto contiene alla fine dei padding bytes, ovvero dei dati aggiuntivi che però non fanno parte del payload; di solito è usato per scopi crittanalitici;
- **Conteggio Report di Ricezione (RC)**: indica il numero di blocchi di report che contiene il pacchetto;
- **Tipo pacchetto (PT)**: indica il tipo di report che il pacchetto trasporta (vedi dopo);
- **Lunghezza**: indica la lunghezza dell'intero pacchetto;
- **SSRC (Synchronization Source Identifier)**: numero che identifica in modo univoco la sessione a cui appartiene il report o l'informazione (ma non il partecipante, che sarà identificato tra i vari blocchi di report). Esso non è sempre presente nell'header, ma in base al tipo di pacchetto può essere specificato nei blocchi di informazioni che seguono l'header.

Tipo di pacchetto RTCP

Esistono 5 diversi tipi di pacchetti che RTCP può trasmettere:

- **SR (Sender Report, PT=200)**: inviato dal sender al receiver ogni intervallo di tempo definito, trasporta le statistiche riguardanti la spedizione dei pacchetti RTP effettuate dai vari partecipanti, in particolare fissa i vari timestamp, assoluti o RTP, usati dal receiver per sincronizzare messaggi RTP;
- **RR (Receiver Report, PT=201)**: inviato dal receiver al sender ogni intervallo di tempo definito, trasporta le statistiche riguardanti la ricezione dei pacchetti RTP dai vari partecipanti, quindi specifica se ci sono stati pacchetti persi, con quanto ritardo sono arrivati, e via dicendo;
- **SDES (Source Description, PT=202)**: contiene elementi di descrizione dei vari partecipanti, come il canonico CNAME (usato per associare un nome di dominio a un alias) l'email, ecc.;
- **BYE (Goodbye, PT=203)**: indica quale partecipante vuole lasciare la sessione (e anche il motivo) così da fermare il flusso RTP da quella sorgente e informare di ciò gli altri partecipanti;
- **APP (Application-specific message, PT=204)**: usato per progettare estensioni specifiche dell'applicazione al protocollo RTCP

2.5 Host multi-homing e Multipath-RTP

Ai fini del nostro progetto, fondamentale è la possibilità del dispositivo elettronico, come un personal computer ad esempio, che “ospita” il receiver, di avere la proprietà di multihoming.

Il multihoming consiste nel fatto che tale dispositivo possegga diverse interfacce di rete (Ethernet, Wireless, connessioni 5G, ecc.), ognuna col proprio indirizzo IP, con l'obiettivo di usarle allo stesso tempo. In questo modo, l'applicazione può ricevere pacchetti da più di una interfaccia di rete.

In questo contesto si inserisce Multipath-RTP (MPRTP), estensione facoltativa del protocollo RTP, descritta in [8], che consente di suddividere

un singolo flusso RTP in più flussi secondari che sono trasmessi su percorsi differenti. Consentendo a RTP di utilizzare più percorsi di trasmissione contemporaneamente, si otterranno i seguenti benefici:

- **Qualità superiore:** mettendo in comune le risorse di molteplici percorsi, è possibile l'utilizzo di codec di qualità superiore e bit rate più elevati;
- **Bilanciamento del carico:** trasmettendo un flusso RTP su più percorsi si riduce l'utilizzo dell'ampiezza di banda su un unico percorso, riservando maggiori risorse per un altro traffico di dati sullo stesso percorso;
- **Tolleranza agli errori:** quando vengono utilizzati più percorsi insieme a meccanismi di ridondanza (FEC³, ritrasmissioni, ecc.), si ha un impatto minore sulla qualità complessiva del flusso RTP.

La principale funzionalità dell'estensione è quindi la gestione dei pacchetti nei vari sotto-flussi, aventi SSRC differenti (e quindi anche numeri di sequenza differenti) tra loro. La divisione del flusso RTP originale sarà affidato a un algoritmo di scheduling che smisterà i vari pacchetti nei percorsi in base alla qualità della connessione di rete e della finestra di congestione. Se il dispositivo ricevente è un host multi-homed, i pacchetti possono essere inviati e ricevuti da interfacce di rete differenti. Infine MPRTP dovrà occuparsi di mixare i diversi sotto-flussi in uno, così da ricreare il flusso originale.

Ogni sotto-flusso avrà la sua sessione RTP e il suo report RTCP. Il sender invierà periodicamente al receiver un pacchetto RTCP di tipo SR per ogni sotto-flusso, contenente il timestamp dell'ultimo pacchetto spedito, il numero di pacchetti RTP e bytes spediti riguardanti appunto quella sessione. Il receiver riceve tale pacchetto SR e a sua volta invia al sender, sempre per ogni sotto-flusso, un pacchetto RR, contentente questa volta tutte le statistiche della sessione come numero di pacchetti persi, jitter, ecc. che poi lo sviluppatore potrà sfruttare per manipolare le impostazioni del sender a livello applicativo.

In sostanza si crea un loop di feedback RTCP.

³Forward Error Correction.

Capitolo 3

GStreamer



Figura 3.1: Logo di GStreamer[2]

La piattaforma software selezionata per implementare il sistema di sender e receiver è GStreamer¹.

GStreamer è molto diffuso in applicazioni multimediali ed è di fatto utilizzato in applicazioni di Ground Control Station (GCS) come QgroundControl, piattaforma open-source adibita al controllo del volo di droni, e presenta numerose caratteristiche come, ad esempio, la pianificazione della missione in caso di volo autonomo, la visualizzazione della mappa di volo mostrando posizione, coordinate e strumenti del veicolo, e, in particolare, streaming video dal veivolo.

GStreamer è un software gratuito e open source soggetto ai termini della GNU Lesser General Public License (LGPL). Gstreamer supporta un'ampia varietà di componenti per la gestione dei media, inclusa la semplice riproduzione audio, video, registrazione, streaming e editing. È progettato per funzionare su una varietà di sistemi operativi, ad esempio Linux, OpenSolaris, Android, macOS, iOS, Windows, ecc.

¹<https://gstreamer.freedesktop.org/>

3.1 Architettura del framework

Come illustrato nel suo manuale [1], alla base di Gstreamer, c'è il design della **pipeline**, ovvero componenti software collegati fra loro in cascata: l'input di un elemento diventa l'output dell'elemento precedente. Le pipelines, eseguite in un thread a parte, possono essere impostate in PAUSED o in PLAYING; nel secondo caso, il flusso di dati multimediale inizierà a fluire attraverso i vari elementi di cui la pipeline è composta.

Gli elementi software che caratterizzano Gstreamer presentano uno o più pad di entrata e uscita, dove per pad si intendono le "porte" dell'elemento.

Illustrati in figura 3.2, abbiamo 3 possibili tipi di elementi o plug-in:

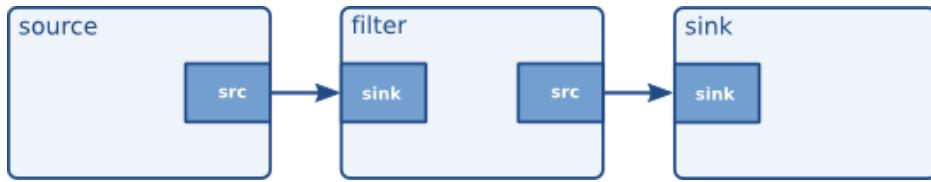


Figura 3.2: Esempio di una pipeline in GStreamer[1]

- **Source (sorgente)**: è l'elemento che ottiene i dati da una struttura esterna alla piattaforma, come connessioni di rete o un file. Presenta solo un pad di uscita.
- **Filter (filtro)**: elemento che riceve dati da un altro plug-in dal proprio pad di entrata, manipola i dati secondo la funzione per cui è stato sviluppato (fanno parte di questo insieme elementi che codificano, incapsulano, ecc.), quindi passa tali dati verso il plug-in successivo attraverso il pad di uscita.
- **Sink (pozzo)**: è l'elemento che invece "dialoga" con i driver video o audio per visualizzare e riprodurre i dati in ingresso dall'unico pad di entrata.

Inoltre un insieme di elementi può essere contenuto in un bin, astraendo così molta complessità all'applicazione che si sta sviluppando.

Un flusso di dati può trasferirsi da un elemento a un altro se i 2 elementi consentono il passaggio di tipi di dati compatibili fra loro: facendo un'analogia, è come collegare un lettore DVD e un'amplificatore poiché hanno

un cavo jack compatibile, mentre non è possibile collegare un proiettore a un'amplificatore, in quanto hanno cavi jack differenti.

GStreamer offre parecchi meccanismi per la comunicazione e lo scambio dei dati tra gli elementi (vedi figura 3.3):

- **buffers**: oggetti per il passaggio di dati in streaming tra gli elementi nella pipeline. I buffer viaggiano sempre dalle sorgenti ai pozzi;
- **eventi**: oggetti inviati tra elementi o dall'applicazione agli elementi. Gli eventi possono viaggiare anche "controcorrente". Gli eventi che seguono la direzione del flusso possono essere sincronizzati con i dati;
- **messaggi**: oggetti inviati dagli elementi sul bus della pipeline, dove verranno conservati per essere raccolti dall'applicazione. Vengono utilizzati per trasmettere in modo thread-safe informazioni come errori, tag, cambiamenti di stato, stato del buffer, reindirizzamenti ecc. Possono essere intercettati in modo sincrono dal contesto del thread di streaming dell'elemento che invia il messaggio, ma in genere vengono gestiti in modo asincrono dall'applicazione dal thread principale.
- **queries**: consentono alle applicazioni o agli elementi di richiedere informazioni come la durata o la posizione di riproduzione corrente dalla pipeline. Ricevono sempre una risposta sincrona.

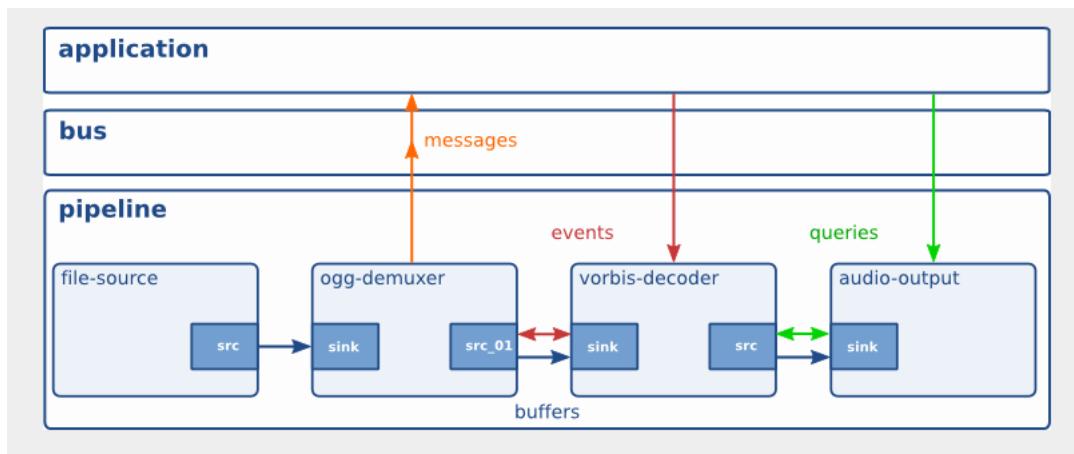


Figura 3.3: Funzionamento di un bus in GStreamer[1]

Una delle peculiarità di GStreamer è la possibilità, essendo open-source, di inserire nuovi plug-in e di modificare quelli esistenti. Tutti i plug-in sono scritti mediante il linguaggio di programmazione C; nonostante ciò, i plug-in possono essere anche richiamati da pipeline realizzate mediante Python.

3.2 Librerie principali

Tutti i plug-in che compongono GStreamer fanno parte di librerie, e in particolare si ricordano:

- **gststreamer**: contiene quegli headers, elementi e file di librerie che sono il cuore della piattaforma;
- **gst_plugins_base**: contiene quei plug-ins essenziali al funzionamento dell'intero sistema, ben curati e ben mantenuti, fornisce buona parte degli elementi per scrivere nuovi plug-ins. Includono, fra gli altri, plug-ins di supporto, encoder, decoder, filtri;
- **gst_plugins_good**: contiene quei plug-ins “buoni”, ovvero plug-ins che hanno un codice di ottima qualità, corrette funzionalità;
- **gst_plugins_ugly**: contiene plug-ins di buona qualità ma che magari potrebbero creare dei problemi nella loro distribuzione, in quanto licenze e librerie di supporto potrebbero non essere come vorremmo.
- **gst_plugins_bad**: contiene quei plug-ins che potrebbero essere di buona qualità, ma spesso peccano di una buona revisione del codice, di documentazione affidabile, di essere ben testati oppure ancora di una continua manutenzione.
- **gst_libav**: contiene i principali plug-ins con funzioni di encoder o decoder.

Capitolo 4

Sviluppo delle pipelines

Il nucleo di questo tirocinio è costituito dalle 2 pipelines sviluppate con GStreamer: il sender, posto sul sistema locato sul drone, che cattura i pacchetti video in Real-Time e li smista nei vari canali UDP; il receiver, posto sul sistema della stazione di controllo, che ricava le statistiche di ogni canale e ricrea il flusso originale con l'obiettivo di visualizzarlo a schermo. Particolare attenzione è stata rivolta alla creazione di uno scheduler per lo smistamento e la ridondanza dei pacchetti RTP nel sender, basata sull'idea del tipo di scheduler noto come “Interleaving Weighted Round-Robin”, modellato sulle informazioni ricevute a run-time da un “controller” che elabora le statistiche dei canali provenienti dal receiver sotto forma di pacchetti RTCP e restituisce dei nuovi tassi di smistamento e ridondanza in modo da rendere sempre efficiente la qualità del video ottenuto.

4.1 Sender

La prima funzione del sender è quella di catturare il video dalla telecamera posta sul drone. Tale telecamera è collegata a un Raspberry PI, sul quale è installato il sistema operativo Raspbian. Per prelevare i dati della telecamera è stato usato il comando da terminale *raspivid*.

La codifica del video in entrata è realizzata nel formato H264, usato per la compressione di contenuto video che permette di registrare e distribuire video con qualità full-HD cercando di sfruttare meno larghezza di banda possibile. Altri vantaggi della codifica H264 sono quelli di essere adatto alla trasmissione multicast del flusso video, avere un bit-rate minore rispetto ad

altri tipi di codifica e possibilità di usare buffers di minori dimensioni per memorizzare i pacchetti.

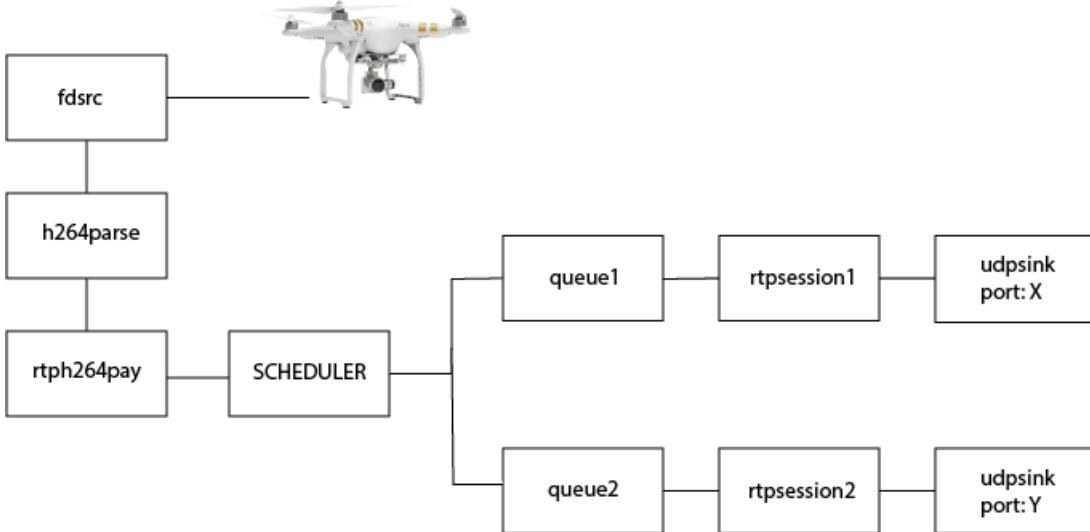


Figura 4.1: Pipeline Sender

Come si può notare in figura 4.1, per manipolare il video catturato, Gstreamer fornisce il plug-in *fdsrc*, elemento che inoltra i dati ricevuti da un descrittore di file, in questo caso raspivid. Tramite *h264parse*, i pacchetti vengono controllati qualora ci fossero imperfezioni nella struttura della codifica H264, quindi vengono incapsulati in pacchetti RTP. Con *rtpH264pay*, viene assegnato un SSRC casuale ai pacchetti RTP, che identifica univocamente la sorgente del flusso video originale.

A questo punto i pacchetti vengono smistati dallo scheduler (la spiegazione dettagliata del suo funzionamento si trova nel paragrafo successivo) nei vari canali, ciascuno identificato da una differente sessione RTP, tramite *rtpsession*. Infine i pacchetti sono inviati tramite il protocollo UDP verso l'host indicato e la porta assegnata, rigorosamente di numero pari.

4.2 Scheduler

Lo scheduler è il componente della pipeline adibito alla funzione di gestione dell'inoltro dei pacchetti in entrata (*packet forwarder*) ai componenti suc-

cessivi secondo un algoritmo di scheduling. Lo scheduler è associato a una serie di code, tante quanto il numero di canali UDP previsti: la coda sarà di tipo FIFO¹, dove il primo oggetto, che sarà inserito nella coda, sarà anche il primo che verrà prelevato dal componente successivo.

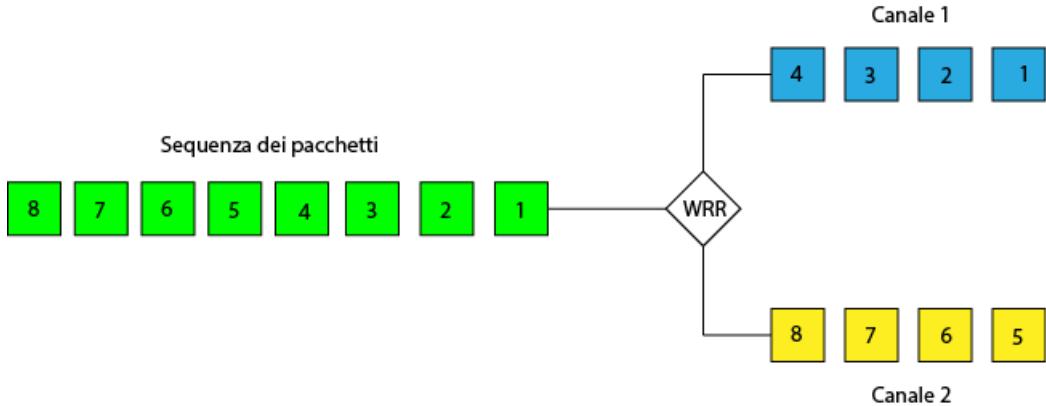


Figura 4.2: Esempio di Weighted Round-Robin, con pesi al 50%

Per realizzare lo scheduler, l'idea di base è stato il modello di scheduling Weighted Round-Robin (WRR). Il fatto che tale scheduler sia *weighted*, cioè pesato, significa che ad ogni canale di uscita sono assegnati dei pesi che corrispondono, in proporzione, al numero di pacchetti che devono passare consecutivamente attraverso quel determinato canale in ogni ciclo di pacchetti fissato. La pecca di questo tipo di scheduler è che c'è il rischio di non usare alcuni canali per lunghi periodi di tempo: per esempio, come illustrato in figura 4.2, nel caso di 2 canali e di pesi pari al 50%, entrambi i canali vengono trascurati uno alla volta per la metà del tempo di un ciclo, e questo potrebbe essere inefficiente qualora un ciclo durasse per un tempo prolungato e uno dei 2 canali non funzionasse in modo ottimale.

¹First In, First Out.

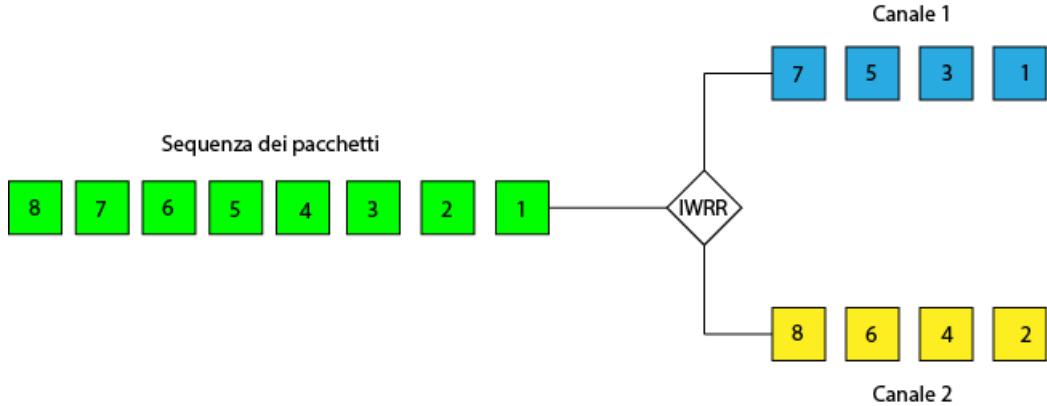


Figura 4.3: Esempio di Interleaving Weighted Round-Robin, con pesi al 50%

Per ovviare a questo problema, si adotta una strategia *interleaving* (vedi figura 4.3), ovvero smistare pacchetti consecutivi verso canali diversi, sempre in relazione ai pesi assegnati ad ogni flusso; quando, per esempio, $n - 1$ canali “esauriscono” la coda di pacchetti assegnati, a quel punto i pacchetti rimanenti del ciclo vengono mandati in modo consecutivo solo sul canale n .

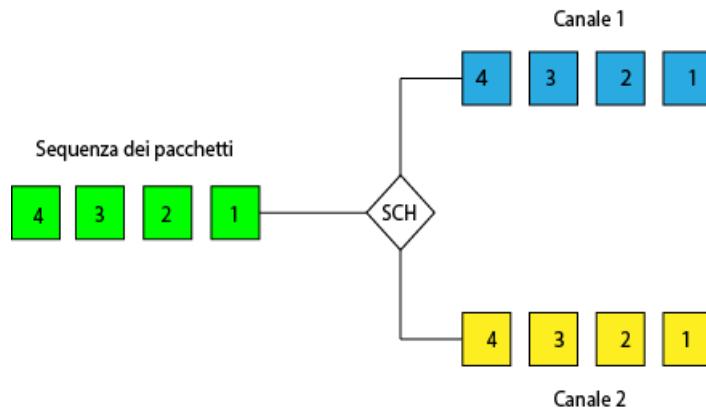


Figura 4.4: Esempio di Interleaving Weighted Round-Robin con ripetizione di pacchetti

E' stata infine sviluppata anche una variante di questo tipo di scheduler, detta *repeat-IWRR* (rIWRR), che ha lo scopo di replicare i pacchetti RTP in ingresso sui percorsi multipli in uscita, al fine di ridurre l'incidenza delle perdite di pacchetti sui canali radio. Questa aggiunta evita/limita l'utilizz-

zo del FEC, ovvero un meccanismo di correzione degli errori che consiste nell'aggiunta di simboli ridondanti (pacchetti RTP in questo caso) al flusso uscente in modo che il ricevente possa correggere in tutta autonomia possibili errori presenti senza chiedere la ritrasmissione del pacchetto in questione. Un esempio di questo scheduler è mostrato in figura 4.4.

Anche in questo caso, abbiamo dei tassi di ridondanza che determinano quante copie di pacchetti sono richieste in un ciclo. Il massimo numero di copie di un pacchetto è uguale al numero di canali implementati: questo è dovuto al fatto che un altro obiettivo dello scheduler è quello di non smistare pacchetti duplicati sullo stesso canale, poiché, se accadesse, qualora quel canale avesse dei problemi, si perderebbero tutte le copie del pacchetto rendendo così inutile la replicazione.

La terza e ultima funzione implementata sia per lo scheduler IWRR che per rIWRR è dovuta a realizzare la tecnica di multipath per RTP. Di fatto, un flusso RTP (*flow*) è identificato da un SSRC e dai numeri di sequenza dei singoli pacchetti RTP. Per implementare un sistema multipath è necessario creare n sotto-flussi (*subflow*) quanti sono i percorsi e generare per ciascuno di essi un sub-SSRC ed un numero di sequenza all'interno del header, in modo che nel receiver si possano ottenere delle statistiche relative a ogni sotto-flusso: a questo scopo è stato utilizzato l'header esteso di RTP al fine di memorizzare SSRC e numero di sequenza originali di ogni pacchetto, prima di essere inoltrato sul sotto-flusso assegnato dall'IWRR. Tale procedura viene invertita in ricezione quando viene ricreato il flusso: prima di passare nel modulo che fa da mixer dei flussi, SSRC e numero di sequenza vengono ripresi dal header esteso e ripristinati al loro posto nel header fisso. Così facendo è possibile eliminare gli eventuali duplicati e riordinare il flusso originale.

4.2.1 Implementazione

Gstreamer fornisce fra i suoi plug-in appartenenti alla libreria *bad* l'elemento *round-robin* che implementa in maniera molto semplice l'inoltro dei pacchetti in entrata verso n pad di uscita, in maniera ciclica dal primo all' n -esimo pad. Partendo da questa base, il plug-in è stato modificato fino a renderlo pesato con tassi variabili da pad a pad e dinamici nel tempo.

Proprietà

Sono state create delle proprietà, relative al plug-in, ovvero valori da fissare prima di mandare a run-time la pipeline tramite la funzione *set_property* in modo da impostare correttamente lo scheduler:

- specificare il numero n di path di uscita, utile quando è necessario allocare le varie strutture dati dello scheduler;

```
// salvo il numero di canali previsti
case PROP_NUM_SRC_PADS:
    disp->nsrcpads = g_value_get_int (value);
    break;
```

- specificare i pesi $w_1 \dots w_n$ che indicano la percentuale di pacchetti devono essere inoltrati su ciascuno degli n path;

```
case PROP_PYTHON_RATE:{  
    // numero di canali  
    nsrcpads = disp->nsrcpads;  
  
    if(!store_rate)  
        store_rate = g_new (gfloat, nsrcpads);  
  
    //memorizza nuovo peso  
    store_rate[count_rate] = g_value_get_float (value);  
    count_rate++;  
  
    // se sono memorizzati tutti i pesi, aggiorna  
    if(count_rate == nsrcpads) {  
        update = FALSE;  
        def = FALSE;  
        count_rate = 0;  
    }  
    break;  
}
```

- specificare i tassi $r_1 \dots r_n$ di ripetizione dei pacchetti: r_1 indica la percentuale di pacchetti singoli, r_2 la percentuale di pacchetti duplicati, r_n la percentuale di pacchetti replicati n volte.
-

```

case PROP_PYTHON_REPETITION:{
    // numero di canali
    nsrccpads = disp->nsrccpads;

    if(!store_repetition)
        store_repetition = g_new (gfloat, nsrccpads);

    //memorizza nuovo rate
    store_repetition[count_repetition] = g_value_get_float (value);
    count_repetition++;

    // se sono memorizzati tutti i pesi, aggiorno
    if(count_repetition == nsrccpads) {
        update = FALSE;
        def = FALSE;
        count_repetition = 0;
    }

    break;
}

```

Nel sender ogni volta che bisogna aggiungere un tasso in uno degli array, è necessario richiamare il metodo `set_property`: in sostanza sarà necessario chiamarlo, per ogni proprietà, tante volte quanto il numero di canali in uscita precedentemente settato.

Funzione Chain

La maggior parte dei plug-in in GStreamer contengono la funzione `chain`, richiamata ogni qual volta arriva un pacchetto dal pad di entrata, resettando tutte le variabili locali alla funzione (ma non quelle globali).

All'interno dello scheduler, in principio di questa funzione, è eseguito un controllo della correttezza dei valori inseriti:

- confronto del numero inserito di canali di uscita e numero di pad di uscita implementati nella pipeline, se sono diversi dà errore;
- controllo della somma dei valori percentuali, se diversa da 100 dà errore;
- qualora ci sia un tasso relativo a un canale pari a 100, i tassi vengono leggermente modificati in modo che non sia consentito che in nessun canale non passino pacchetti per tutta la durata del ciclo, poiché i test hanno rilevato problemi nei casi in cui alcuni canali venivano completamente ignorati.

Quando vengono inseriti tassi, il flag delle impostazioni di default viene modificato a “false”; quando invece non vengono settate le proprietà, questo flag rimane “true” e ciò permette al plug-in di impostare i pesi di default, in modo che in ogni canale passi la stessa quantità di pacchetti e che nessun pacchetto sia duplicato, trasformandosi in pratica in un classico scheduler di tipo Round-Robin.

Dopo tutti i controlli di correttezza, il plug-in si prepara a smistare i pacchetti di un ciclo: ogni ciclo è formato da W pacchetti in ingresso ($W = 100$), nonostante il numero di pacchetti in uscita potrebbe essere maggiore di W in presenza di pacchetti duplicati. In funzione dei tassi assegnati e del calcolo del numero di pacchetti totali, compreso i duplicati, sono fissati dei vincoli per i valori di smistamento affinché pacchetti duplicati non passino per lo stesso canale:

```

for(gint i = nsrccpads-1; i >= 0; i--) {
    m = disp->channel[j].pkt_count = ceil((disp->channel[j].rate)*ptot);
    disp->channel[j].jump = 100 - m;
    // settaggio vincoli
    min = round ((int)((disp->repetition[nsrccpads-1])*10000)/ptot)/100;
    max = round(10000 / ptot) / 100 + 0.01;
    // controllo vincoli
    if (disp->channel[j].rate < min || disp->channel[j].rate > max){
        constrains = TRUE;
        GST_LOG_OBJECT (disp, "Constrains not respected!");
        break;
    }
    j++; }
```

- il peso assegnato deve essere tale che il canale deve poter ricevere un numero minimo di pacchetti uguali a quelli da replicare n volte;
- il peso assegnato deve essere anche tale che il canale deve massimo ricevere 100 pacchetti e non di più, ovvero il numero di pacchetti in un ciclo: se il canale accogliesse più di 100 pacchetti, alcuni di essi sarebbero sicuramente dei duplicati, e ciò non dev'essere consentito.

Se i vincoli non sono rispettati, i tassi vengono settati in modo che in ogni canale passi la stessa quantità di pacchetti. Inoltre, assieme ai vincoli, sono calcolati il numero di pacchetti che devono passare attraverso un certo canale, e il numero massimo di pacchetti che lo scheduler può scegliere di non smistare in un dato canale.

Se sono rispettati i vincoli, lo scheduler stabilisce se il pacchetto è da replicare e quali canali selezionare per il passaggio del pacchetto e delle eventuali copie:

- ad ogni pacchetto in entrata, sono alternate le scelte di replicare n volte, $n - 1$ volte, ... o di non replicare il pacchetto, rispettando i tassi assegnati all'array di ripetizione;

```

restart:
  if(in < 0)
    in = nsrccpads-1;
  if(disp->pkt_rep[in] > 0) {
    disp->pkt_rep[in] -= 1;
    dup = in+1;
    in--;
  }
  else {
    in--;
    goto restart;
}

```

- dopo aver stabilito il numero di copie del pacchetto, vengono selezionati quei canali che lo scheduler deve obbligatoriamente scegliere per far passare il pacchetto in questione, in modo da rispettare i pesi assegnati.

```
j=0;
for(gint i = 0; i < nsrccpads; i++){
    if (num1 >= nsrccpads)
        num1 = 0;
    if(disp->channel[num1].jump<=0 && disp->channel[num1].pkt_count>0){
        a_jump[j] = num1;
        j++;
        c_jump1++;
        c_jump2++;
    }
    num1++;
}
```

- se necessario, saranno scelti ulteriori canali secondo la strategia *inter-leaved* dello scheduler, in modo da alternare sempre l'utilizzo dei canali, sempre in modo da rispettare i pesi assegnati a quest'ultimi.
-

```
j=0;
while(done < dup) {

    // se si devono selezionare altri canali
    if(dup - done > c_jump1){
        if (num1 >= nsrccpads)
            num1 = 0;
        if(disp->channel[num1].pkt_count <= 0) {
            num1++;
            continue;
        }
        else {
            num = num1;
            num1++;
        }
    }
}
```

```

else {
    num = a_jump[j];
    j++;
}

choose[num] = 1;
done++;
disp->channel[num].pkt_count -= 1;

// aggiorna seqnum del canale scelto
disp->channel[num].seqnum += 1;

for(gint i=0; i < c_jump2; i++)
    if(num == a_jump[i])
        c_jump1--;

buffer = gst_buffer_make_writable(buffer);
gst_rtp_buffer_map (buffer, GST_MAP_READWRITE, &rtp);

gst_rtp_buffer_set_extension(&rtp, TRUE);
if(done == 1) {
    data[0] = gst_rtp_buffer_get_ssrc (&rtp);
    data[1] = gst_rtp_buffer_get_seq (&rtp);
}
// aggiunge header esteso
gst_rtp_buffer_add_extension_onebyte_header(&rtp,5,data,sizeof(data));
// setta nuovo SSRC
gst_rtp_buffer_set_ssrc (&rtp,ssrc+num);
// setta nuovo seqnum
gst_rtp_buffer_set_seq (&rtp,(disp->channel[num].seqnum + 10000));

gst_rtp_buffer_unmap (&rtp);

// seleziona il pad corrispondente al canale scelto
src_pad = g_list_nth_data (pads, num);
if (src_pad)
    gst_object_ref (src_pad);
else
    continue;

```

```

// invia il pacchetto al pad corretto
ret = gst_pad_push (src_pad, gst_buffer_ref(buffer));
gst_object_unref (src_pad);

if(ret != GST_FLOW_OK)
    break;
}

```

Come si può notare nell'ultima sezione di codice, scelto il canale, prima di immettere il pacchetto sul pad di uscita corretto, viene esteso e modificato l'header RTP del pacchetto in questione: l'SSRC del flusso originale e il numero di sequenza del pacchetto vengono salvati nell'header esteso di 1 byte, così da essere ripristinati dal receiver quando i sotto-flussi vengono ricomposti. Al loro posto vengono sostituiti da un altro SSRC, differente da canale a canale, e un altro numero di sequenza, che salva la posizione del pacchetto nella sequenza passante per un determinato canale. A questo punto, il pacchetto passerà per la sotto-sessione RTP e quindi mandato nella rete attraverso un socket UDP.

4.3 Receiver

Il receiver (la prima parte della pipeline è mostrata in figura 4.5) accoglie tutti i flussi video creati nel sender da diverse porte del dispositivo e li fa filtrare prima nel plug-in che identifica la sessione RTP, *rtpsession*, e poi nel plug-in *rtpjitterbuffer*, il quale ha molteplici funzioni:

- elimina i possibili pacchetti duplicati;
- riordina l'ordine di sequenza dei pacchetti;
- fornisce le statistiche relative alla sessione, e quindi al canale, che poi saranno incapsulate in pacchetti RTCP.

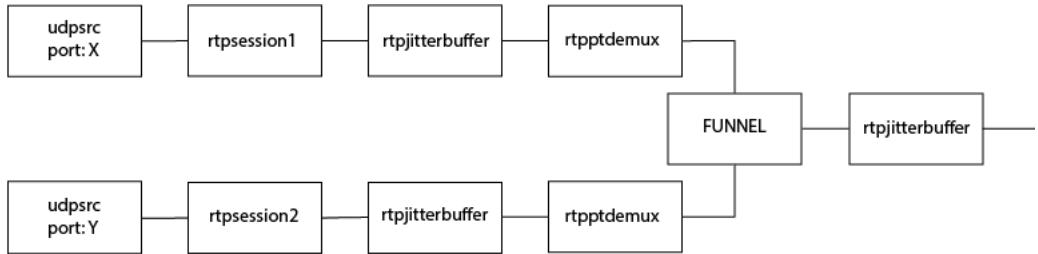


Figura 4.5: Pipeline Receiver - prima parte

Acquisite le statistiche, i pacchetti RTP devono essere ripristinati con il loro SSRC e del numero di sequenza originale, ricavandoli dal header esteso nel sender, al momento di smistarli nel canale stabilito. Per far ciò, è stato modificato il plug-in *rppptdemux*, che di base divide in più flussi i pacchetti che hanno un tipo di payload differente, ma, nel nostro caso, fornisce anche la funzione di ripristino dei campi RTP.

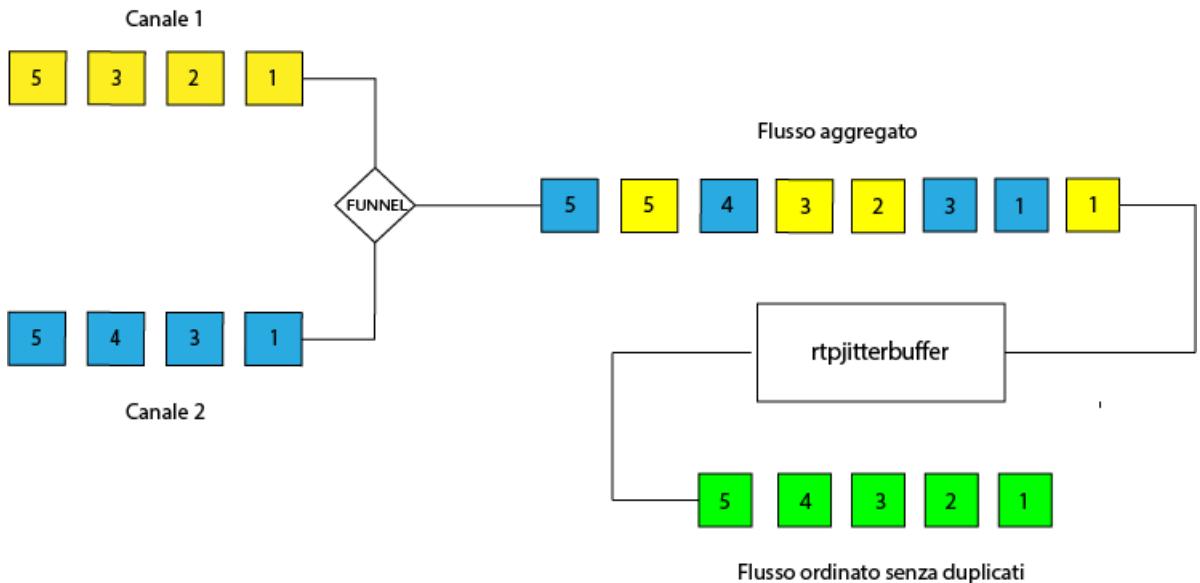


Figura 4.6: Aggregazione e riordinamento dei pacchetti del flusso video

A questo punto, tutti i diversi flussi in entrata vengono mischiati assieme dal plug-in *funnel* dai suoi numerosi pad di entrata e inoltrati al plug-in successivo da un solo pad di uscita. I pacchetti vengono filtrati da

rtpjitterbuffer, sistemando nuovamente la sequenza dei pacchetti a valle del *funnel*. Il funzionamento dei plug-in *funnel* e *rtpjitterbuffer* è mostrato in figura 4.6.

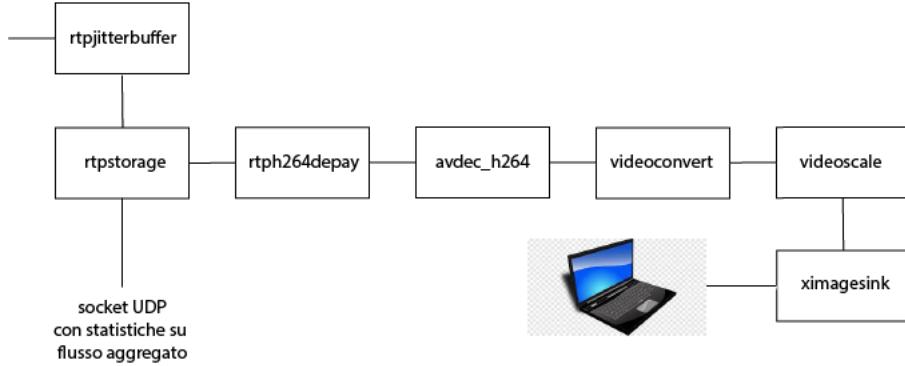


Figura 4.7: Pipeline Receiver - seconda parte

Riuniti i flussi, è necessario sapere quali pacchetti sono stati effettivamente persi al fine di misurare la qualità del servizio percepita a destinazione: per questo è stato modificato e inserito il plug-in *rtpstorage*, che di base aiuta altri plug-in adibiti alla correzione degli errori memorizzando i pacchetti da recuperare, ma, con la nostra modifica, crea anche una socket UDP allo scopo di mandare al sender il numero di pacchetti persi nel flusso aggregato, confrontando i numeri di sequenza dei pacchetti: se il numero di sequenza del pacchetto in questione non è il successivo di quello del pacchetto precedente già elaborato, allora il conteggio complessivo di pacchetti persi sarà incrementato.

Decapsulando i pacchetti dagli header RTP con il plug-in *rtph264depay* e decodificandoli con il plug-in *avdec_h264*, il video è pronto a essere visualizzato attraverso una finestra grazie al plug-in *ximagesink* (vedi figura 4.7).

4.4 Controller

Parallelamente ai canali UDP per il passaggio dei pacchetti RTP per il video, sia nel sender che nel receiver troviamo dei canali UDP per il passaggio delle statistiche dei canali sotto forma di pacchetti RTCP: il sender crea

dei pacchetti di tipo SR, uno per ogni sessione RTP, con i vari timestamp dei pacchetti, utili alla sincronizzazione dei messaggi, e li invia al receiver mentre quest'ultimo invia al sender le statistiche ricavate dal *rtpjitterbuffer* su pacchetti persi, jitter, RTT, e ancora altro.

Modificando il plug-in *rtpsession*, ogni volta che il sender riceve le statistiche su tutti i canali, inoltra alla funzione *Controller* il numero di pacchetti persi da ogni singolo canale.

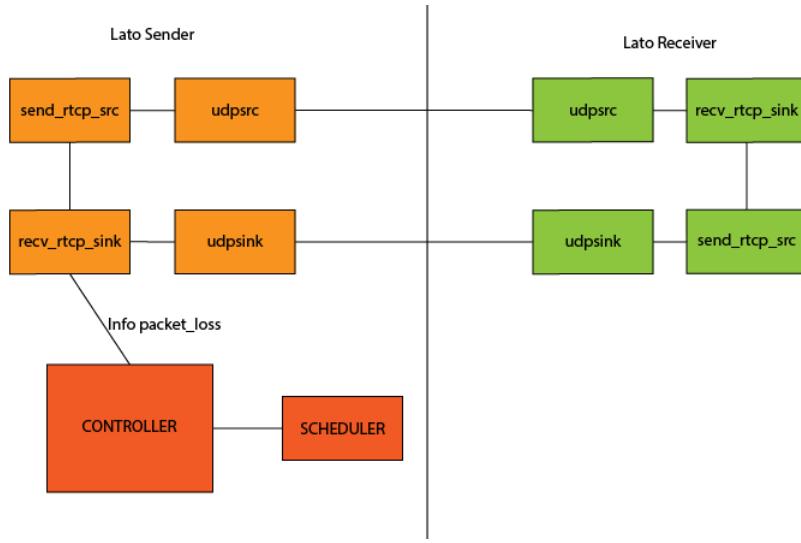


Figura 4.8: Loop di feedback RTCP

La funzione *Controller*, presente all'interno del sender, è una callback che viene chiamata ogni 3 secondi dal momento in cui è mandata in azione la pipeline: la sua funzione è quella di aggiustare i pesi dei canali e i tassi di ripetizione dei pacchetti in accordo con le statistiche dei percorsi, rendendo attive queste modifiche solo all'inizio di un ciclo di pacchetti nello scheduler. Questo significa che se, per esempio, un canale perde più di altri, sarebbe più efficiente diminuire il numero di pacchetti che dovrebbe passare per quel canale; oppure, se il numero di pacchetti persi complessivamente è alto, sarebbe meglio ridondare di più i singoli pacchetti.

Pertanto il controller ha bisogno delle statistiche sui pacchetti persi dai vari canali, informazione ricavata dai pacchetti RTCP entranti nel sender, e del numero di pacchetti persi dopo l'aggregazione dei flussi RTP, questa

invece ricavata tramite la socket UDP creata nel plug-in *rtpstorage* locata nel receiver.

Ottenute queste informazioni, si può passare quindi al calcolo dei nuovi tassi di ripetizione: per far ciò, come si può dedurre dalle tabelle seguenti, sono state create delle classi di efficienza, 4 nel caso della presenza di 2 canali e 7 nel caso di 3 canali, ognuna con un array di tassi di ripetizione prestabilito.

Caso 2 canali	
Classe di efficienza	Coding Rate
Classe A	[1.0, 0.0]
Classe B	[0.75, 0.25]
Classe C	[0.5, 0.5]
Classe D	[0.0, 1.0]

Caso 3 canali	
Classe di efficienza	Coding Rate
Classe A	[1.0, 0.0, 0.0]
Classe B	[0.75, 0.25, 0.0]
Classe C	[0.25, 0.75, 0.0]
Classe D	[0.0, 1.0, 0.0]
Classe E	[0.0, 0.75, 0.25]
Classe F	[0.0, 0.5, 0.5]
Classe G	[0.0, 0.0, 1.0]

Per passare da una classe di efficienza ad un'altra, viene confrontato il dato sulla perdita dei pacchetti del flusso aggregato: se questo è minore o uguale rispetto all'ultima rilevazione, si decrementa la classe di efficienza, ridondando meno i pacchetti; invece, se sono stati persi più pacchetti, si incrementa la classe di efficienza, ridondando di più. Chiaramente la peggior classe di efficienza (che è anche quella selezionata al momento di eseguire la pipeline) determina che ogni pacchetto viene ridondato tante volte quanto il numero di canali in uscita dallo scheduler, mentre la migliore (la classe A) è quella classe in cui nessun pacchetto viene duplicato.

Scelta la classe di efficienza per il prossimo ciclo di pacchetti, si passa a calcolare i nuovi pesi per i vari canali: se la classe scelta è quella peggiore, non è necessario calcolare pesi poiché tutti i pacchetti devono passare attraverso tutti i canali, quindi i pesi di smistamento sono impostati di default a $r_0 = 1/n$, dove n è il numero di canali; altrimenti, per stabilire i nuovi pesi per

ogni canale, si applica la seguente formula matematica, arrotondata fino alla seconda cifra decimale:

$$\alpha * r_0[i] + (1 - \alpha) * r_t[i]$$

dove:

- α , parametro di bilanciamento, è un numero compreso tra 0 e 1 fissato in base alla classe di efficienza;
- $r_t[i]$ il peso calcolato all'istante t come:

$$\frac{loss_i^{-1}}{\sum_{i=0}^{n-1} loss_i^{-1}}$$

in cui $loss_i$ è il numero di pacchetti persi dal canale i nell'ultima rilevazione.

Infine vengono salvati i dati correnti per realizzare il confronto con la prossima rilevazione e, tramite il metodo `set_property`, settati i nuovi valori di smistamento e ripetizione, che saranno resi ufficiali all'inizio del prossimo ciclo di pacchetti nello scheduler.

Capitolo 5

Test e grafici

I test sono stati eseguiti con un Raspberry Pi, dove vi era montata una telecamera, che inviava i pacchetti verso un portatile con il quale si poteva vedere ciò che la telecamera inquadrava in tempo reale. Il Raspberry è una unità facilmente installabile a bordo di un drone grazie al peso e ai consumi energetici contenuti, e al piccolo fattore di forma.

Per monitorare il comportamento dei valori dinamici durante l'esecuzione della pipeline, ho fatto uso di un'applicazione web, *Grafana*¹, che offre una visualizzazione in tempo reale mediante grafici lineari per l'analisi interattiva di dati. In breve, Grafana elabora dati presenti in un database, che viene aggiornato ogni volta che è richiamata la funzione Controller nel sender con i valori di cui vogliamo mostrare il comportamento dinamico nel tempo, e restituisce un tracciato.

Nelle immagini seguenti, sono mostrate la finestra del video e un'interfaccia di Grafana costituita da 4 grafici: quello in basso a destra mostra l'andamento del livello di ridondanza dei pacchetti, mentre gli altri 3 pannelli mostrano la variazione dei pesi assegnati a ciascun canale in accordo con le loro statistiche.

Le perdite sono state simulate mediante il plug-in di GStreamer *identity* che, per mezzo della proprietà *drop-probability*, mi permetteva di assegnare una percentuale di perdita di pacchetti per ogni canale.

¹<https://grafana.com/>

Test con perdite nulle

Il primo test realizzato (figura 5.1) mostra una situazione dove non c'è alcuna perdita di pacchetti: i pesi dei canali pertanto rimangono stabili, mentre il livello di ridondanza decrementa rapidamente fino ad assestarsi nella classe di efficienza migliore, ovvero quella dove non è duplicato alcun pacchetto.

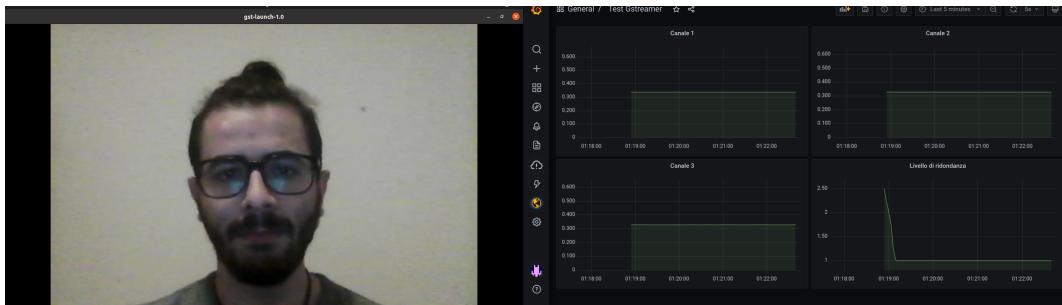


Figura 5.1: Test 1

Test con 1 canale difettoso

In un secondo test(figura 5.2), viene aggiunta una perdita pari al 10% al primo canale: come si può vedere dal grafico, il sistema di gestione dei pesi va a smistare più pacchetti video negli altri 2 canali, mentre la ridondanza oscilla fra valori bassi.

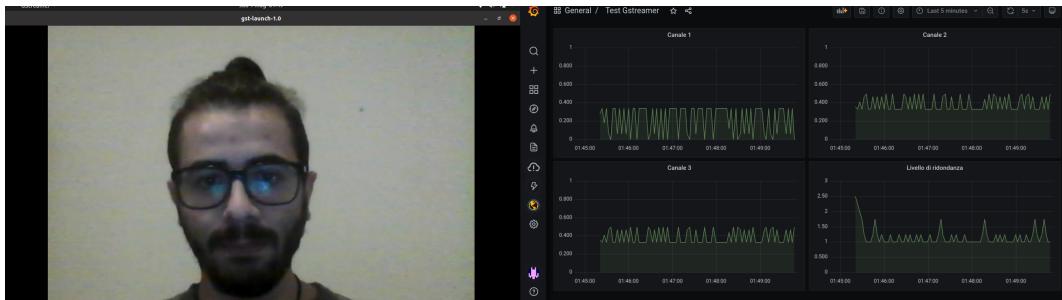


Figura 5.2: Test 2

Test con 2 canali difettosi

Nel terzo test effettuato (figura 5.3), vengono settate delle perdite sia al primo canale sia al secondo canale, rispettivamente del 10% e 1% dei pacchetti transitati. Il risultato è la preferenza, spesso totale, del sistema nell’usufruire del terzo canale, senza perdite. Il livello di ridondanza si mantiene medio.

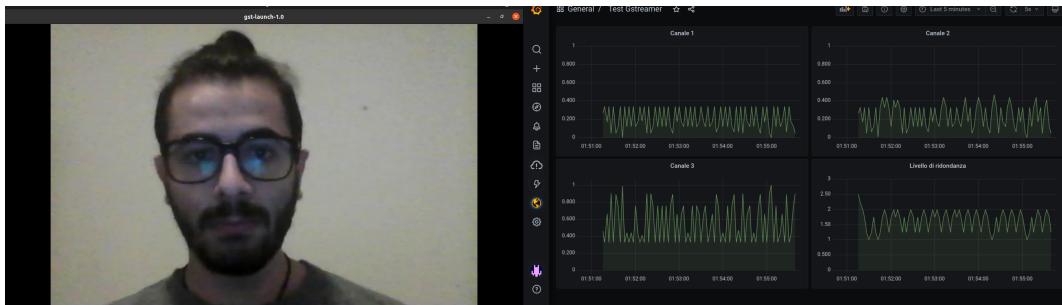


Figura 5.3: Test 3

Test con 3 canali difettosi

Nell’ultimo test mostrato (figura 5.4), si può subito notare come, assegnando delle perdite a tutti i canali, rispettivamente del 40%, del 10% e del 5%, il video inizia a presentare delle imperfezioni, quindi il sistema imposta una ridondanza massima dei pacchetti per tutto il tempo del test. Come detto nei capitoli precedenti, quando la ridondanza è massima, i pacchetti vengono smistati in egual peso in tutti i canali.

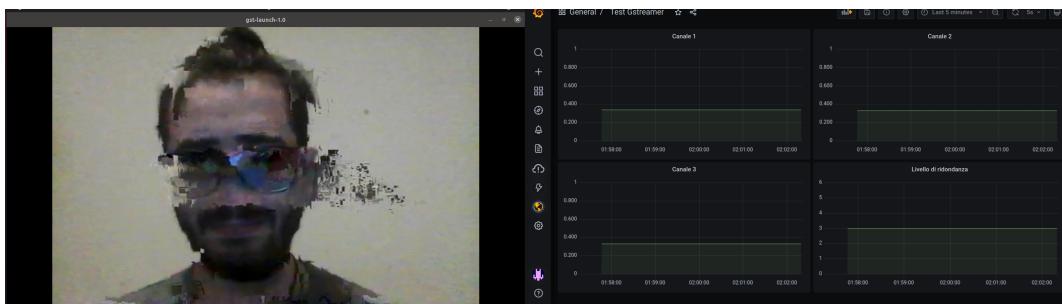


Figura 5.4: Test 4

Capitolo 6

Conclusione e lavori futuri

L'obiettivo di questo tirocinio è stato lo sviluppo di un sistema per la trasmissione di un flusso video da un drone a una stazione di comando su più canali di comunicazione. Il protocollo scelto per gestire un flusso in Real-Time è stato RTP, protocollo molto flessibile e adatto a moltissimi esperimenti in questo settore. È stato anche implementato uno scheduler dinamico che indirizzi i pacchetti video sui vari canali in modo efficiente in tempo reale: i canali con le migliori statistiche, e quindi minor perdite di pacchetti video, aumenteranno la loro frazione di flusso multimediale da ricevere in un ciclo dello scheduler. Per avere tali statistiche, è stato realizzato un loop di feedback tra sender e receiver facendo uso del protocollo RTCP. Sempre al fine di migliore l'affidabilità del sistema, è stata anche prevista una ridondanza dei pacchetti nei casi peggiori, ovvero quando i canali perdono contemporaneamente pacchetti video.

La piattaforma software scelta per implementare tutto il sistema è stata Gstreamer, poiché è usato in QGroundControl, software adibito al controllo completo del volo di un drone.

Guardando tutto il complesso sistema con occhio critico, noto come sia possibile apportare ancora diversi miglioramenti, come la creazione di plug-ins indipendenti per GStreamer che assolvono le funzioni da me implementate in plug-ins già esistenti a cui sono state apportate delle modifiche; l'implementazione di una sessione RTP per il flusso aggregato nel receiver al fine di ottenere statistiche più accurate mediante RTCP; l'aggiornamento delle classi di efficienza per il controllo della ridondanza dei pacchetti nei casi in cui vi siano 4 o più canali; sviluppo di algoritmi basati sull'intelligenza artificiale per il controllo dinamico dei pesi assegnati ai canali.

Durante questo tirocinio, ho dovuto affrontare molte sfide che sicuramente mi hanno fatto crescere e soprattutto imparare che c'è sempre un modo per superare anche quegli ostacoli che sembrano insormontabili. Alla fine di questo percorso potrò sfruttare la conoscenza di nuovi protocolli di rete, come RTP e RTCP, che non avevo mai trattato prima, e l'abilità di manipolare piattaforme open-source. Semmai dovesse tornare in questo settore, approfondirei tutta la teoria di base e le tecnologie riguardanti la trasmissione audio, che in questo tirocinio non è stata trattata.

Ringraziamenti

Alla fine di questo tirocinio, trovo doveroso menzionare tutte le persone che mi hanno accompagnato durante gli ultimi mesi e, in generale, durante tutto il mio percorso di studi universitari.

Innanzitutto ringrazio il Dott.Alberto Gotta, che mi ha assistito dal primo fino all'ultimo istante di questo tirocinio, cercando di incoraggiarmi nei momenti più duri, disponibile ad aiutarmi tra le difficoltà che ha presentato tale lavoro. Ringrazio anche i suoi collaboratori al CNR di Pisa, i dottori Manlio Bacco e Pietro Cassarà, e il dottorando Achilles Machumilane, che hanno contribuito allo sviluppo di questo progetto. Ringrazio il tutore accademico, la Prof.ssa Federica Paganelli, che mi ha fornito la sua esperienza per la stesura di questa tesi.

Ringrazio anche i miei colleghi di università e i miei attuali coinquilini, che in questi anni hanno giocato un ruolo importante tanto nella mia formazione quanto nell'allietare anche le giornate meno fortunate.

Infine, una menzione speciale va ai miei genitori, senza i quali nulla di tutto ciò avrebbe potuto essere realizzato: oltre all'immancabile sostegno materiale, non si sono mai stancati di fornirmi supporto morale e li ringrazio di aver sempre creduto in me in ogni momento.

Appendice A

Installazione del sistema

Per sfruttare il framework di GStreamer, è stata scaricata da Github la repository `gst-build` dal sito <https://github.com/Gstreamer/gst-build>, la quale contiene la versione 1.19 (allo stato attuale) di GStreamer. Per compilare tale build sono stati usati due sistemi di compilazione:

- `meson`¹: sistema di compilazione open-source alternativo a Cmake, usato per generare il makefile;
- `ninja`: sistema di compilazione alternativo a Make, usato per compilare file.

Installati questi 2 programmi, si procede con la compilazione di tutte le librerie Gstreamer eseguendo in sequenza tali comandi da terminale:

meson build

ninja -C build

dove `build` è la cartella contenente i file oggetto generati dalla compilazione (se non esiste, verrà automaticamente creata).

È possibile che, durante la compilazione, il sistema chieda di installare molteplici dipendenze di GStreamer mancanti.

Per eseguire le pipeline, è necessario creare un ambiente ad hoc nel quale poter richiamare tutti i plug-ins usati: per questo si esegue il comando *ninja -C build uninсталled*.

Sostituiti i plug-ins che sono stati modificati, e ricompilato il framework, si è pronti per eseguire le pipelines: prima si manda in esecuzione il receiver,

¹<https://mesonbuild.com/>

uno script bash, tramite il comando

./receiver.sh

quindi il sender, scritto in Python, tramite il comando

python3 sender.py

visualizzando così sul dispositivo ricevente il video catturato.

Appendice B

Codice Pipelines

B.1 sender.py

```
#!/usr/bin/python3

import sys
import gi
import os
import socket
gi.require_version('Gst', '1.0')
gi.require_version('GLib', '2.0')
gi.require_version('GObject', '2.0')
gi.require_version("Gtk", '3.0')
from gi.repository import GLib, Gst, Gtk, GObject

PATH_PLOSS = "path/ploss.txt"
UDP_IP = "127.0.0.1" # ip socket statistiche flusso aggregato
REM_IP = "127.0.0.1" # ip destinatario
UDP_PORT = 12000
channels = 3 #numero di canali
default = False
rate_0 = [0.34, 0.33, 0.33] # pesi iniziali
rep_0 = [0.0, 0.0, 1.0] # coding rate iniziale
index = 7 #indice classe di efficienza
alfa = 1
```

```

ploss_0 = [0] * (channels)
agg_0 = 0

def bus_call(bus, msg, *args):
    if msg.type == Gst.MessageType.EOS:
        print("End-of-stream")
        loop.quit()
        return
    elif msg.type == Gst.MessageType.ERROR:
        print("GST ERROR", msg.parse_error())
        loop.quit()
        return
    return True

#definizione classi di efficienza
def eff_class(ind):
    global alfa
    #caso 2 canali
    if (channels == 2):
        if (ind == 1):
            alfa = 0
            return [1.0, 0.0]
        elif (ind == 2):
            alfa = 0.25
            return [0.75, 0.25]
        elif (ind == 3):
            alfa = 0.5
            return [0.5, 0.5]
        elif (ind == 4):
            alfa = 1
            return [0.0, 1.0]
    #caso 3 canali
    elif (channels == 3):
        if (ind == 1):
            alfa = 0
            return [1.0, 0.0, 0.0]
        elif (ind == 2):

```

```

        alfa = 0.15
        return [0.75, 0.25, 0.0]
    elif (ind == 3):
        alfa = 0.35
        return [0.25, 0.75, 0.0]
    elif (ind == 4):
        alfa = 0.5
        return [0.0, 1.0, 0.0]
    elif (ind == 5):
        alfa = 0.65
        return [0.0, 0.75, 0.25]
    elif (ind == 6):
        alfa = 0.8
        return [0.0, 0.5, 0.5]
    elif (ind == 7):
        alfa = 1
        return [0.0, 0.0, 1.0]

def controller():
    print ("\n Leggo...\n")
    global index
    global ploss_0
    global agg_0
    ploss_1 = []    # array valori di packet loss dei canali
    sum_loss = 0
    rate = [] # array per nuovi pesi
    rep = [] # array per nuovi rate
    agg_1 = 0 # packet loss flusso aggregato

    #lettura dei vari packet loss dei canali
    try:
        with open (PATH_PLOSS, "r") as f:
            txt = f.readlines()
            for value in txt:
                ploss_1.append(int(value))
    except FileNotFoundError:
        print ("File .txt not found!")
        return True

```

```

# ricezione del valore di packet loss del flusso aggregato
try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((UDP_IP, UDP_PORT))
    data, addr = sock.recvfrom(1024)
    agg_1 = int.from_bytes(data, "little")
finally:
    sock.close()

#gestione del coding rate
if (agg_1 > agg_0): #confronto con dato precedente
    if ((channels == 2 and index != 4) or
        (channels == 3 and index != 7)):
        index += 1      # incremento classe di efficienza
else:
    if (index != 1):
        index -= 1      # decremento classe di efficienza

rep = eff_class (index)    # nuovi tassi di repetition

#gestione dei pesi
# se stiamo nella classe limite, metto rate a massima fairness
if ((channels == 2 and index == 4) or (channels == 3 and index == 7)):
    for i in range (channels):
        rate.append(round(1 / channels, 2))
else:
    #calcolo somma delle perdite
    for i in range (channels):
        s = ploss_1[i]-ploss_0[i]
        if(s <= 0):
            s = 10**-2
        sum_loss += (s**-1)

#nuovo peso
for i in range (channels):
    s = ploss_1[i]-ploss_0[i]
    if (s <= 0):

```

```

        s = 10**-2
        rt = (s**-1) / sum_loss
        rate.append (round (alfa*rate_0[i] + (1-alfa)*rt, 2))
# controllo valori dei pesi
if (sum(rate) != 1.0):
    i = rate.index(min(rate))
    rate[i] += 0.01

#salvo i dati per il prossimo ciclo
ploss_0 = ploss_1.copy()
agg_0 = agg_1

#settaggio pesi
print ("\n Setting rate...\n")
for value in rate:
    rr.set_property("python-rate", value)
    print(value)

#settaggio rate
print ("\n Setting repetition...\n")
for value in rep:
    rr.set_property("python-repetition", value)
    print(value)

return True

if __name__ == "__main__":
    # inizializzazione
    loop = GLib.MainLoop()
    Gst.init(None)
    # Crea pipeline senza nome
    pipeline = Gst.Pipeline()
    # Crea bus per i messaggi di errore
    bus = pipeline.get_bus()
    bus.add_watch(0, bus_call, loop)

#Creazione degli elementi

```

```


src = Gst.ElementFactory.make('v4l2src', None)

conv = Gst.ElementFactory.make('videoconvert', None)
caps = Gst.Caps.from_string("video/x-raw, format=I420")
camerafilter = Gst.ElementFactory.make("capsfilter", None)
camerafilter.set_property("caps", caps)
#H264
encoder = Gst.ElementFactory.make('x264enc', None)
encoder.set_property("tune", "zerolatency")
parse = Gst.ElementFactory.make('h264parse', None)
#rtp-payloader
pay = Gst.ElementFactory.make('rtph264pay', None)
pay.set_property("config-interval", 1)
#scheduler
rr = Gst.ElementFactory.make('roundrobin', None)
rr.set_property("srcpads", channels)
if(default == False):
    for value in rate_0:
        rr.set_property("python-rate", value)
    for value in rep_0:
        rr.set_property("python-repetition", value)
rrpad = Gst.Element.get_static_pad(rr, "sink")
rrpad1 = Gst.Element.get_request_pad(rr, "src_0")
rrpad2 = Gst.Element.get_request_pad(rr, "src_1")
rrpad3 = Gst.Element.get_request_pad(rr, "src_2")
queue1 = Gst.ElementFactory.make('queue', None)
queue2 = Gst.ElementFactory.make('queue', None)
queue3 = Gst.ElementFactory.make('queue', None)
qsrc1 = Gst.Element.get_static_pad(queue1, "src")
qsrc2 = Gst.Element.get_static_pad(queue2, "src")
qsrc3 = Gst.Element.get_static_pad(queue3, "src")
qsink1 = Gst.Element.get_static_pad(queue1, "sink")
qsink2 = Gst.Element.get_static_pad(queue2, "sink")
qsink3 = Gst.Element.get_static_pad(queue3, "sink")
#rtpsession
rtpsession1 = Gst.ElementFactory.make('rtpsession', None)
rtpsession2 = Gst.ElementFactory.make('rtpsession', None)

```

```

rtpsession3 = Gst.ElementFactory.make('rtpsession', None)
rtpsession1.set_property("srcpads", channels)
rtpsession2.set_property("srcpads", channels)
rtpsession3.set_property("srcpads", channels)
rtpsink1 = Gst.Element.get_request_pad(rtpsession1, "send_rtp_sink")
rtpsink2 = Gst.Element.get_request_pad(rtpsession2, "send_rtp_sink")
rtpsink3 = Gst.Element.get_request_pad(rtpsession3, "send_rtp_sink")
rtpsrc1 = Gst.Element.get_static_pad(rtpsession1, "send_rtp_src")
rtpsrc2 = Gst.Element.get_static_pad(rtpsession2, "send_rtp_src")
rtpsrc3 = Gst.Element.get_static_pad(rtpsession3, "send_rtp_src")
# udp
udpsink1 = Gst.ElementFactory.make('udpsink', None)
udpsink1.set_property("host", REM_IP)
udpsink1.set_property("port", 4000)
usink1 = Gst.Element.get_static_pad(udpsink1, "sink")
udpsink2 = Gst.ElementFactory.make('udpsink', None)
udpsink2.set_property("host", REM_IP)
udpsink2.set_property("port", 5000)
usink2 = Gst.Element.get_static_pad(udpsink2, 'sink')
udpsink3 = Gst.ElementFactory.make('udpsink', None)
udpsink3.set_property("host", REM_IP)
udpsink3.set_property("port", 7000)
usink3 = Gst.Element.get_static_pad(udpsink3, "sink")
#rtcp
rtcp_caps = Gst.Caps.from_string("application/x-rtcp")
udpsrc1 = Gst.ElementFactory.make('udpsrc', None)
udpsrc1.set_property("port", 4003)
udpsrc1.set_property("caps", rtcp_caps)
usrc1 = Gst.Element.get_static_pad(udpsrc1, "src")
udpsrc2 = Gst.ElementFactory.make('udpsrc', None)
udpsrc2.set_property("port", 5003)
udpsrc2.set_property("caps", rtcp_caps)
usrc2 = Gst.Element.get_static_pad(udpsrc2, "src")
udpsrc3 = Gst.ElementFactory.make('udpsrc', None)
udpsrc3.set_property("port", 7003)
udpsrc3.set_property("caps", rtcp_caps)
usrc3 = Gst.Element.get_static_pad(udpsrc3, "src")
rtcpsink1 = Gst.Element.get_request_pad(rtpsession1, "recv_rtcp_sink")

```

```

rtcpsink2 = Gst.Element.get_request_pad(rtpsession2, "recv_rtcp_sink")
rtcpsink3 = Gst.Element.get_request_pad(rtpsession3, "recv_rtcp_sink")
rtcpsrc1 = Gst.Element.get_request_pad(rtpsession1, "send_rtcp_src")
rtcpsrc2 = Gst.Element.get_request_pad(rtpsession2, "send_rtcp_src")
rtcpsrc3 = Gst.Element.get_request_pad(rtpsession3, "send_rtcp_src")
udpsink4 = Gst.ElementFactory.make('udpsink', None)
udpsink4.set_property("host", REM_IP)
udpsink4.set_property("port", 4001)
udpsink4.set_property('async', False)
udpsink4.set_property('sync', False)
usink4 = Gst.Element.get_static_pad(udpsink4, 'sink')
udpsink5 = Gst.ElementFactory.make('udpsink', None)
udpsink5.set_property("host", REM_IP)
udpsink5.set_property("port", 5001)
udpsink5.set_property('async', False)
udpsink5.set_property('sync', False)
usink5 = Gst.Element.get_static_pad(udpsink5, "sink")
udpsink6 = Gst.ElementFactory.make('udpsink', None)
udpsink6.set_property("host", REM_IP)
udpsink6.set_property("port", 7001)
udpsink6.set_property('async', False)
udpsink6.set_property('sync', False)
usink6 = Gst.Element.get_static_pad(udpsink6, "sink")

# Aggiunge elementi alla pipeline
pipeline.add(src)
pipeline.add(conv)
pipeline.add(camerafilter)
pipeline.add(encoder)
pipeline.add(parse)
pipeline.add(pay)
pipeline.add(rr)
pipeline.add(queue1)
pipeline.add(queue2)
pipeline.add(queue3)
pipeline.add(rtpsession1)
pipeline.add(rtpsession2)
pipeline.add(rtpsession3)

```

```

pipeline.add(udpsink1)
pipeline.add(udpsink2)
pipeline.add(udpsink3)
pipeline.add(udpsink4)
pipeline.add(udpsink5)
pipeline.add(udpsink6)
pipeline.add(udpsrc1)
pipeline.add(udpsrc2)
pipeline.add(udpsrc3)

# Collega fra loro tutti gli elementi
src.link(conv)
conv.link(camerafilter)
camerafilter.link(encoder)
encoder.link(parse)
parse.link(pay)
pay.link(rr)
# canale_1
Gst.Pad.link(rrpad1, qsink1)
Gst.Pad.link(qsrc1, rtpsink1)
Gst.Pad.link(rtpsrc1, usink1)
# canale_2
Gst.Pad.link(rrpad2, qsink2)
Gst.Pad.link(qsrc2, rtpsink2)
Gst.Pad.link(rtpsrc2, usink2)
# canale_3
Gst.Pad.link(rrpad3, qsink3)
Gst.Pad.link(qsrc3, rtpsink3)
Gst.Pad.link(rtpsrc3, usink3)
# rtcp_1
Gst.Pad.link(usrc1, rtcpsink1)
Gst.Pad.link(rtcpsrc1, usink4)
# rtcp_2
Gst.Pad.link(usrc2, rtcpsink2)
Gst.Pad.link(rtcpsrc2, usink5)
# rtcp_3
Gst.Pad.link(usrc3, rtcpsink3)
Gst.Pad.link(rtcpsrc3, usink6)

```

```

# Cambia valori a run-time
GLib.timeout_add(3000, controller)

# Esegui la pipeline
pipeline.set_state(Gst.State.PLAYING)
try:
    loop.run()
except Exception as e:
    print(e)

# Clean-up
pipeline.set_state(Gst.State.NULL)

```

B.2 receiver.sh

```

#!/bin/bash
./gststreamer/gst-build/build/subprojects/gststreamer/tools/gst-launch-1.0 -e \
    funnel name=f forward-sticky-events=false ! \
        rtpjitterbuffer ! rtppstorage ! rtph264depay ! h264parse ! \
            avdec_h264 ! videoconvert ! videoscale ! \
                video/x-raw, width=640, height=480 ! ximagesink sync=false \
    udpsrc port=4000 caps="application/x-rtp, media=(string)video, \
        clock-rate=(int)90000, encoding-name=(string)H264, payload=(int)96" ! \
            .recv_rtp_sink rtpsession name=ses1 probation=0 .recv_rtp_src ! \
                rtpjitterbuffer ! rtppdemux ! f. \
    udpsrc port=5000 caps="application/x-rtp, media=(string)video, \
        clock-rate=(int)90000, encoding-name=(string)H264, payload=(int)96" ! \
            .recv_rtp_sink rtpsession name=ses2 probation=0 .recv_rtp_src ! \
                rtpjitterbuffer ! rtppdemux ! f. \
    udpsrc port=7000 caps="application/x-rtp, media=(string)video, \
        clock-rate=(int)90000, encoding-name=(string)H264, payload=(int)96" ! \
            .recv_rtp_sink rtpsession name=ses3 probation=0 .recv_rtp_src ! \
                rtpjitterbuffer ! rtppdemux ! f. \
    udpsrc port=4001 caps="application/x-rtcp" ! ses1.recv_rtcp_sink \
    udpsrc port=5001 caps="application/x-rtcp" ! ses2.recv_rtcp_sink \
    udpsrc port=7001 caps="application/x-rtcp" ! ses3.recv_rtcp_sink \

```

```
ses1.send_rtcp_src ! udpsink host=127.0.0.1 port=4003 sync=false async=false \
ses2.send_rtcp_src ! udpsink host=127.0.0.1 port=5003 sync=false async=false \
ses3.send_rtcp_src ! udpsink host=127.0.0.1 port=7003 sync=false async=false
```

Bibliografia

- [1] *GStreamer: a flexible, fast and multiplatform multimedia framework.*
- [2] <https://gstreamer.freedesktop.org/artwork/>.
- [3] Overview · QGroundControl User Guide.
- [4] Behrouz A. Forouzan (Autore), Firoeuz Mosharraf (Autore), G. D'Angelo (Traduttore), and G. Maselli (Traduttore). *Reti di calcolatori. Un approccio top-down.*, chapter Multimedia e qualità del servizio. Pearson, 2013.
- [5] V. Ghini. RTP-RTCP. <http://www.cs.unibo.it/ghini/didattica/sistemi3/RTP-RTCP.pdf>, 2001.
- [6] P. Barisocchi. Progettazione dell'architettura di interconnessione tra rete wireless di tipo manet e rete satellitare, con relativa sperimentazione . Master's thesis, Università di Pisa, 2003.
- [7] H. Schulzrinne, Columbia University, S. Casner, Packet Design, R. Frederick, Blue Coat Systems Inc., and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. <https://tools.ietf.org/html/rfc3550>, 2003.
- [8] V. Singh, T. Karkainen, J. Ott, S. Ahsan, Aalto University, and L. Eggert. Multipath RTP (MPRTP). <https://tools.ietf.org/html/draft-singh-avtcore-mprtp-10>, 2014.