**Università degli Studi di Udine**

DIPARTIMENTO POLITECNICO DI INGEGNERIA E ARCHITETTURA

Corso di laurea in Ingegneria Elettronica

Tesi di laurea

# Studio e sviluppo di un sistema di controllo remoto di LIDAR su drone

Candidato:
**Alessandro Nadalutti**

Relatore:
**Prof. Pier Luca Montessoro**

**Anno Accademico 2021-2022**

# Sommario

Il problema considerato è lo sviluppo di un'applicazione che permetta di controllare e gestire il sistema di logging di un drone. Il drone è il modello DronutX1 ed è un *Inspection Drone*. Esso monta un sensore LIDAR che è la fonte principale dei dati, prodotti nella forma di diversi file dal peso considerevole. L'applicazione sviluppata deve quindi essere in grado di controllare la partenza e la terminazione dell'acquisizione dei dati e inoltre deve permettere il trasferimento dei dati dal drone a un'altra macchina durante la generazione. La natura del problema impone lo sviluppo di un'applicazione di rete Client-Server.

Per prima cosa è stato definito il protocollo ad alto livello per la comunicazione, quindi sono state definite tutte le primitive che il Server (il drone) deve fornire per permettere l'implementazione di tutte le funzionalità necessarie. La caratteristica principale del protocollo è stata quella d'impedire la cancellazione dei file dal Server senza che ci si la sicurezza che questi ultimi siano stati copiati correttamente dal Client.

Un altro aspetto che è stato considerato è stato l'uso della compressione per diminuire il più possibile la quantità di dati effettivamente da trasferire.

Tutto lo sviluppo è stato fatto in ambiente simulato, grazie a un programma che genera dei dati copiando quelli acquisiti in una sessione di logging reale.

Tutti i programmi sono stati scritti in linguaggio C.

# Indice

# Introduzione

L'obiettivo di questa tesi sperimentale è quello di studiare e implementare un sistema di controllo remoto del sistema di logging del sensore LIDAR di un drone. Il drone in oggetto è il modello DronutX1, prodotto dell'azienda statunitense Cleo Robotics. Esso utilizza un'applicazione di logging per ottenere i dati del sensore LIDAR, che corrispondono a un insieme di punti discreto nello spazio. I punti possono essere utilizzati per costruire un'immagine 3D. Il drone è infatti un *Inspection Drone*, usato per effettuare delle rilevazioni, anche mediante una videocamera, in spazi pericolosi o difficilmente accessibili. Lo scopo del progetto è quindi quello di creare un sistema che permetta di far partire e fermare il sistema di logging e che permetta inoltre di trasferire i dati dal drone a un computer durante il volo.

Nel primo capitolo di questa Tesi saranno discussi i problemi da risolvere e saranno quindi definiti tutti gli obiettivi da raggiungere. Nel secondo capitolo verrà descritta ad alto livello l'architettura dell'applicazione. Il terzo capitolo sarà dedicato ad alcuni aspetti preliminari dell'implementazione, mentre il quarto e quinto capitolo saranno dedicati al dettaglio dell'implementazione dell'applicazione di rete. L'ultimo capitolo sarà dedicato all'utilizzo della compressione, tecnica utile a diminuire la quantità di dati da trasferire.

# Capitolo 1

# Descrizione del problema

## 1.1 Caratteristiche del drone DronutX1

Il drone DronutX1 (figura 1.1) è stato sviluppato dall'azienda Statunitense Cleo Robotics. La dotazione comprende una telecamera, usata per la navigazione, e diversi sensori, fra i quali un sensore LIDAR. Si nota l'assenza di un sensore GPS, infatti il drone è pensato per operare in spazi confinati e in assenza di copertura GPS.

Figura 1.1: Vista frontale del DronutX1

Un altro aspetto, come si può notare guardando la figura 1.1 è l'assenza di eliche esposte, fattore che rende il velivolo più sicuro vicino alle persone e resistente alle collisioni con muri e oggetti a velocità inferiori a 1 Km/h, stando al manuale fornito dal produttore.

Il sensore LIDAR può essere inoltre usato per raccogliere dati riguardanti l'ambiente, che poi possono essere usati per ricostruire un'immagine 3D grazie alla Computer Vision.

Grazie a tutti questi fattori può essere definito *Inspection Drone*.

### 1.1.1 LIDAR

LIDAR, acronimo di *Laser Imaging Detection and Ranging*, è una tecnica di telerilevamento basata sull'invio d'impulsi laser e misura del tempo impiegato dal raggio per tornare alla sorgente. In questo modo è possibile determinare la distanza di un oggetto. Nell'ambito del drone, questa tecnica può essere utilizzata per effettuare delle scansioni continue dell'ambiente circostante mediante l'indirizzamento di raggi laser, ottenendo un insieme discreto di punti con le loro coordinate (figura 1.2).

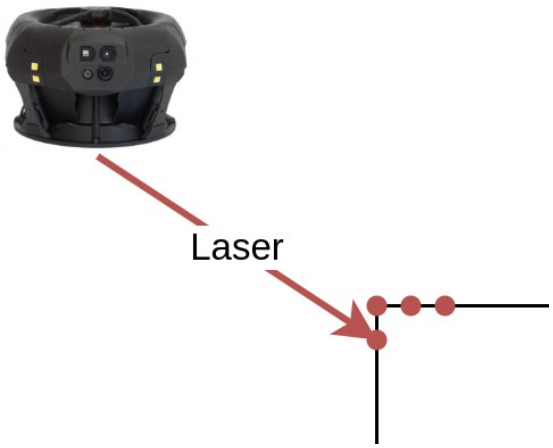


Figura 1.2: Esempio di scansione dell'ambiente mediante LIDAR

### 1.1.2 Hardware e sistema operativo

La connessione al drone avviene tramite Wi-Fi, con un range dichiarato dal produttore di poco meno di 1 km. Il drone funge da *access point* al quale connettere ad esempio uno smartphone, che tramite un'applicazione diventa il telecomando.

Il dispositivo monta inoltre un sistema operativo *Unix*, in particolare una versione modificata del sistema *Linux Yocto*, molto usato nei sistemi IoT ed embedded.

Grazie alla connessione Wi-Fi e alla natura del sistema operativo è quindi possibile connettere un computer al drone tramite *Secure SHell* (SSH) collegandosi al suo indirizzo ip, 192.168.8.1, in modo da ottenere un terminale con il quale interagire con ogni aspetto del sistema.

Il trasferimento di file può avvenire tramite il protocollo *Secure Copy* (SCP), largamente utilizzato.

## 1.2 Il sistema di logging

Esaminando i programmi d'inizio e terminazione del sistema di logging forniti dal produttore e riportati rispettivamente nei codici 1.1 e 1.2, si evince che il drone utilizza una versione del programma *voxl-logger*, modificata al fine di permettere di raccogliere i dati dal sensore LIDAR montato sul drone.

Codice 1.1: start.sh

```
#!/bin/bash
rm -rf /data/voxl-logger
./voxl-logger --point_cloud &
./voxl-logger --vio vvpx4_body_wrt_local &
```

Codice 1.2: stop.sh

```
#!/bin/bash
killall -9 voxl-logger
```

### 1.2.1 voxl-logger

Il programma `voxl-logger`, sviluppato da ModalAi, serve a salvare i dati prodotti dalle architetture *Modal Pipe Architecture* (MPA). Si assume quindi che il drone utilizzi la piattaforma *VOXL*, anche se non menzionata nelle risorse del produttore. La piattaforma ha bisogno di utilizzare un meccanismo di *Inter-Process Communication*, la MPA, basata sulle *POSIX pipes*. Le diverse componenti che compongono il sistema hanno infatti un costante bisogno di scambiare dati, ad esempio un sistema di guida autonomo ha bisogno dei dati di videocamere e sensori per raggiungere il suo scopo.

Il `voxl-logger` permette quindi di leggere direttamente i dati provenienti dalle pipes (e quindi dai sensori) e permette di salvarli sul disco fisso.

### 1.2.2 Dati raccolti dal drone

Il modo in cui si inizia o termina la raccolta dei dati è quello di connettersi al drone tramite SSH per avviare manualmente gli script 1.1 e 1.2. L'operazione è chiaramente scomoda, infatti uno degli obiettivi di questa tesi è rendere questo processo più semplice e veloce, ad esempio con un'interfaccia che permetta di eseguire direttamente gli script.

Guardando il codice 1.1, si può notare che vengono attivate due diverse istanze del logger.

La prima richiama l'opzione `--point-cloud`, aggiunta dal produttore in quanto non presente nella versione originale del logger, e si riferisce ai dati provenienti dal sensore LIDAR, che sono sostanzialmente un insieme di punti nello spazio. Il formato di questi file verrà trattato nella sezione 1.2.3.

La seconda istanza, chiamata con l'opzione `--vio vvpx4_body_wrt_local` si riferisce a misure della IMU, *Inertial Measurement Unit*, dispositivo che è formato da sensori quali accelerometri e giroscopi. La tabella 1.1 rappresenta un estratto del file `data.csv`, file in formato tabulare risultato del logger. Guardando quest'ultima è infatti possibile notare dati quali la posizione, il rollio (*roll*), il beccheggio (*pitch*) e l'imbardata (*yaw*). Il file risultante è un semplice file in formato tabulare di peso non eccessivo (minore di 1 MB) che non necessita particolari attenzioni per quanto riguarda, il trasferimento, infatti basterà trasferire il file al termine della sessione di logging.

Le due istanze scrivono i loro dati in due cartelle distinte (figura 1.3), sottocartelle della cartella `voxl-logger`, chiamate `log0000` e `log0001`. Il numero della cartella dipende da quale istanza inizia per prima la scrittura dei dati, sarà quindi necessario recuperare le informazioni su dove sono ubicati i diversi tipi di dati.

Tabella 1.1: Estratto del file relativo ai dati raccolti dall'IMU

| frame_id | timestamp(ns) | T_imu_wrt_vio_x(m) | T_imu_wrt_vio_y(m) | T_imu_wrt_vio_z(m) | roll(rad) | pitch(rad) | yaw(rad) |
|---|---|---|---|---|---|---|---|
| 0 | 93967399251 | 0.0044 | -0.0005 | -0.1023 | 0.0239 | 0.0336 | 0.0177 |
| 1 | 93993735706 | 0.0045 | -0.0006 | -0.1023 | 0.0242 | 0.0337 | 0.0219 |
| 2 | 94028864205 | 0.0048 | -0.0004 | -0.1019 | 0.0246 | 0.0338 | 0.0267 |
| 3 | 94064002940 | 0.0051 | 0.0002 | -0.1011 | 0.0255 | 0.0338 | 0.0298 |
| 4 | 94099101122 | 0.0054 | 0.0010 | -0.1002 | 0.0257 | 0.0343 | 0.0319 |
| 5 | 94134255899 | 0.0059 | 0.0021 | -0.0992 | 0.0260 | 0.0344 | 0.0332 |

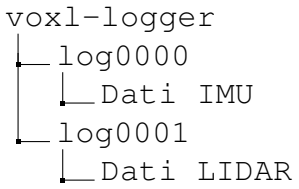


Figura 1.3: Esempio delle cartelle generate dalle due istanze del voxl-logger

I file della IMU e dei file del LIDAR saranno combinati, attraverso un eseguibile Matlab fornito dal produttore, in modo da ottenere una ricostruzione tridimensionale dello spazio ispezionato dal drone.

Entrambe le istante producono un altro file, chiamato `info.json`, riportato nel codice 1.3 (file relativo ai dati dell'IMU).

Codice 1.3: File info.json (dati IMU)

```
{
    "log_format_version":       1,
    "note":     "na",
    "n_channels":       1,
    "start_time_monotonic_ns":  93927993088,
    "start_time_date":  "1970-01-01␣00:01:35",
    "duration_s":       273.342006042,
    "channels": [{
            "channel":  0,
            "n_samples":        8027,
            "n_to_skip":        0,
            "type":     3,
            "type_string":      "vio",
            "pipe_path":        "/run/mpa/vvpx4_body_wrt_local/"
        }]
}
```

Nel file sono riportate alcune informazioni sulla sessione di logging, ad esempio il timestamp d' inizio, la durata in secondi e il numero di campioni. I file sono di peso esiguo e basterà trasferirli al termine della sessione (come il file `data.csv`).

### 1.2.3 I dati prodotti dal LIDAR

L'albero delle cartelle generato è rappresentato in figura 1.4. In questo caso i dati del LIDAR vengono scritti nella cartella `log0001`. Vengono generati due tipi di file. Il primo è un file tabulare, anch'esso nominato `data.csv` e di cui si riporta un estratto nella tabella 1.2.

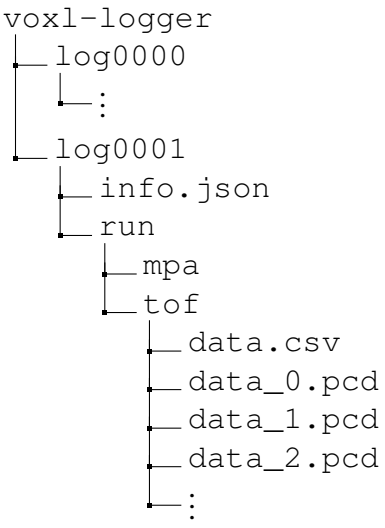


Figura 1.4: File generati dal logger (LIDAR)

Tabella 1.2: Estratto del file data.csv (file Point Cloud)

| frame_id | timestamp(ns) |
|---|---|
| 0 | 94260770431 |
| 1 | 94324318974 |
| 2 | 94391063921 |
| 3 | 94457491995 |
| 4 | 94525940692 |
| 5 | 94590910118 |

Si può notare che è composto solamente da un identificativo numerico e da un timestamp. Questo file viene aggiornato continuamente durante il volo, pertanto bisogna aspettare la fine della sessione prima di trasferirlo, ma è comunque di dimensioni esigue, quindi non presenta particolari criticità.

Il secondo tipo di file è quello più importante e critico per il progetto presentato in questa Tesi, infatti vengono creati una serie di file in formato PCD (Point Cloud Format), dei quali si riporta un estratto nel codice 1.4. Il numero di campioni presenti in `data.csv` è di poco inferiore al numero di file Cloud Point generati[1]. Si può quindi assumere che un file `.pcd` sia il risultato di una singola scansione, corrispondente a una riga del file `data.csv`. Esaminando inoltre la differenza dei timestamp, si può concludere che un file venga generato ogni circa 60 ms. La dimensione di questi file è variabile (vale in media circa 500 KB) e la sua dimensione dipende da quanti punti sono stati effettivamente generati. Dal codice 1.4 si nota infatti che sono indicati 38528 punti, ma questo numero non viene mai effettivamente raggiunto[2], probabilmente perché il sistema è troppo lento. Considerando comunque la dimensione media dei file

[1]Il motivo della disparità dei file non è ancora stato capito, tuttavia i file successivi che rappresentano una scansione sono in generale pochi, e ci sono un gran numero di file effettivamente vuoti. Una supposizione è che ci sia un ritardo fra la fine della scrittura dei due tipi di file.

[2]Ad esempio, un file che presenta una scansione effettiva di 35277 punti ha una dimensione di 671,1 kB, mentre un file con una scansione effettiva di 2773 punti ha una dimensione di 52,8 kB.

e considerando la generazione di un file ogni 60 ms, si ottiene una dimensione totale di circa 8 MB al secondo, e quindi di 500 MB al minuto.

Codice 1.4: Estratto del file data_0.pcd

```
VERSION .7
FIELDS x y z
SIZE 4 4 4
TYPE F F F
COUNT 1 1 1
WIDTH     38528
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS     38528
DATA ascii
1.340 -0.859 -1.282
1.398 -0.738 -1.307
1.287 -0.341 -1.154
1.285 -0.327 -1.152
1.287 -0.314 -1.151
1.298 -0.289 -1.159
1.304 -0.276 -1.163
1.232 -0.248 -1.098
```

Il problema nasce dal fatto che la dimensione della partizione di memoria più ampia del drone (`/data`) è di 16 GB e deve contenere altri dati come ad esempio le immagini della fotocamera.

Un altro aspetto da considerare è quello di dover scaricare i tutti i dati una volta concluso il volo. Il processo è quello di effettuare un trasferimento tramite SCP. Oltre alla scomodità dell'operazione in termini di tempo, va considerato che questa operazione risulta altamente impattante sulla batteria del dispositivo, quindi risulterebbe scomodo fare diverse sessioni di volo con conseguente immagazzinamento dei dati. L'obiettivo è quindi quello di, oltre che la possibilità di controllare facilmente l'attivazione o meno del logger, anche quello di trasferire i dati del LIDAR man mano che vengono generati, in modo da liberare memoria e semplificare l'operazione di trasferimento dei dati.

### 1.2.4 Salvataggio di sessioni multiple

Un ulteriore problema è che non è possibile salvare più sessioni di misure. Guardando il codice 1.1, si nota infatti che prima di eseguire i programmi di logging, viene eliminata la cartella contenente le misure precedenti. Questo problema rende ancora più scomoda l'acquisizione dei dati, infatti al fine di fare più misurazioni in sequenza si devono scaricare necessariamente i dati dal drone, con conseguente impatto sulla batteria. Un ulteriore obiettivo dell'applicazione è quindi quello di permettere di salvare più sessioni senza che sia necessario trasferire prima i dati.

# Capitolo 2

# Descrizione dell'applicazione

Le problematiche esposte nel capitolo 1 e gli obiettivi posti portano allo sviluppo di un'applicazione Client-Server.

## 2.1 Aspetti Generali

### 2.1.1 Architettura Client-Server

Il modello Clent-Server è il modello principale usato per le applicazioni di rete. Il modello prevede che le informazioni siano salvate su una macchina, detta Server, che ha il compito di fornire servizi ad altre macchine, dette Client. Una macchina Client infatti può inviare un messaggio alla macchina Server attraverso la rete. Quest'ultima recupererà dei dati, eseguirà del lavoro e invierà la risposta (il risultato) alla macchina Client che aveva fatto la richiesta. Il modello è quindi un modello Domanda-Risposta. Un esempio di questo modello è rappresentato in figura 2.1.

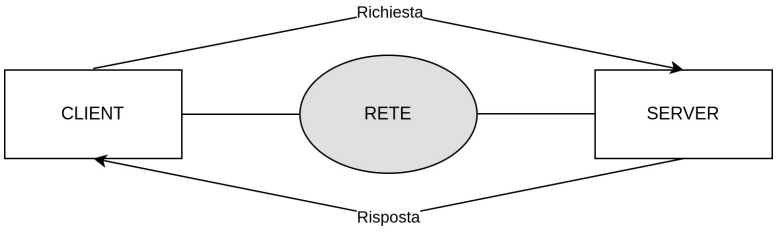


Figura 2.1: Architettura Client-Server con un solo Client e un Server

Dal punto di vista pratico, sia il Client che il Server sono dei processi in esecuzione, di solito su due macchine diverse interconnesse da una rete.

Nel caso in esame viene naturale pensare che il Server sia rappresentato dal drone, che al suo interno immagazzina i dati forniti dal logger. Il Client è invece una macchina connessa al drone tramite Wi-Fi e usata dall'operatore, con la quale è possibile richiedere al server di eseguire delle azioni, per esempio l'inizio della raccolta dati oppure il trasferimento di questi ultimi. Lo schema appena descritto è raffigurato in figura 2.2.


Figura 2.2: Architettura Client-Server reale

### 2.1.2 Il tipo di servizio

Dopo aver definito il modello dell'applicazione, il livello immediatamente inferiore nello stack di rete è il *livello di trasporto* dell'architettura *TCP/IP* (figura 2.3).


Figura 2.3: Stack di rete dell'architettura TCP/IP

In questo momento bisogna definire il tipo di servizio che si vuole implementare, quindi un servizio connesso/non connesso e affidabile/non affidabile.

Un servizio orientato alla connessione prevede che ci sia un *setup* iniziale (apertura della connessione), un periodo in cui c'è uno scambio di messaggi e infine la chiusura della connessione. In un tipo di connessione di questo genere è inoltre garantito l'ordine di arrivo dei pacchetti inviati e c'è anche un controllo di flusso della trasmissione.

Un servizio affidabile utilizza messaggi di riscontro per assicurarsi d'inviare completamente e correttamente tutti i pacchetti, eventualmente effettuando ritrasmissioni.

I servizi non connessi e affidabili, che sono più semplici, si usano solitamente quando non è strettamente necessario che i pacchetti arrivino tutti e in ordine, un classico esempio è lo streaming degli eventi in diretta. Si possono anche usare nel caso in cui la gestione delle ritrasmissioni sia gestita dal livello superiore, ovvero dall'applicazione.

Nel caso dell'applicazione descritta in questo capitolo, evidentemente si è scelto un servizio connesso e affidabile, infatti è fondamentale garantire la corretta trasmissione dell'interezza dei dati.

### 2.1.3 Le Socket

L'interfaccia fra il livello di trasporto e l'applicazione è fornita dalle *Socket*, le quali implementano un insieme di primitive che rendono disponibile un punto di accesso al servizio per i Client remoti.

Le Socket TCP forniscono un servizio connesso e affidabile, quindi, in accordo con quanto detto in sezione 2.1.2, devono fornire una serie di primitive:

**SOCKET:** creazione un punto finale di comunicazione

**BIND:** assegnazione di un indirizzo locale al socket

**LISTEN:** annuncio della capacità di accettare connesioni

**ACCEPT:** blocco fino all'arrivo di una connessione

**CONNECT:** tentativo di connessione

**SEND:** invio di dati sulla connessione

**RECEIVE:** ricezioni di dati sulla connessione

**CLOSE:** chiusura della connessione

Un esempio di connessione fra un Client e un Server in termini di primitive è mostrato in figura 2.4.

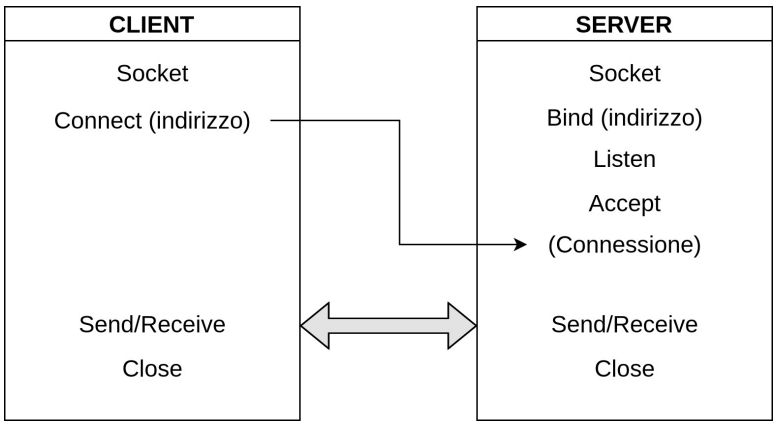


Figura 2.4: Esempio di connessione fra Client e Server in termini di primitive delle Socket

## 2.2 Il funzionamento dell'applicazione

### 2.2.1 Salvataggio dei dati in una cartella personalizzata

Va considerata una versione modificata dello script di start in quanto, al fine di poter salvare più sessioni di logging nella memoria del drone (sezione 1.2.4), occorre modificare lo script di start (codice 1.1) ottenendo il codice 2.1.

Codice 2.1: Versione modificata di start.sh

```
#!/bin/bash
if [ "$#" -ne 1 ]; then
    echo "Provide Timestamp"
    exit 1
fi

./voxl-logger -d $1 --point_cloud &
./voxl-logger -d $1 --vio vvpx4_body_wrt_local &
```

L'opzione `-d` data al `voxl-logger` permette infatti d'indicare la cartella dove salvare i dati. Si può quindi comunicare al Server il nome della cartella (mediante la primitiva START). Il nome corrisponde al *timestamp* dell'invio del comando, quindi avrà la forma `20230110-120000`. La stringa sarà poi passata come argomento allo script di start. Si noti anche che la nuova versione dello script non procede a eliminare nessuna cartella, in contrasto con la versione originale che elimina la cartella relativa al logging prima dell'avvio. Il risultato è che a ogni lancio dello script di start, verrà creata una cartella, nominata con il *timestamp* d' inizio della procedura di logging, come mostrato in figura 2.5.

```
voxl-logger
├── 20230120-153000
├── 20230120-153143
├── 20230120-153356
├── 20230120-153503
└── 20230120-153811
```

Figura 2.5: Esempio di diverse cartelle relative a diverse sessioni di logging salvate nel drone

### 2.2.2 Le primitive di controllo

Il primo aspetto da considerare nello sviluppo dell'applicazione è quello di permette di eseguire gli script (codice 2.1 e 1.2) relativi all'inizio e termine del sistema di logging. Sono state definite tre primitive di controllo che il Server (drone) deve fornire:

**STATUS:** ritorna lo stato del sistema di logging (attivo/non attivo) (figura 2.6)

**START:** se il sistema di logging non è già attivo, esegue lo script di start (figura 2.7)

**STOP:** se il sistema di logging è attivo, esegue lo script di stop (figura 2.8)

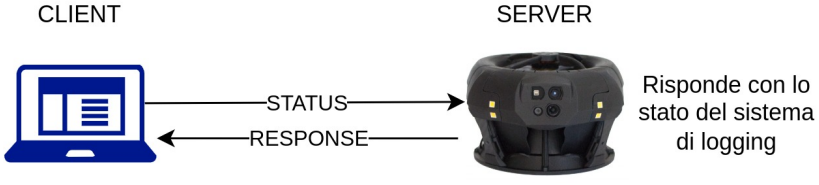


Figura 2.6: Esempio di richiesta dello stato


Figura 2.7: Esempio di richiesta di attivazione del sistema di logging

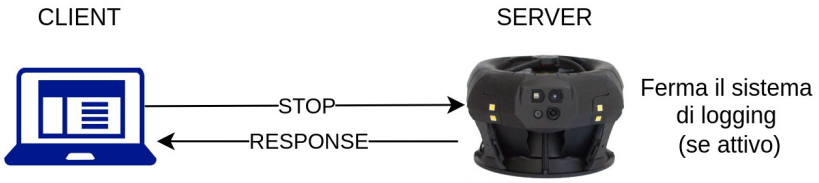


Figura 2.8: Esempio di richiesta di fermare il sistema di logging

### 2.2.3 Le primitive di trasmissione

L'aspetto più complesso è quello della gestione della trasmissione dei file relativi al Point Cloud dal Server al Client durante la generazione degli stessi. Si vuole avere la sicurezza di eliminare i file dal drone solo dopo che il Client ha scritto correttamente e senza errori il file sul suo disco fisso. La soluzione a questo aspetto è quella di far gestire interamente al Client la richiesta d'invio e cancellazione dei file, in modo che sia sicuro che il Client richieda al Server di eliminare un file solo dopo l'avvenuto salvataggio. Un altro aspetto del protocollo è la richiesta del file da inviare. Si è scelto di lasciare al server la scelta del file, in particolare verrà scelto il primo file disponibile (il più vecchio) nella cartella dei log meno recente. In questo modo verrà sempre inviato il file meno recente, indipendentemente dallo stato iniziale del Client e del Server. Servirà anche un'altra primitiva che permetta di richiedere uno specifico file al Server, utile per trasferire i file mancanti (a fine trasmissione) (sezione 1.2.2). Un'ulteriore primitiva è necessaria per richiedere al Server di eliminare l'intera cartella relativa alla trasmissione quando quest'ultima è terminata.

Le primitive di trasmissione del Server sono quindi:

**SEND_NEXT:** invio del file meno recente nella cartella meno recente. Si possono presentare i casi seguenti:

1. **SENDING:** il file è stato trovato e si procede a inviarlo. Vengono anche trasmesse delle informazioni utili alla trasmissione (come ad esempio il numero del file e la sua dimensione) che verranno trattate in seguito (figura 2.9)
2. **WAIT:** non sono stati trovati file ed è presente una sola cartella (la più recente). Inoltre il sistema di logging è attivo (dovrebbero arrivare dei nuovi file a breve) (figura 2.10)
3. **EMPTY:** non sono stati trovati file nella cartella e (figura 2.10)
   - non si è nella cartella più recente (quindi ci sono più cartelle e la cartella corrente non riceverà nuovi file)

- si è nella cartella più recente (è presente solamente una cartella) e il sistema di logging non è attivo

4. **NO_TR:** non ci sono cartelle (e quindi file) da inviare (figura 2.12)

**REMOVE_FILE:** rimuove il file indicato. La primitiva sarà sempre chiamata successivamente alla SEND_NEXT, quindi la rimozione sarà relativa al file appena scritto (correttamente) in memoria (figura 2.13)

**SEND_FILE:** invio del file indicato. La primitiva sarà chiamata solo dopo una ricezione di EMPTY, risultato della chiamata alla SEND_NEXT e sarà usata per ottenere i file specifici rimanenti (a questo punto non ci sono più file di Point Cloud da inviare) (figura 2.14)

**REMOVE_DIR:** rimuove la cartella indicata, quindi la primitiva verrà sempre chiamata al termine della corretta ricezione di tutti i file rimanenti (ottenuti tramite SEND_FILE). Al termine dell'esecuzione con successo la trasmissione relativa alla specifica cartella può ritenersi conclusa (figura 2.15)

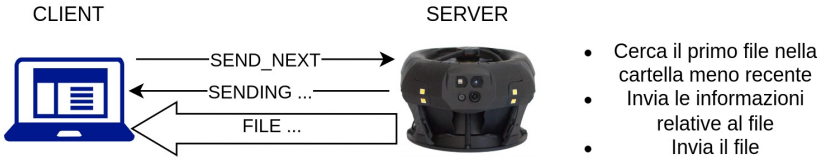


Figura 2.9: Esempio di richiesta del prossimo file con invio del file


Figura 2.10: Esempio di richiesta del prossimo file con risposta di attendere


Figura 2.11: Esempio di richiesta del prossimo file quando i file (nella cartella) sono terminati


Figura 2.12: Esempio di richiesta del prossimo file quando i file sono terminati


Figura 2.13: Esempio di richiesta di eliminazione di un file

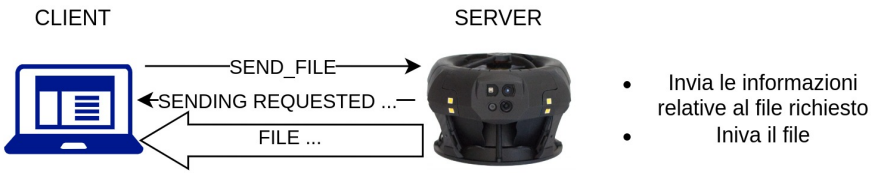


Figura 2.14: Esempio di richiesta d'invio di un file specifico


Figura 2.15: Esempio di richiesta di eliminazione di una cartella specificata

## 2.3 Il ruolo del Client

Come già detto in sezione 2.2.3, il ruolo del Client è fondamentale, infatti esso avrà il compito di gestire interamente la trasmissione dei file, quindi deve richiedere continuamente nuovi dati e richiederne l'eliminazione dopo la scrittura. Deve essere in grado di aspettare un certo tempo e ripetere la richiesta nel caso di arrivo di un messaggio di WAIT. Deve essere in grado di capire quando una trasmissione è conclusa (messaggio di EMPTY) e quindi gestire la richiesta dei file aggiuntivi e, al termine, richiedere la rimozione della cartella relativa alla sessione di acquisizione.

Il processo di una trasmissione completa di una sessione può essere riassunto come in figura 2.16, dove si è omesso per semplicità il caso di gestione della risposta di WAIT, che corrisponde ad aspettare un certo tempo per poi ripetere la richiesta SEND_NEXT.

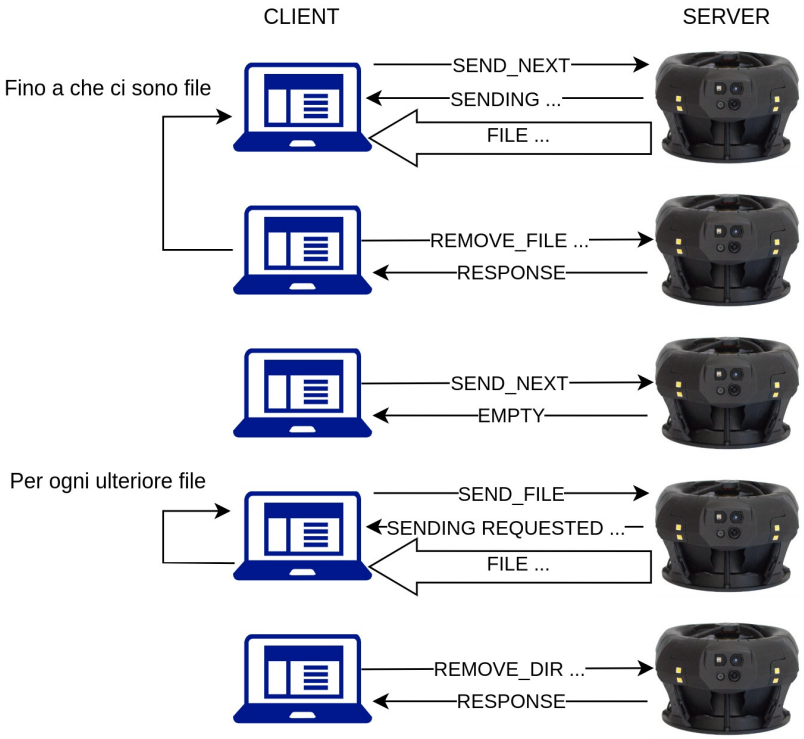


Figura 2.16: Esempio di trasmissione completa di una cartella (sessione di acquisizione)

## 2.4 Parallelismo

Le primitive di "controllo" e "trasmissione" descritte nelle sezioni 2.2.2 e 2.2.3 sono completamente indipendenti. Questo fatto apre alla possibilità di utilizzare due connessioni (due Socket) differenti per l'invio dei due tipi di comandi. Dal punto di vista pratico è necessario sviluppare i programmi Client e Server in maniera che sfruttino il parallelismo, rispettivamente usando il *Multithreading* e la generazione di *processi figli*. Il risultato è quello di avere di fatto due processi Client e due processi Server, che utilizzando due Socket distinti gestiscono i due aspetti dell'applicazione, come mostrato nella figura 2.17.

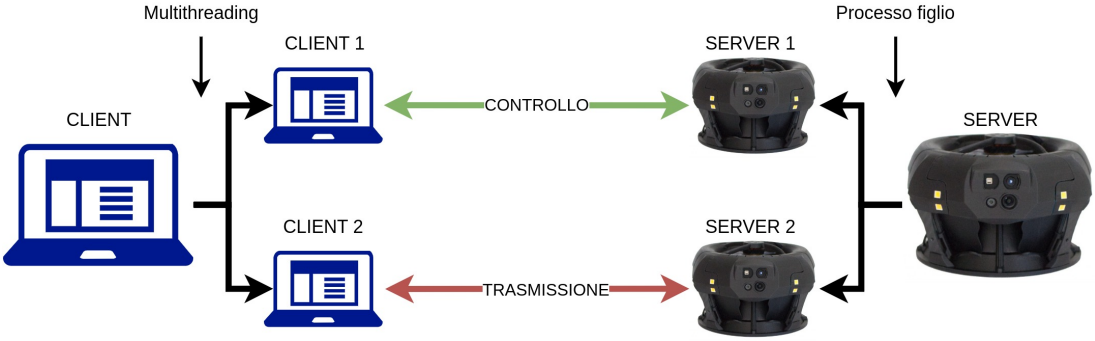


Figura 2.17: Client-Server paralleli

Il vantaggio di questo approccio è quello di poter immediatamente gestire un comando di controllo, ad esempio la richiesta di fermare il sistema di logging, anche nel mezzo di una trasmissione di un file. Un'applicazione sequenziale gestisce le richieste (i comandi) uno alla volta, quindi, nel caso di arrivo di un comando di STOP (o di START) nel mezzo di una trasmissione, sarebbe necessario aspettare la conclusione della gestione della primitiva dell'invio del file, con conseguente latenza nell'esecuzione, anche di qualche secondo nel caso di connessione non ottimale.

La parte principale dell'applicazione è quella relativa al controllo, infatti ha poco senso avere a disposizione solo il thread (processo) che si occupa della trasmissione, mentre ha senso avere a disposizione solo la parte di applicazione relativa al controllo. Si definisce quindi *main* thread (processo) quello relativo al controllo. Non è infatti necessario che entrambi i thread (processi) siano attivi contemporaneamente. In particolare, quello relativo alla trasmissione verrà generato solo al momento del bisogno (trasmissione in corso).

### 2.4.1 La primitiva CLOSE

Fin'ora è stata tralasciata una primitiva che permetta di porre fine alla comunicazione terminando correttamente le Socket. Questa primitiva sarà anche utile per terminare i thread (o processi) creati al fine gestire l'applicazione parallela (in accordo con quanto visto nella sezione 2.4). Si definisce la primitiva:

**CLOSE:** notifica del termine della trasmissione, al fine di permettere la chiusura adeguata del socket e il termine del thread (processo) (figura 2.18)


Figura 2.18: Esempio di terminazione della trasmissione

## 2.5 Riassunto delle primitive

Un riassunto di tutte le primitive descritte e della propria tipologia si trova nella tabella 2.1.

Tabella 2.1: Riassunto delle primitive

| Controllo | Trasmissione | Generale |
|---|---|---|
| STATUS | SEND_NEXT | CLOSE |
| START | REMOVE_FILE | |
| STOP | SEND_FILE | |
| | REMOVE_DIR | |

L'architettura è basata su un protocollo richiesta e risposta, quindi un qualsiasi tipo di errore avvenuto durante l'elaborazione del Server può essere comunicato nella risposta, ad esempio rispondendo con la stringa `ERROR` seguita dalla descrizione dell'errore. Il Client, ricevuto un errore, potrà gestirlo nel modo più opportuno.

## 2.6 Uso della compressione

La connessione al drone non è ottimale, infatti è una connessione wireless e quindi intrinsecamente più soggetta alla perdita e ritrasmissione dei pacchetti. L'access point Wi-Fi in questo caso è inoltre il drone stesso, che è in movimento, quindi la banda disponibile è molto variabile, senza contare il fatto che il volo può avvenire all'interno di edifici e luoghi chiusi, altro fattore che contribuisce alla degradazione del collegamento.

Un altro obiettivo che si può identificare è quindi quello di limitare la quantità di dati trasmettere il più possibile.

I file da inviare sono file testuali (file ASCII), quindi essi si prestano bene alla compressione. A titolo di esempio è riportato nel codice 2.2 il risultato dell'uso del programma linux `zip` su un file del Point Cloud, della dimensione di circa 500 kB.

Codice 2.2: Esempio di compressione di un file Point Cloud

```
$ zip data_0.zip data_0.pcd
adding: data_0.pcd (deflated 78%)
```

Dal risultato ci si può aspettare quindi una riduzione della dimensione dei dati da inviare di circa l'ottanta percento.

# Capitolo 3

# Aspetti preliminari dell'implementazione

In questo capitolo saranno analizzati alcuni aspetti preliminari, che precedono lo sviluppo delle applicazioni Server e Client, descritte in dei capitoli appositi. Tutti i programmi che saranno descritti in seguito sono stati scritti in linguaggio C. Il codice sorgente completo dell'applicazione può essere trovato nella repository GitHub: `https://github.com/uThings/Study-and-development-of-a-LIDAR-remote-control-system-on-a-drone`

## 3.1 La simulazione del sistema di logging

Una delle parti più importanti dello sviluppo dell'applicazione è stata quella di creare un programma che potesse replicare fedelmente il sistema di logging del drone. I file relativi a una reale sessione di logging saranno copiati da una cartella di riferimento, letta dal programma di simulazione. Per prima cosa il programma deve creare la cartella (l'albero di cartelle) dove salvare i dati, nominata come l'ingresso esterno, corrispondente al timestamp. L'albero di cartelle deve essere generato in maniera tale che sia casuale l'ubicazione dei dati Point Cloud (se generati nella cartella `log0000` o `log0001`). In seguito deve iniziare a generare i dati Point Cloud, uno ogni circa 60 ms, fintanto che il programma non viene fermato tramite un segnale esterno (lo script di stop (codice 1.2) invia un segnale al programma `voxl-logger` per terminarlo). Infine, prima di uscire, il simulatore deve copiare i file rimanenti (i file `data.csv` e `info.json`).

I dati nella cartella di riferimento devono essere organizzati come mostrato in figura 3.1.

### 3.1.1 La funzione main

Si riporta l'algoritmo 1 che rappresenta quanto detto in sezione 3.1.

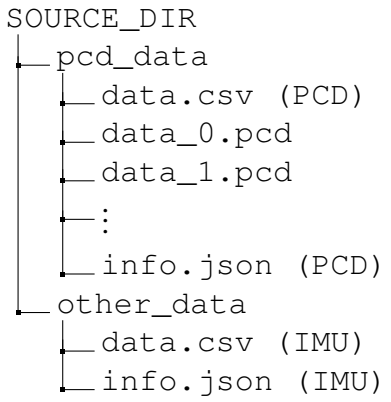


Figura 3.1: Organizzazione dei file nella cartella di riferimento per il voxl-logger simulato

**Algoritmo 1** Pseudocodice del programma di simulazione del logger

```
Ottiene il timestamp dall'argomento
Installa i gestori di segnali (per la terminazione)
Genera casualmente il numero della cartella dei dati pcd (log0000 o log0001)
Crea le cartelle per i dati Point Cloud
Crea le cartelle per i dati dell'IMU
while running do          ▷ il ciclo termina quando arriva un determinato segnale
    Copia il file successivo dalla cartella di riferimento
    Attende 55 ms
end while
Copia i file rimanenti
Fine
```

Nel codice 3.1 è riportata la funzione `main`.

Codice 3.1: Funzione main del programma di simulazione del voxl-logger

```
1  int main (int argc, char *argv[])
2  {
3      char in_path[MAX_PATH_LEN];
4      char out_path[MAX_PATH_LEN];
5      int cnt = 0;
6      char timestamp[TIMESTAMP_LEN];
7      int r, pcd_data_n, other_data_n;
8      int opt;
9
10     while ((opt = getopt(argc, argv, "d:")) != -1) {
11         switch (opt) {
12             case 'd': snprintf(timestamp, TIMESTAMP_LEN, "%s",
                   optarg); break;
13             case '?': printf("Must provide the -d option with a
                   timestamp (es: 20230102-083000)\n"); return
                   EXIT_FAILURE;
14             default: return EXIT_FAILURE;
15         }
16     }
17
18     if (optind == 1) {
```

```
19          printf("Must provide the -d option with a timestamp (es:
                20230102-083000)\n");  return EXIT_FAILURE;
20      }
21
22      struct sigaction s_handler;
23      s_handler.sa_handler = SigHandler;
24      sigaction(SIGINT, &s_handler, NULL);
25      sigaction(SIGQUIT, &s_handler, NULL);
26      sigaction(SIGKILL, &s_handler, NULL);
27      sigaction(SIGTERM, &s_handler, NULL);
28
29      srand(time(NULL));
30
31      /* generate the folder that contains pcd data */
32      r = rand() % 100;
33      if (r > 50) {
34          pcd_data_n = 1;
35          other_data_n = 0;
36      }
37      else {
38          pcd_data_n = 0;
39          other_data_n = 1;
40      }
41
42      snprintf(out_path, MAX_PATH_LEN, "%s/%s/voxl-logger/log%04d/
            run/mpa/tof", OUT_DIR, timestamp, pcd_data_n);
43      if (MakePath(out_path, 0777) != 0) {
44          fprintf(stderr, "Error creating path\n");
45          return EXIT_FAILURE;
46      }
47
48      snprintf(out_path, MAX_PATH_LEN, "%s/%s/voxl-logger/log%04d/
            run/mpa/vvpx4_body_wrt_local", OUT_DIR, timestamp,
            other_data_n);
49      if (MakePath(out_path, 0777) != 0) {
50          fprintf(stderr, "Error creating path\n");
51          return EXIT_FAILURE;
52      }
53
54      /* generation */
55      while (running) {
56
57          snprintf(in_path, MAX_PATH_LEN, "%s/pcd_data/data_%d.pcd"
                , SOURCE_DIR, cnt);
58          snprintf(out_path, MAX_PATH_LEN, "%s/%s/voxl-logger/log
                %04d/run/mpa/tof/data_%d.pcd", OUT_DIR, timestamp,
                pcd_data_n, cnt);
59
60          if (access(in_path, F_OK) != 0) {
61              sleep(1);
62              continue;
63          }
64
65          if (CopyFile(in_path, out_path) < 0) {
66              fprintf(stderr, "Unable to copy the file %s\n",
                    in_path);
```

```
67              return EXIT_FAILURE;
68          }
69
70          cnt++;
71
72          sleepms(DELAY_MS);
73      }
74
75      printf("Finished\n");
76
77      if (CopyOtherFiles(timestamp, pcd_data_n) < 0) {
78          fprintf(stderr, "Failed to copy other files\n");
79          return EXIT_FAILURE;
80      }
81
82      printf("\nCLOSING, written %d files\n", cnt);
83
84      return EXIT_SUCCESS;
85  }
```

La prima cosa fatta dal programma è quella di ottenere la stringa del timestamp tramite argomento obbligatorio da fornire al lancio del programma. Per questo scopo è stata usata la funzione `getopt`. Dopodiché il programma crea le cartelle del logger grazie alla funzione `MakePath` (sezione 3.1.2). La casualità della cartella dei dati Point Cloud è assicurata grazie all'uso della funzione `rand`. Dopodiché vengono installati dei gestori di segnali, infatti, come detto nella sezione 3.1, il programma deve essere fermato tramite un segnale esterno, che in questo caso è inviato dallo script di stop (codice 1.2). Il gestore si limita a impostare la variabile globale `running` a zero, permettendo così la terminazione del ciclo di generazione (copia) dei file. La copia dei file dalla cartella di riferimento avviene aumentando un contatore e controllando se il file esiste. La funzione `access` serve a questo scopo. Si noti che in caso il file non sia presente, e quindi non ci siano più file da copiare, il programma non termina ma fa ripartire il ciclo di generazione, creando di fatto un'attesa infinita. Si ricorda a tal proposito che si vuole la causare la terminazione del programma solo mediante segnali esterni. La copia del singolo file avviene grazie alla funzione `CopyFile` (sezione 3.1.3). Al termine della generazione vengono poi copiati i file rimanenti grazie alla funzione `CopyOtherFiles`.

### 3.1.2 La funzione MakePath

La funzione `MakePath` è riportata nel codice 3.2.

Codice 3.2: Funzione MakePath

```
1  int MakePath (const char path[], mode_t perm)
2  {
3      int ret, len;
4      int i = 0;
5      int j = 0;
6      char tmp[MAX_PATH_LEN];
7
8      len = strlen(path);
9
```

```
10      while (i < len) {
11
12          /* get the next folder name */
13          do {
14              tmp[j++] = path[i++];
15          } while (tmp[j-1] != '/' && tmp[j-1] != '\0');
16
17          tmp[j] = '\0';
18
19          /* try to creake dir (if it already exists it is not
                considered an error) */
20          ret = mkdir(tmp, perm);
21          if (ret != 0 && errno != EEXIST) {
22              return -1;
23          }
24      }
25
26      return 0;
27  }
```

La funzione serve a creare un percorso completo di cartelle a partire da una stringa. Ad esempio la stringa `voxl-logger/log0000/run/mpa/tof` corrisponde alla creazione di tutte le cartelle, in ordine e a partire dalla prima, nella cartella di lavoro (la stringa indica un percorso relativo). Si nota che viene controllata l'esistenza della cartella, in particolare viene creata la cartella e viene controllato l'eventuale errore (tramite la variabile `errno`). Se la cartella esiste già non è considerato un errore, infatti in questo caso si passa alla cartella successiva del percorso. Il secondo argomento della funzione serve a impostare i permessi delle cartelle create. Nel caso del programma, come si può vedere nel codice 3.1, viene passato il valore `0777`, corrispondente ai permessi di lettura e scrittura per ogni utente. I possibili valori di ritorno sono:

- 0: operazione avvenuta con successo
- -1: errore nella creazione della cartella (tranne l'errore relativo all'esistenza della cartella)

### 3.1.3 La funzione CopyFile

La funzione `CopyFile` è riportata nel codice 3.3.

Codice 3.3: Funzione CopyFile

```
1  int CopyFile (const char in[], const char out[])
2  {
3      FILE *f_in, *f_out;
4      char buffer[MAX_LINE_LEN];
5
6      if((f_in = fopen (in, "r" )) == NULL) {
7          return -1;
8      }
9
10     if((f_out = fopen (out, "w+" )) == NULL) {
11         return -2;
12     }
```

```
13
14      /* copies the file from the reference data line by line */
15      while (fgets(buffer, MAX_LINE_LEN-1, f_in) != NULL) {
16          fputs(buffer, f_out);
17      }
18
19      fclose(f_in);
20      fclose(f_out);
21
22      return 0;
23  }
```

La funzione serve a copiare, linea per linea il file specificato dal primo argomento tramite il suo nome (`in`) nel file, sempre specificato tramite il suo nome, passato tramite il secondo argomento (`out`). I possibili valori di ritorno sono:

- 0: operazione avvenuta con successo
- -1: errore nell'apertura del file d'ingresso
- -2: errore nell'apertura del file d'uscita

La funzione `CopyOtherFiles` si limita a costruire i percorsi dei vari file rimanenti e a chiamare la funzione `CopyFile` per ognuno.

### 3.1.4 Lo script di start modificato

Lo script `start.sh` è stato modificato in modo da renderlo compatibile con il `voxl-logger` simulato. Lo script è riportato nel codice 3.4.

Codice 3.4: Versione modificata di start.sh (voxl-logger simulato)

```
#!/bin/bash

cd "${0%/*}"

if [ "$#" -ne 1 ]; then
    echo "Provide Timestamp"
    exit 1
fi

./voxl-logger -d $1 &
```

La prima istruzione serve a spostarsi nella cartella corrente di lavoro (dove è il programma `voxl_logger` e lo script), infatti il programma Server si trova in una cartella diversa rispetto al programma di generazione dei dati ed è necessario trovarsi nella cartella del logger per la generazione dei dati (sono stati impostati dei percorsi relativi nel programma).

## 3.2 Le funzioni relative alla comunicazione di rete

Questa sezione sarà dedicata al commento delle funzioni generali scritte per la comunicazione (tramite Socket) fra Client e Server.

### 3.2.1 La libreria socketlib

Per la scrittura dei programmi di rete, e quindi delle funzioni relative alla comunicazione fra Client e Server, è stata utilizzata la libreria `socketlib`[1], sviluppata dal Prof. Pier Luca Montessoro, relatore di questa Tesi. In particolare, è stata usata la componente `tcpsocketlib`, infatti le Socket da utilizzare sono relative a un servizio connesso e affidabile (come già discusso nella sezione 2.1.2).

Si riporta, a titolo di esempio, il codice 3.5 relativo alla creazione della Socket da parte del Client (deve essere presente un Server che accetti la connessione).

Codice 3.5: Creazione della Socket da parte del Client

```
1  /* create a socket descriptor */
2  if ((sk = socket (AF_INET, SOCK_STREAM, 0)) < 0)
3  {
4      error_handler ("socket() [create_tcp_server()]");
5      return -1;
6  }
```

Si può notare (linea 1), il passaggio dell'argomento `SOCK_STREAM`, che indica la creazione di una Socket di tipo Stream, quindi basato sul protocollo TCP (connesso e affidabile). Il primo argomento passato alla funzione, `AF_INET`, fa riferimento al fatto che la Socket è usata per la comunicazione di rete tramite IPv4. Il terzo argomento rappresenta il protocollo da utilizzare e, nel caso in esame vale 0 in quanto c'è un singolo protocollo relativo alle Socket di tipo Stream.

### 3.2.2 Le funzioni tcp_getchar e tcp_getchar_bs

Nel codice 3.6 è riportata l'implementazione della funzione `tcp_getchar()` della libreria `tcpsocketlib`.

Codice 3.6: Funzione tcp_getchar()

```
1  int tcp_getchar (int sk)
2  {
3      static int i = 0;
4      static int dim = 0;
5      static char buffer[BUFSIZ + 1];
6
7      if (i >= dim || buffer[i] == '\0')
8      {
9          /* reload the buffer */
10         if ((dim = tcp_receive (sk, buffer)) == -1)
11             return EOF;
12         i = 0;
13     }
14
15     return buffer[i++];
16 }
```

La funzione è utile in quanto permette di ricevere sempre un carattere alla volta dal buffer di ricezione della Socket (indicato nell'unico argomento), senza dover pensare a

[1] La libreria è reperibile all'indirizzo https://github.com/uThings/socketlib

gestire quest'ultimo, infatti la richiesta dei dati nel caso siano finiti è gestita direttamente dalla funzione. Essa ritorna quindi il contenuto alla posizione specificata del buffer. In caso di errore, la funzione ritorna il carattere EOF.

Le variabili che gestiscono la dimensione e la posizione del buffer e il buffer stesso sono *statici*, quindi mantengono il proprio valore in memoria anche dopo diverse chiamate alla funzione. Il comportamento è corretto nel caso si abbia solamente un buffer, ma è incompatibile con l'implementazione dell'applicazione, che richiede di avere due buffer separati, uno di controllo e uno di trasmissione, che si mischierebbero usando la versione originale della funzione. La soluzione è di passare come argomento alla funzione sia il buffer sia le variabili associate alla sua gestione, per esempio mediante una `struct` (codice 3.7).

Codice 3.7: Struttura usata per gestione del buffer

```
1 typedef struct buffer_tcp {
2     int dim;
3     int i;
4     char buffer[BUFSIZ+1];
5 } buffer_tcp;
```

Nel codice 3.8 è riportata la funzione usata per l'inizializzazione della struttura.

Codice 3.8: Funzione usata per l'inizializzazione della struttura

```
1 buffer_tcp init_buffer_tcp ()
2 {
3     buffer_tcp b;
4     b.dim = 0;
5     b.i = 0;
6
7     return b;
8 }
```

La funzione modificata è riportata nel codice 3.9, dove si può notare il passaggio di un puntatore alla struttura riportata nel codice 3.7 e l'uso dei campi di quest'ultima al posto delle variabili statiche.

Codice 3.9: Versione modificata della funzione tcp_getchar()

```
1  int tcp_getchar_bs (int sk, buffer_tcp* b)
2  {
3      if (b->i >= b->dim || b->buffer[b->i] == '\0')
4      {
5          /* reload the buffer */
6          if ((b->dim = tcp_receive (sk, b->buffer)) == -1)
7              return EOF;
8          b->i = 0;
9      }
10
11     return b->buffer[b->i++];
12 }
```

Non ci sono differenze nel valore di ritorno rispetto alla versione originale della funzione.

### 3.2.3 La funzione tcp_readline

Tutti i messaggi scambiati fra Client e Server sono delle stringhe terminate dal valore di *newline*: `\n`. Ad esempio, il messaggio che il Client invia al Server al fine di richiamare la primitiva SEND_NEXT è `SEND_NEXT\n`. Risulta comodo sviluppare una funzione che, sfruttando la funzione `tcp_getchar_bs`, legga i caratteri fino all'arrivo del carattere `\n`, in modo da ottenere l'intero messaggio. La funzione, chiamata `tcp_readline`, è riportata nel codice 3.10.

Codice 3.10: Funzione tcp_readline()

```
1  int tcp_readline (int csk, buffer_tcp* b, char buf[])
2  {
3      char ch;
4      int i = 0;
5      while ((ch = tcp_getchar_bs(csk, b)) != '\n' && ch != '\0') {
6          if (ch != EOF) {
7              buf[i++] = ch;
8          } else {
9              return EOF;
10         }
11     }
12     buf[i] = '\0';
13
14     return i;
15 }
```

I primi due argomenti servono alla funzione `tcp_getchar_bs` e sono rispettivamente la Socket e la struttura descritta nel codice 3.7. Il terzo argomento è invece un vettore di caratteri, dove verrà salvata la stringa relativa al messaggio. Si nota la terminazione di quest'ultima al termine del ciclo di acquisizione dei caratteri. I possibili valori di ritorno sono:

- i: operazione avvenuta con successo, viene ritornato il numero di caratteri letti (escluso quindi il terminatore)
- EOF: errore (della funzione `tcp_getchar_bs`)

Anche in questo caso viene ritornato il carattere EOF in caso di errore, mentre viene ritornato il numero di caratteri letto (quindi escluso il terminatore) in caso di successo.

# Capitolo 4

# Implementazione del Server

## 4.1 Struttura generale del Server

### 4.1.1 La funzione main

Il codice della funzione `main` è riportato nel codice 4.1.

Codice 4.1: Main del Server

```
1  int main(int argc, char *argv[])
2  {
3      if (argc != 3) {
4          fprintf(stderr, "args: server_ip_address, tcp_port\n");
5          exit(EXIT_FAILURE);
6      }
7
8      /* avoid child processes becoming zombies */
9      signal(SIGCHLD, SIG_IGN);
10
11     if (create_tcp_server (argv[1], atoi (argv[2])) < 0) {
12         fprintf(stderr, "Error creating the TCP server\n");
13         exit(EXIT_FAILURE);
14     }
15
16     return EXIT_SUCCESS;
17 }
```

Il programma Server ha due argomenti obbligatori da fornire, l'indirizzo ip e la porta. L'indirizzo ip, essendo un Server, è `0.0.0.0`. Per quanto riguarda la porta, basta che sia un valore più alto delle *well-known ports* del TCP/IP, quindi un valore maggiore di 1023.

Di seguito si può notare che viene ignorato il segnale `SIGCHLD`, segnale che viene generato alla terminazione di un processo figlio (si ricorda infatti che l'implementazione del Server sfrutterà il parallelismo e in particolare la generazione di processi figli). Il segnale viene ignorato per evitare la creazione dei cosiddetti processi *zombie*, che sono processi che non utilizzano alcuna risorsa ma mantengono delle informazioni in memoria per analizzare successivamente ad esempio il motivo di terminazione del processo. Questo succede quando il valore di ritorno di un processo non viene letto

dal processo padre (che l'ha generato), come nel caso dell'applicazione in questione. Ignorando il segnale si evita questo comportamento.

La funzione principale del `main` è la `create_tcp_server`, funzione della libreria `tcpsocketlib` che implementa tutte le primitive lato Server utili a creare la connessione (come descritto in sezione 2.1.3). In particolare, all'interno della funzione, viene creato il *Server Loop*, ciclo che chiama la funzione `server_handler` ogni volta che viene stabilita una connessione, in maniera sequenziale. Il loop termina quando la funzione `server_handler` ritorna il valore 0.

### 4.1.2 La funzione server_handler

La funzione deve essere definita dall'utente e descrive il comportamento dell'applicazione, infatti sarà dentro questa funzione che dovranno essere gestite tutte le primitive offerte dal Server al Client.

Come detto in sezione 4.1, il server loop è sequenziale, quindi gestisce una chiamata della funzione `server-handler` alla volta. Ne consegue che l'implementazione del parallelismo vada gestita internamente a questa funzione. Lo schema generale per generare un processo figlio e quindi avere un Server parallelo è riportato nel codice 4.2.

Codice 4.2: Schema generale di un Server parallelo

```
1  int server_handler ( ... )
2  {
3      ...
4      pid_t pid = 0;
5
6      if ((pid = fork()) == -1) {
7          printf("Unable to create child process\n");
8          exit (EXIT_FAILURE);
9      }
10
11     if (pid != 0) {
12
13         /* Processo padre, il quale compito e' di far rimanere
               attivo il Server */
14
15         printf("Child process created (pid = %X)\n", pid);
16         return 1;   /* keep parent server alive */
17     }
18     else {
19
20         /* Processo figlio, dove vengono effettivamente gestite
               le primitive */
21
22         ...
23
24         return 0;
25     }
26 }
```

La funzione `fork` è la funzione che permette di creare il processo figlio e ritorna l'identificativo del processo (il *PID*). Controllando questo valore, che va preventivamente inizializzato a zero, è possibile capire se si è nel processo padre o nel processo

figlio. Un valore diverso da zero indica di essere nel processo padre, infatti la tale condizione si può solo verificare dopo aver chiamato la funzione `fork`, compito del padre.

Nel caso dell'applicazione, assumendo attive entrambe le connessioni (controllo e trasmissione), saranno attive tre istanze del processo Server. La prima è il processo padre che mantiene attivo il server e smista le richieste. Le altre due sono i processi figli che gestiscono le connessioni e quindi le primitive chiamate dal Client.

Nel codice 4.3 è riportata l'intera funzione `server_handler`.

Codice 4.3: Funzione server_handler

```
1  int server_handler (int csk, char *ip_addr, int port)
2  {
3      buffer_tcp b = init_buffer_tcp();
4      char cmd [BUFSIZ+1];
5
6      pid_t pid = 0;
7      int active_connection = 1;
8
9      printf("connected to %s: %d\n", ip_addr, port);
10
11     if ((pid = fork()) == -1) {
12         printf("Unable to create child process\n");
13         exit (EXIT_FAILURE);
14     }
15
16     if (pid != 0) {
17
18         /* parent */
19
20         printf("Child process created (pid = %X)\n", pid);
21         return 1;   /* keep parent server alive */
22     }
23     else {
24
25         /* child */
26
27         while (active_connection) {
28             /* wait for cmd from the client */
29             tcp_readline(csk, &b, cmd);
30
31             if (strcmp(cmd, "STATUS") != 0)
32                 printf("TCP receive: %s\n", cmd);
33
34             if (!strcmp(cmd, "STATUS")) {
35                 Status(csk, LOGGER_NAME);
36             }
37             else if (!strncmp(cmd, "START", 5)) {
38                 Start(csk, cmd, PATH_START, LOGGER_NAME);
39             }
40             else if (!strcmp(cmd, "STOP")) {
41                 Stop(csk, PATH_STOP, LOGGER_NAME);
42             }
43             else if (!strcmp(cmd, "SEND_NEXT")) {
44                 SendNext(csk, LOGGER_NAME, DATA_DIR);
```

```
45              }
46              else if (!strncmp(cmd, "REMOVE_FILE", 11)) {
47                  RemoveFile(csk, cmd, DATA_DIR);
48              }
49              else if (!strncmp(cmd, "REMOVE_DIR", 10)) {
50                  RemoveDir(csk, cmd, DATA_DIR);
51              }
52              else if (!strncmp(cmd, "SEND_FILE", 9)) {
53                  SendFile(csk, cmd);
54              }
55              else if (!strcmp(cmd, "CLOSE")) {
56                  active_connection = 0;
57              }
58              else {
59                  tcp_send(csk, "Not valid command\n");
60                  printf("Not valid command (%s)\n", cmd);
61              }
62          }
63
64          printf("Closing child server process...\n");
65          return 0;
66      }
67  }
```

Gli argomenti, forniti dalla funzione chiamante (`create_tcp_server`) sono l'identificativo della Scoket, l'indirizzo ip del server e la porta. Guardando il codice del processo figlio è possibile notare come ci sia un ciclo attivo per tutta la durata della connessione. Il ciclo attende un comando dal Client e chiama la primitiva (la funzione) appropriata (sezione 2.2).

Un aspetto importante è che il Server non ha modo di sapere se la connessione è quella di controllo o trasmissione (i processi figli sono identici). Ne consegue che sarà compito del Client chiamare le primitive e gestire il parallelismo in modo appropriato, al fine di garantire l'indipendenza delle connessioni.

## 4.2 La primitiva STATUS

La primitiva STATUS (codice 4.4) ha il compito di comunicare al Client lo stato del sistema di logging.

Codice 4.4: Funzione Status

```
1  int Status (int csk, const char logger_name[])
2  {
3      int n = ProcNumber(logger_name);
4      if (n == 0) {
5          tcp_send(csk, "STATUS: 0\n");
6      }
7      else if (n > 0) {
8          tcp_send(csk, "STATUS: 1\n");
9      }
10     else {
11         tcp_send(csk, "ERROR FINDING ACTIVE PROCESSES\n");
12         return -1;
```

```
13      }
14
15      return 0;
16  }
```

Per fare ciò è necessario controllare se i processi del logger sono in esecuzione nel sistema operativo. La funzione `ProcNumber`, che ritorna il numero di processi in esecuzione con un certo nome è stata definita a questo scopo (sezione 4.2.1).

I possibili valori di ritorno (della funzione `Status`) sono:

- 1: il sistema di logging è attivo
- 0: il sistema di logging non è attivo
- -1: errore (della funzione `ProcNumber`)

### 4.2.1 La funzione ProcNumber

L'implementazione della funzione `ProcNumber` è riportata nel codice 4.5.

Codice 4.5: Funzione ProcNumber

```
1  int ProcNumber (const char name[])
2  {
3      FILE *res;
4      char buf[BUFFER_DIM];
5      int cnt = 0;
6
7      snprintf(buf, BUFFER_DIM-9, "pgrep -l %s", name);
8      if ((res = popen(buf, "r")) == NULL) {
9          return -1;
10     }
11
12     while (fgets(buf, BUFFER_DIM-1, res) != 0) {
13         cnt++;
14     }
15
16     pclose(res);
17
18     return cnt;
19 }
```

I processi in esecuzione su un sistema operativo Unix possono essere ottenuti tramite il programma `pgrep` con l'opzione `-l`. Nel codice 4.6 è riportato un esempio di uso del programma, usato per trovare le istanze del `voxl-logger` in esecuzione ed il loro identificativo (pid).

Codice 4.6: Esempio di uso del programma pgrep

```
$ pgrep -l voxl-logger
23195 voxl-logger
23227 voxl-logger
```

Contando le linee ritornate dal programma è possibile sapere quanti processi sono in esecuzione con quel nome. Al fine di ottenere l'output del programma Unix dal programma c è necessario usare la funzione `popen`, che esegue un processo (mediante

`fork` ed `execl`) creando una `pipe` fra il processo padre e figlio, in modo che sia possibile una comunicazione fra i due. La funzione ritorna uno stream che può essere letto o scritto (come se fosse un normale file). Nel caso del'applicazione viene letto lo stream (che sarà simile a quanto visto nel codice 4.6) linea per linea incrementando un contatore.

I possibili valori di ritorno (della funzione `ProcNumber`) sono:

- cnt: il numero di processi con un determinato nome in esecuzione sul sistema
- -1: errore della funzione `popen`

## 4.3 La primitiva START

La primitiva di START è associata alla comunicazione del nome della cartella (timestamp) da passare allo script di start del sistema di logging. Il comando ricevuto dal Server è pertanto del tipo: `START 20230210-153024`.

La funzione `Start` è riportata nel codice 4.7.

Codice 4.7: Funzione Start

```
1  int Start (int csk, const char cmd[], const char start_name[],
       const char logger_name[])
2  {
3      int n;
4      char voxl_command[128];
5      char timestamp[TIMESTAMP_LEN];
6
7      sscanf(cmd, "%*s %s", timestamp);
8      snprintf(voxl_command, 128, "%s %s", start_name, timestamp);
9
10     if ((n = ProcNumber(logger_name)) == 0) {
11         system(voxl_command);
12         printf("Starting the logging system\n");
13         tcp_send(csk, "Logging system STARTED\n");
14     }
15     else if (n > 0) {
16         printf("Logging system is already active\n");
17         tcp_send(csk, "The logging system is already running\n");
18     }
19     else {
20         printf("Error finding active processes\n");
21         tcp_send(csk, "ERROR FINDING ACTIVE PROCESSES\n");
22         return -1;
23     }
24
25     return 0;
26 }
```

Gli argomenti della funzione sono l'identificativo della Socket, il nome (percorso) dello script di lancio del programma di logging e il suo nome. La funzione estrae il nome della cartella e compone la stringa da usare come script di start (`voxl-logger -d 20230210-153024...`) e, se il logger non è già attivo, procede a lanciare lo script mediante la funzione `system`.

I possibili valori di ritorno sono:

- 0: primitiva eseguita con successo
- -1: errore (della funzione `ProcNumber`)

## 4.4 La primitiva STOP

La primitiva di STOP (codice 4.8) è molto simile a quella di START.

Codice 4.8: Funzione Stop

```
1  int Stop (int csk, const char stop_name[], const char logger_name
       [])
2  {
3      int n;
4
5      if ((n = ProcNumber(logger_name)) > 0) {
6          system(stop_name);
7          printf("Stopping the logging system\n");
8          tcp_send(csk, "Logging system STOPPED\n");
9      }
10     else if (n == 0){
11         printf("Logging system is NOT active\n");
12         tcp_send(csk, "The logging system is NOT running\n");
13     }
14     else {
15         printf("Error finding active processes\n");
16         tcp_send(csk, "ERROR FINDING ACTIVE PROCESSES\n");
17         return -1;
18     }
19
20     return 0;
21 }
```

I possibili valori di ritorno sono analoghi alla funzione di Start (sezione 4.3).

## 4.5 La primitiva SEND_NEXT

La funzione più complessa da sviluppare di tutto il Server è stata la `SendNext`, relativa alla primitiva SEND_NEXT. Nell'algoritmo 2 si riportano i passi da effettuare.

**Algoritmo 2** Pseudocodice della funzione SendNext

```
Trova il primo file nella cartella meno recente
Salva lo stato del sistema di logging

if Logger attivo AND c'è solo una cartella then
    min_count = 2        ▷ Numero minimo di file che ci devono essere nella cartella
else
    min_count = 1
end if

if Non ci sono cartelle then
    Invia NO_TR
else if conteggio file ≥ min_count then
    Comprime il file                    ▷ C'è (almeno) un file pronto da inviare
    Invia SENDING + informazioni
    Invia il file compresso
else if Logger attivo AND c'è solo una cartella then
    Invia WAIT                          ▷ C'è solo un file che sta venendo aggiornato
else
    Invia EMPTY                         ▷ Non ci sono file ed il logger è inattivo
end if
```

Il codice completo della funzione `SendNext` è riportato nel codice 4.9.

Codice 4.9: Funzione SendNext

```
1  int SendNext (int csk, const char logger_name[], const char
       dir_name[])
2  {
3      struct tr_info tr;
4      char buf[BUFSIZ+1];
5      char file_name[MAX_PATH_LEN];
6      FILE *f;
7      int logger_state;
8      int min_count;
9      int ret;
10
11     Bytef* comp_buffer = NULL;
12     long int source_len;
13     long int comp_len;
14
15     /* find the next dir and file */
16     ret = FindNextTr(&tr, dir_name);
17     if (ret == -1) {
18         fprintf(stderr, "Error finding the next dir\n");
19         tcp_send(csk, "ERROR FINDING NEXT DIR\n");
20         return -1;
21     }
22     else if (ret == -2) {
23         fprintf(stderr, "Error finding the next file\n");
24         tcp_send(csk, "ERROR FINDING NEXT FILE\n");
```

```
25          return -1;
26      }
27
28      /* check the logger_state */
29      if ((logger_state = ProcNumber(logger_name)) < 0) {
30          fprintf(stderr, "Error finding active processes\n");
31          tcp_send(csk, "ERROR FINDING ACTIVE PROCESSES\n");
32          return -1;
33      }
34
35      if (logger_state && tr.d.count == 1)
36          min_count = 2;
37      else
38          min_count = 1;
39
40      if (tr.d.count == 0) {
41          printf("No transmission folders found, sending NO_TR\n");
42          tcp_send(csk, "NO_TR\n");
43      }
44      else if (tr.f.count >= min_count) {
45          snprintf(buf, BUFSIZ+1, "SENDING %s %d %d %d %d", tr.d.
                first, tr.d.logger_n, tr.f.first, tr.f.last, tr.f.
                count);
46          snprintf(file_name, MAX_PATH_LEN, "%s/%s/voxl-logger/log
                %04d/run/mpa/tof/data_%d.pcd", dir_name, tr.d.first,
                tr.d.logger_n, tr.f.first);
47          printf("Sending %s\n", file_name);
48
49          if ((ret = CompressAndSend(csk, file_name, buf)) < 0) {
50              fprintf(stderr, "Error compressing buffer\n");
51              return -1;
52          }
53
54          printf("Finished sending %s\n", file_name);
55          return 0;
56      }
57      else if (logger_state && tr.d.count == 1) {
58          /* there is only one file that is being currently written
                */
59          printf("Sending WAIT\n");
60          tcp_send(csk, "WAIT\n");
61      }
62      else {
63          /* there are no files (and the logger is inactive) */
64          printf("Sending EMPTY\n");
65          snprintf(buf, BUFSIZ+1, "EMPTY %s %d\n", tr.d.first, tr.d
                .logger_n);
66          tcp_send(csk, buf);
67      }
68
69      return 0;
70  }
```

Gli argomenti della funzione sono l'identificativo della Socket, il nome del programma di logging e la cartella radice, sotto la quale andranno salvati i file ricevuti.

Le informazioni relative al primo file nella cartella meno recente vengono trovate

grazie alla funzione `FindNextTr`, alla quale viene passata una struttura del tipo `tr_info`, che a sua volta contiene due strutture del tipo `next_file` e `next_dir`, riportate nei codici 4.10 e 4.11.

Codice 4.10: Struct next_file

```
1 struct next_file
2 {
3     int first;      // numero del primo file (nome, es data_0.pcd)
4     int last;       // numero dell'ultimo file
5     int count;      // numero di file presenti nella cartella
6 };
```

Codice 4.11: Struct next_dir

```
1 struct next_dir
2 {
3     char first[TIMESTAMP_LEN];    // nome della prima cartella
4     char last[TIMESTAMP_LEN];     // nome dell'ultima cartella
5     int count;                    // numero di cartelle presenti
6     int logger_n;                 // cartella dove sono i dati
7                                      Point Cloud (log0000/log0001)
8 };
```

La funzione `FindNextDir` (e le altre funzioni chiamate dalla stessa) servono a compilare i campi di queste strutture.

Un aspetto importante è riconoscere la situazione in cui un file sta venendo scritto dal sistema di logging e quindi non è ancora disponibile all'invio. Per fare ciò è stata definita la variabile `min_count`, che rappresenta il numero minimo di file che devono essere presenti nella cartella al fine di procedere all'invio del file meno recente. Questo numero è pari a 1 nel caso il logger non sia attivo oppure nel caso siano presenti più cartelle (questo fatto implica che la cartella meno recente non riceverà nuovi file in quanto l'acquisizione dati è terminata). Se invece è presente una sola cartella e il sistema di logging è attivo allora devono essere presenti almeno due file, in quanto il file più recente è in fase di aggiornamento.

La scelta dell'invio della risposta al Client ricalca quanto detto in sezione 2.2.3.

Nel caso la risposta da inviare sia EMPTY, il Server deve anche allegare le informazioni relative alla cartella (timestamp) e al numero della cartella contenente i dati pcd, in modo che il Client possa poi richiedere i file rimanenti da inviare e possa richiedere l'eliminazione della cartella. Il messaggio di risposta inviato è riportato in figura 4.1.

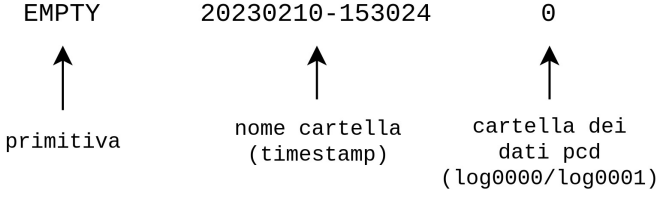


Figura 4.1: Esempio di risposta EMPTY

Si commenterà ora il caso d'invio del file. La prima cosa che viene fatta è la compressione del file, operazione che verrà discussa in dettaglio nel capitolo 6. Dopodiché

viene inviata la risposta SENDING al Client insieme alle informazioni utili a gestire la trasmissione. Un esempio di risposta, con i campi commentati è riportata in figura 4.2.

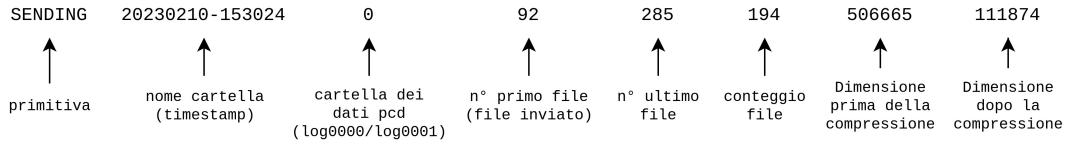


Figura 4.2: Esempio di risposta SENDING

La funzione utilizzata a tale scopo, `CompressAndSend`, è trattata in sezione 4.5.1.

### 4.5.1 La funzione CompressAndSend

L'implementazione della funzione `CompressAndSend` è riportata nel codice 4.12.

Codice 4.12: funzione CompressAndSend

```
1  int CompressAndSend (int csk, const char file_path[], const char
       buf[])
2  {
3      char send_buf[BUFSIZ+1];
4      Bytef *comp_buffer = NULL;
5      long int source_len = 0;
6      long int comp_len = 0;
7
8      /* compress the file */
9      if (compressor(file_path, &source_len, &comp_buffer, &
           comp_len) < 0) {
10         free(comp_buffer);
11         tcp_send(csk, "ERROR compressing buffer\n");
12         return -1;
13     }
14
15     snprintf(send_buf, BUFSIZ+1, "%s %ld %ld\n", buf, source_len,
           comp_len);
16     tcp_send(csk, send_buf);
17
18     /* send the file */
19     tcp_binary_send (csk, comp_buffer, comp_len);
20     free(comp_buffer);
21     tcp_send(csk, "END\n");
22     return 0;
23 }
```

Gli argomenti della funzione sono l'identificativo della Socket, il percorso del file da inviare e una stringa contenente le informazioni preliminari (che non dipendono dalla compressione) da allegare alla risposta SENDING (vedi figura 4.2). La scelta di spezzare il componimento delle informazioni d'invio è stata fatta per poter riutilizzare la funzione `CompressAndSend`, infatti essa sarà utilizzata anche dalla `SendFile`, che verrà trattata nella sezione 4.7 e che necessita d'informazioni di trasmissione diverse.

Come detto in precedenza, la funzione di compressione (`compressor`) verrà trattata nel dettaglio nel capitolo 6. Essenzialmente essa comprime il file d'ingresso in un buffer allocato dinamicamente e passato dall'esterno. Inoltre essa salva le informazioni relative alla lunghezza in byte del file non compresso e del file (buffer) compresso. Sono quest'ultime che devono essere aggiunte alle informazioni inviate dalla risposta SENDING (si vedano gli ultimi due campi della figura 4.2) in quanto sono necessario alla funzione `decompressor` del Client (duale della funzione `compressor`).

Il buffer contenente il file compresso non è più una stringa, pertanto deve essere inviato in modalità binaria (tramite la funzione `tcp_binary_send` della libreria `tcpsocketlib`). In coda al file viene inoltre inviata la stringa END per segnalare la fine della trasmissione del file binario, cosa che sarà utile lato Client per identificare una trasmissione non andata a buon fine. Al termine dell'invio il buffer dinamico viene liberato e la funzione termina.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore nella compressione del file (errore della funzione `compressor`)

## 4.6 La primitiva REMOVE_FILE

Il comando di rimozione di un file ha la forma riportata in figura 4.3.

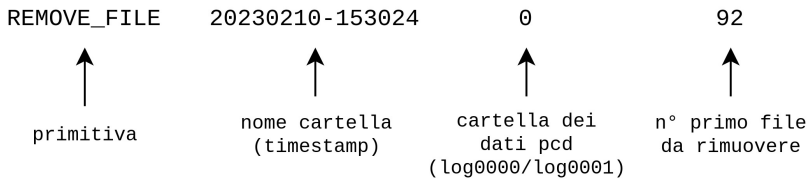


Figura 4.3: Esempio di comando REMOVE_FILE

Come detto in sezione 2.3, è infatti compito del Client indicare il file da rimuovere (esso chiamerà la funzione solo dopo aver finito di scrivere lo stesso file, appena ricevuto, in memoria).

Il codice che implementa la funzione `RemoveFile` è riportato nel codice 4.13.

Codice 4.13: funzione RemoveFile

```
1 int RemoveFile (int csk, const char cmd[], const char dir_name[])
2 {
3     int file_number;
4     int logger_n;
5     char file_name[MAX_PATH_LEN];
6     char timestamp[TIMESTAMP_LEN];
7     char buf[BUFSIZ+1];
8
9     sscanf(cmd, "%*s %s %d %d", timestamp, &logger_n, &
          file_number);
```

```
10      snprintf(file_name, MAX_PATH_LEN, "%s/%s/voxl-logger/log%04d/
            run/mpa/tof/data_%d.pcd", dir_name, timestamp, logger_n,
            file_number);
11      printf("Removing %s\n", file_name);
12
13      /* removes the file */
14      if (remove(file_name) != 0) {
15          tcp_send(csk, "ERROR DELETING FILE\n");
16          return -1;
17      }
18
19      sprintf(buf, "FILE %d, %s REMOVED\n", file_number, timestamp)
            ;
20      tcp_send(csk, buf);
21      return 0;
22  }
```

I parametri della funzione sono l'identificativo della Socket, il comando ricevuto, dal quale vengono estratte le informazioni relative al file da eliminare, e il nome della cartella radice dei dati, analogamente alla funzione `SendNext`. L'eliminazione del file avviene grazie alla funzione di libreria standard `remove`, che richiede il nome del file (percorso) completo, costruito opportunamente grazie alle informazioni estratte dal comando inviato dal Client.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore nell'eliminazione del file (errore della funzione `remove`)

## 4.7 La primitiva SEND_FILE

Il comando di invio di un file specifico richiede d'indicare il percorso del file ed ha una forma del tipo:

`SEND_FILE 20230210-153.../.../log0000/.../data.csv`

L'implementazione della funzione `SendFile` è riportata nel codice 4.14.

Codice 4.14: funzione SendFile

```
1   int SendFile (int csk, const char cmd[])
2   {
3       char buf[BUFSIZ+1];
4       char rel_path[MAX_PATH_LEN/2];
5       char file_path[MAX_PATH_LEN];
6       int ret;
7
8       sscanf(cmd, "%*s %s", rel_path);
9       snprintf(file_path, MAX_PATH_LEN, "%s/%s", DATA_DIR, rel_path
            );
10      snprintf(buf, BUFSIZ+1, "SENDING REQUESTED %s", file_path);
11
12      if ((ret = CompressAndSend(csk, file_path, buf)) < 0) {
13          fprintf(stderr, "ERROR compressing buffer\n");
14          return -1;
```

```
15      }
16
17      printf("Finished sending %s\n", file_path);
18      return 0;
19  }
```

La risposta che il Server invia al Client contiene il percorso del file e le informazioni relative alla compressione, come mostrato in figura 4.4.

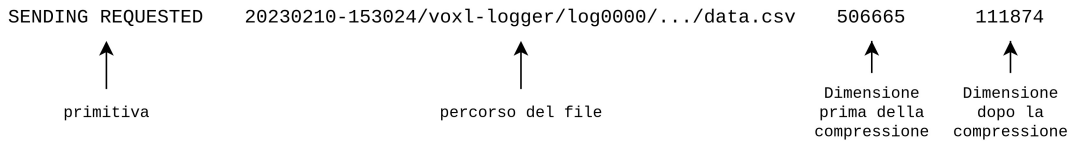


Figura 4.4: Esempio di risposta SENDING REQUESTED

La funzione utilizzata per la compressione e per l'invio del file è la funzione `CompressAndSend`, già discussa nella sezione 4.5.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore nella compressione/invio del file (errore della `CompressAndSend`)

## 4.8 La primitiva REMOVE_DIR

La primitiva serve a eliminare la cartella una volta finito il trasferimento dei dati. Il comando da inviare deve qundni contenere il nome della cartella da eliminare, ad esempio `REMOVE_DIR 20230210-153024`.

L'implementazione della funzione `RemoveDir` è riportata nel codice 4.15.

Codice 4.15: funzione RemoveDir

```
1  int RemoveDir (int csk, const char cmd[], const char dir_name[])
2  {
3      char path[MAX_PATH_LEN];
4      char timestamp[TIMESTAMP_LEN];
5      char buf[BUFSIZ+1];
6
7      sscanf(cmd, "%*s %s", timestamp);
8      snprintf(path, MAX_PATH_LEN, "%s/%s", dir_name, timestamp);
9
10     if (rmrf(path) < 0) {
11         snprintf(buf, BUFSIZ+1, "ERROR REMOVING DIR %s\n",
               timestamp);
12         tcp_send(csk, buf);
13         return -1;
14     }
15
16     snprintf(buf, BUFSIZ+1, "DIR %s REMOVED\n", timestamp);
17     tcp_send(csk, buf);
18
19     return 0;
20 }
```

Gli argomenti sono analoghi alla funzione alla funzione `RemoveFile` (sezione 4.6), infatti anche in questo caso la funzione deve estrapolare il nome della cartella dal comando inviato dal Client e deve conoscere la cartella radice dei dati.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore nella rimozione della cartella

## 4.9 La primitiva CLOSE

La primitiva CLOSE causa la fine del ciclo dei processi figli del Server (i gestori delle primitive) come si può vedere nel codice 4.3. Dopo essere uscito dal ciclo, il programma termina. Si ricorda che la terminazione è quella di un processo figlio, quindi il Server rimane comunque attivo.

## 4.10 Gestione degli errori

Le funzioni della libreria `tcpsocketlib` chiamano la funzione `error_handler` in caso di errori di connessione (delle funzioni `read` o `write` sulle Socket). La funzione è riportata nel codice 4.16.

Codice 4.16: funzione error_handler

```
1 void error_handler (const char *message)
2 {
3     printf("fatal error: %s\n", message);
4     exit(EXIT_FAILURE);
5 }
```

Il comportamento è quindi quello di terminare il processo (quindi il processo figlio) nel caso di un errore di connessione.

Gli errori nelle varie funzioni descritte in precedenza sono segnalati tramite il codice di ritorno ma non sono controllati dal Server. Ogni eventuale errore è infatti segnalato al Client (si vedano tutte le implementazioni precedenti), quindi si lascia a quest'ultimo la scelta di come gestire gli errori del Server.

# Capitolo 5

# Implementazione del Client

## 5.1 Struttura generale del Client

Il Client è stato sviluppato usando l'approccio del *Multithreading* per sfruttare il parallelismo. Questa scelta è stata fatta perché è necessario condividere dei dati fra il thread di controllo e quello di trasmissione. I dati da condividere sono relativi sia a dei flag per il controllo dello stato e sia ai valori relativi alla trasmissione (nome della cartella, file inviato, file rimanenti), che verranno mostrati all'utente tramite un'interfaccia grafica (che verrà trattata nella sezione 5.3). La funzione di creazione di un thread (`pthread_create`) permette infatti di passare un puntatore, ad esempio a una struttura, in modo da condividere facilmente dei dati. La struttura in questione si chiama `thread_data` ed è riportata nel codice 5.1.

Codice 5.1: Struttura thread_data

```
1 typedef struct thread_data {
2     atomic_int status;
3     atomic_int running;
4     char ip[IPLEN];
5     int port;
6     pthread_mutex_t lock;
7     struct tr_info tr;
8     FILE *log_file;
9 } thread_data;
```

Le prime due variabili sono del tipo `atomic_int`, in modo da garantire che scritture e letture su queste variabili siano atomiche. Il main thread controllerà la variabile `status` mentre il transmissione thread controllerà la variabile `running`. La variabile scritta da un thread deve poter essere letta in modo sicuro dall'altro, da questo fatto nasce la necessità di usare un tipo atomico. Nella struttura è anche presente un `mutex`, primitiva di sincronizzazione necessaria per la struttura `tr_info`, che sarà usata per trasmettere le informazioni relative alla trasmissione dal thread adibito a quest'ultima al main thread. Il `mutex` sarà anche necessario alla funzione di stampa su file di log (sezione 5.4).

Il compito del Client (e quindi della funzione `main`) è quello inviare le primitive appropriate al Server in base all'input dato dall'utente. Nel caso sia richiesto l'inizio della trasmissione dei file, deve anche creare il thread di trasmissione, che si occuperà di trasmettere i file in maniera indipendente dal thread principale. Nell'algoritmo 3

è riportato lo pseudocodice della funzione `main`. L'implementazione è riportata nel codice 5.2.

Codice 5.2: Main del Client

```
1  int main (int argc, char *argv[])
2  {
3      int control_sk;
4      char cmd[MAXCMDLEN];
5      char rec[BUFSIZ+1];
6      char timestamp[TIMESTAMP_LEN];
7      char ch;
8
9      int active_connection = 0;
10     int end_client = 0;
11     int logger_status = 0;
12     int transmission_status;
13     thread_data t;
14     pthread_t thread_id;
15     struct tr_info tr;
16     buffer_tcp control_buffer = init_buffer_tcp();
17
18     /* ignore SIGPIPE in order to enter the error_handler after
          write() tries to write on a broken socket */
19     signal(SIGPIPE, SIG_IGN);
20
21     if ((t.log_file = fopen(LOG_FILE, LOG_FILE_MODE)) == NULL) {
22         fprintf(stderr, "Impossible to open the log file %s\n",
              LOG_FILE);
23         exit(EXIT_FAILURE);
24     }
25
26     /* set the log file to line buffered mode */
27     char log_buf[1024];
28     setvbuf(t.log_file, log_buf, _IOLBF, sizeof(log_buf));
29
30     t.running = t.status = 0;
31     strcpy(t.ip, argv[1]);
32     t.port = atoi(argv[2]);
33
34     if (pthread_mutex_init(&(t.lock), NULL) != 0) {
35         fprintf(stderr, "Mutex init failed\n");
36         exit(EXIT_FAILURE);
37     }
38
39     if (argc != 3) {
40         fprintf(stderr, "args: server_ip_address tcp_port\n");
41         exit(EXIT_FAILURE);
42     }
43
44     /* init the I/O */
45     InitIO();
46     PrintLog(t.log_file, &(t.lock), "Starting the system\n");
47
48     do {
49
```

```
50          /* control connection */
51          control_sk = Connect(argv[1], atoi(argv[2]), &
                active_connection, &end_client, &t);
52
53          while (active_connection && !end_client) {
54              /* get status */
55              if (TcpSendReceiveMT(control_sk, &control_buffer, "
                    STATUS\n", rec, &t, thread_id) < 0) {
56                  active_connection = 0;
57                  break;
58              }
59              sscanf(rec, "%*s %d", &logger_status);
60
61              /* check if the transmission thread has to be resumed
                     */
62              if (t.status == -2) {
63                  if (CreateTransmissionThread(control_sk, &t, &
                        thread_id) < 0) {
64                      ExitMainThread(control_sk, &t, thread_id,
                            EXIT_FAILURE);
65                  }
66              }
67
68              if ((ch = GetUserInput()) != EOF) {
69
70                  if (ch == '1') {
71                      PrintLog(t.log_file, &(t.lock), "Sending
                            START command\n");
72                      TimestampString(timestamp, "%Y%m%d-%H%M%S");
73                      snprintf(cmd, MAXCMDLEN, "START %s\n",
                            timestamp);
74                      if (TcpSendReceiveMT(control_sk, &
                            control_buffer, cmd, rec, &t, thread_id)
                            < 0) {
75                          active_connection = 0;
76                          break;
77                      }
78                  }
79
80                  if (ch == '2') {
81                      PrintLog (t.log_file, &(t.lock), "Sending
                            STOP command\n");
82                      if (TcpSendReceiveMT(control_sk, &
                            control_buffer, "STOP\n", rec, &t,
                            thread_id) < 0) {
83                          active_connection = 0;
84                          break;
85                      }
86                  }
87
88                  if (ch == '3') {
89                      if (t.status == 0) {
90                          if (CreateTransmissionThread(control_sk,
                                &t, &thread_id) < 0) {
91                              ExitMainThread(control_sk, &t,
                                    thread_id, EXIT_FAILURE);
```

```
92                          }
93                      }
94                  }
95
96                  if (ch == '4') {
97                      if (t.status == 1) {
98                          t.running = 0;
99                          PrintLog(t.log_file, &(t.lock), "Stopping
                                the transmission thread\n");
100                     }
101                 }
102
103                 if (ch == '5') {
104                     end_client = 1;
105                 }
106             }
107
108             if (t.status == -1) {
109                 PrintLog(t.log_file, &(t.lock), "ERROR detected
                        on transmission thread, ending main thread\n"
                        );
110                 ExitMainThread(control_sk, &t, thread_id,
                        EXIT_FAILURE);
111             }
112
113             /* copy data safely */
114             pthread_mutex_lock(&(t.lock));
115             tr = t.tr;
116             transmission_status = t.status;
117             pthread_mutex_unlock(&(t.lock));
118
119             PrintInfo(logger_status, transmission_status, tr);
120
121             sleepms(MS_DELAY);
122         }
123
124     } while (!end_client);
125
126     ExitMainThread(control_sk, &t, thread_id, EXIT_SUCCESS);
127
128     return EXIT_SUCCESS;
129 }
```

**Algoritmo 3** Pseudocodice della funzione main del Client

```
Ignora il segnale SIGPIPE                          ▷ Utile per la gestione degli errori
Apre e inizializza il file di logging
Inizializza il sistema di I/O
repeat
    Stabilisce la connessione di controllo con il Server          ▷ Tentativi multipli
    while Connessione stabilita AND Client attivo do
        Richiede lo stato del sistema di logging al Server (primitiva STATUS)
        if Il transmission thread va riavviato then
            Avvia il transmission thread                ▷ Inizia la trasmissione dei file
        end if
        Ottiene l'input dall'utente
        if in == 1 then
            Ottiene il timestamp attuale
            Invia primitiva di START
        end if
        if in == 2 then
            Invia primitiva di STOP
        end if
        if in == 3 then
            if Il transmission thread non è attivo then
                Attiva il transmission thread
            end if
        end if
        if in == 4 then
            if Il transmission thread è attivo then
                Ferma il transmission thread
            end if
        end if
        if in == 5 then
            Ferma il Client                ▷ Imposta una variabile per fermare il ciclo
        end if
        Controlla un eventuale errore nel transmission thread
        Aspetta di 100 ms
    end while
until Client attivo
Termina il programma (Termine del sistema di I/O e del transmission thread se attivo)
```

Il sistema di I/O e il file di log verranno trattati nelle sezioni 5.3 e 5.4.

Il segnale `SIGPIPE` viene generato ogni volta che si prova a scrivere su una Socket disconnessa. Il comportamento di default è quello della terminazione del programma, quindi si è scelto d'ignorare il segnale, in modo da gestire l'errore internamente al programma. La gestione degli errori verrà trattata in dettaglio nella sezione 5.5. La disconnessione comunque non deve causare l'arresto, pertanto c'è un ciclo che inizia con il tentativo di connessione al Server e continua a tentare di connettersi dopo un ritardo (3 secondi). La funzione, chiamata `Connect`, è riportata nel codice 5.3.

Codice 5.3: Funzione Connect

```
1  int Connect (char ip[], int port, int *connected, int *end,
       thread_data *t)
2  {
3      int sk;
4
5      PrintLog(t->log_file, &(t->lock), "Trying to connect to the
           Server\n");
6
7      /* try to connect to the server */
8      do {
9          if ((sk = create_tcp_client_connection(ip, port)) < 0) {
10             PrintLog(t->log_file, &(t->lock), "Unable to connect
                   to the Server, retrying in a bit\n");
11             mvprintw(2, 0, "LOGGER: DISCONNECTED");
12             refresh();
13
14             /* if the user inputs 5, the program stops trying to
                   connect */
15             if (GetUserInput() == '5') {
16                 *end = 1;
17                 break;
18             }
19
20             sleep(3);
21         }
22         else {
23             PrintLog(t->log_file, &(t->lock), "Successfully
                   connected to the Server\n");
24             *connected = 1;
25         }
26     } while (!(*connected));
27
28     return sk;
29 }
```

La funzione permette di terminare il ciclo di tentativi di connessione se l'utente immette un determinato input e ritorna la Socket della connessione.

Una volta stabilita la connessione il programma entra in un ciclo (che dipende sia dalla variabile legata alla connessione e sia da un'altra variabile che indica se il Client deve essere attivo). La prima cosa fatta nel ciclo è la richiesta dello stato del sistema di logging al Server, mediante la primitiva STATUS. Dopodiché viene controllato lo stato del transmission thread, infatti se lo stato è pari a -2 significa che il thread va riavviato (c'è stato un problema di connessione in precedenza). Questo aspetto verrà

trattato in dettaglio nella sezione 5.5.2. Il compito principale del ciclo (e dell'intero main thread) è quello di ricevere un input dall'utente (sezione 5.3.3) e d'inviare la primitiva appropriata al Server. Un caso particolare è quando viene dato l'input relativo all'inizio della trasmissione. In questo caso il main thread ha il compito di attivare il transmission thread grazie alla funzione `CreateTransmissionThread`, riportata nel codice 5.4.

Codice 5.4: Funzione CreateTransmissionThread

```
1  int CreateTransmissionThread (int sk, thread_data *t, pthread_t *
       thread_id)
2  {
3      t->running = 1;
4      PrintLog(t->log_file, &(t->lock), "Starting the transmission
           thread\n");
5      if (pthread_create(thread_id, NULL, transmission_thread, (
           void *)t) != 0) {
6          PrintLog(t->log_file, &(t->lock), "ERROR - Failed to
               create transmission thread\n");
7          return -1;
8      }
9
10     PrintLog(t->log_file, &(t->lock), "Transmission thread
           started\n");
11     return 0;
12 }
```

La funzione ha come argomenti l'identificativo della Socket, un puntatore alla struttura `thread_data` e un puntatore all'identificativo del thread, il quale valore viene assegnato dalla funzione in caso di successo.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore nella creazione del thread (errore della funzione `pthread_create`)

Le modalità con cui il transmission thread viene controllato dal main thread (legate alle variabili `running` e `status` della struttura `thread_data`) verranno trattate nella sezione 5.2.

Prima del termine di ogni ciclo viene controllato lo stato del transmission thread (grazie alla variabile `status` della struttura `thread_data`). Grazie a questo controllo è possibile terminare il main thread nel caso sia occorso un errore nel transmission thread (sezione 5.5).

Nel caso d'input di un determinato valore, in questo caso il valore 5, viene impostata una variabile che ha il compito di far finire il ciclo principale del programma e quindi il programma stesso.

Terminato il ciclo il programma termina. Al fine di terminare correttamente, è stata scritta la funzione `ExitMainThread`, riportata nel codice 5.5.

Codice 5.5: Funzione ExitMainThread

```
1  void ExitMainThread (int sk, thread_data *t, pthread_t id, int
       exit_code)
2  {
```

```
3       PrintLog(t->log_file, &(t->lock), "CLOSING MAIN THREAD\n");
4
5       /* safely close the transmissione thread */
6       if (t->status == 1 || t->running == 1) {
7           PrintLog(t->log_file, &(t->lock), "Transmission thread is
                running, trying to join...\n");
8           t->running = 0;
9           if (pthread_join(id, NULL) != 0) {
10              PrintLog(t->log_file, &(t->lock), "Failed to join
                    transmission thread\n");
11              EndIO();
12              fclose(t->log_file);
13              exit(EXIT_FAILURE);
14          }
15          PrintLog(t->log_file, &(t->lock), "Transmission thread
                joined\n");
16      } else {
17          PrintLog(t->log_file, &(t->lock), "Transmission thread is
                not running, nothing to join...\n");
18      }
19
20      /* end curses mode */
21      EndIO();
22
23      /* destroy the mutex */
24      pthread_mutex_destroy(&(t->lock));
25
26      printf("Closing client...\n");
27      tcp_send(sk, "CLOSE\n");
28      PrintLog(t->log_file, &(t->lock), "CLIENT CLOSED\n");
29
30      fclose(t->log_file);
31      exit(exit_code);
32  }
```

Gli argomenti della funzione sono l'identificativo della Socket, un puntatore alla struttura `thread_data`, l'identificativo del transmission thread e il valore di ritorno del programma, che viene passato alla funzione standard `exit`. L'obiettivo della funzione è quello di chiudere in maniera sicura il thread di trasmissione (se attivo), grazie alla funzione `pthread_join`.

La funzione inoltre distrugge il mutex, chiude il file di log e chiama la funzione `EndIO`, che serve a chiudere il sistema di I/O (sezione 5.3.2).

## 5.2 Il thread di trasmissione

Il thread di trasmissione ha il compito di gestire completamente la trasmissione ed eliminazione dei file, come detto in sezione 2.3. L'algoritmo 4 riporta lo pseudocodice relativo a quest'operazione.

**Algoritmo 4** Pseudocodice del transmission thread

```
Stabilisce la connessione di trasmissione con il Server
repeat
    Invia la primitiva SEND_NEXT
    while Riceve WAIT AND il thread è attivo do
        Aspetta 100 ms
        Invia la primitiva SEND_NEXT
    end while
    if Riceve SENDING then
        Estrae le informazioni di trasmissione
        if Cartella (timestamp) non esiste then
            Crea la cartella di trasmissione        ▷ Aggiungendo -active al nome
        end if
        Riceve il file compresso
        Decomprime il file
        Scrive il file (decompresso) sul disco
        Invia la primitiva REMOVE_FILE              ▷ Relativa al file appena scritto
    end if
    if riceve EMPTY then
        Estrae le informazioni relative alla cartella vuota
        Richiede tutti i file rimanenti             ▷ Riceve, decomprime e scrive i file
        Richiede la cancellazione della cartella sul Server (primitiva REMOVE_DIR)
        Rinomina la cartella locale                 ▷ Rimuove -active dal nome della cartella
    end if
until Non riceve NO_TR AND il thread è attivo
Termina il thread di trasmissione
```

L'implementazione completa è riportata nel codice 5.6.

Codice 5.6: Funzione transmission_thread

```
1  void *transmission_thread (void* arg)
2  {
3      int transmission_sk;
4      buffer_tcp b = init_buffer_tcp();
5      char rec[BUFSIZ+1];
6      char file_name[MAX_PATH_LEN];
7      char transmission_dir[MAX_REL_PATH_LEN];
8      char cmd[MAXCMDLEN];
9      char folder[TIMESTAMP_LEN];
10     int logger_n;
11     int ret;
12     struct tr_info tr;
13     thread_data *t = (thread_data *)arg;
14     t->status = 1;
15
16     long int rec_len;
17     long int orig_len;
18
19     /* transmission connection */
```

```
20      if ((transmission_sk = create_tcp_client_connection(t->ip, t
            ->port)) < 0) {
21          PrintLog(t->log_file, &(t->lock), "ERROR - TT - cannot
                open TCP connection\n");
22          ExitTransmissionThread(transmission_sk, t, -1);
23      }
24
25      do {
26          PrintLog(t->log_file, &(t->lock), "TT - Requesting the
                next file\n");
27          TcpSendReceiveTT(transmission_sk, &b, "SEND_NEXT\n", rec,
                t);
28
29          while (strncmp(rec, "WAIT", 4) == 0 && t->running) {
30              PrintLog(t->log_file, &(t->lock), "TT - Wait received
                    , waiting...\n");
31              sleepms(WAIT_DELAY_MS);
32              PrintLog(t->log_file, &(t->lock), "TT - Requesting
                    the next file\n");
33              TcpSendReceiveTT(transmission_sk, &b, "SEND_NEXT\n",
                    rec, t);
34          }
35
36          if (strncmp(rec, "SENDING", 7) == 0) {
37
38              sscanf(rec, "%*s %s %d %d %d %d %ld %ld", tr.d.first,
                     &tr.d.logger_n, &tr.f.first, &tr.f.last, &tr.f.
                    count, &orig_len, &rec_len);
39
40              /* safely copy the transmission values */
41              pthread_mutex_lock(&(t->lock));
42              t->tr = tr;
43              pthread_mutex_unlock(&(t->lock));
44
45
46              snprintf(transmission_dir, MAX_REL_PATH_LEN, "%s/%s-
                    active", DATA_DIR, tr.d.first);
47              if (DirExists(transmission_dir) == 0) {
48                  CreateTransmissionDir(transmission_dir, tr.d.
                        logger_n);
49              }
50
51              snprintf(file_name, MAX_PATH_LEN, "%s/voxl-logger/log
                    %04d/run/mpa/tof/data_%d.pcd", transmission_dir,
                    tr.d.logger_n, tr.f.first);
52
53
54              PrintLog(t->log_file, &(t->lock), "TT - Receiving %s,
                     compressed size: %ld, original size: %ld\n",
                    file_name, rec_len, orig_len);
55
56              if ((ret = ReceiveAndDecompress(transmission_sk, t, &
                    b, file_name, rec_len, &orig_len)) < 0) {
57                  PrintLog(t->log_file, &(t->lock), " ERROR - TT -
                        ReceiveAndDecompress() ret: %d\n", ret);
58                  ExitTransmissionThread(transmission_sk, t, -1);
```

```
59              }
60
61              PrintLog(t->log_file, &(t->lock), "TT - Finished
                    writing %s\n", file_name);
62
63              if (ret == 0) {
64                  PrintLog(t->log_file, &(t->lock), "TT - Request
                        to remove file %d, dir %s\n", tr.f.first, tr.
                        d.first);
65
66                  sprintf(cmd, "REMOVE_FILE %s %d %d\n", tr.d.first
                        , tr.d.logger_n, tr.f.first);
67
68                  TcpSendReceiveTT(transmission_sk, &b, cmd, rec, t
                        );
69              }
70          }
71
72          /* if the transmission is finished, send the other files
                */
73          if (!strncmp(rec, "EMPTY", 5)) {
74              sscanf(rec, "%*s %s %d", folder, &logger_n);
75              PrintLog(t->log_file, &(t->lock), "TT - Finished
                    sending files (EMPTY)\n");
76
77              /* change to request */
78              if (RequestAllReaminingFiles(transmission_sk, &b, t,
                    transmission_dir) < 0) {
79                  PrintLog(t->log_file, &(t->lock), "ERROR - TT -
                        Failed to requenst remaining files\n");
80                  ExitTransmissionThread(transmission_sk, t, -1);
81              }
82
83              /* remove the dir from the server */
84              PrintLog(t->log_file, &(t->lock), "TT - Request to
                    remove %s from the server\n", folder);
85              snprintf(file_name, MAX_REL_PATH_LEN, "REMOVE_DIR %s\
                    n", folder);
86              TcpSendReceiveTT(transmission_sk, &b, file_name, rec,
                     t);
87
88              /* rename local dir */
89              PrintLog(t->log_file, &(t->lock), "TT - Renaming
                    local dir to %s\n", folder);
90              snprintf(file_name, MAX_REL_PATH_LEN, "%s/%s",
                    DATA_DIR, folder);
91              rename(transmission_dir, file_name);
92          }
93
94      } while ((strncmp(rec, "NO_TR", 4) != 0) && t->running);
95
96      PrintLog(t->log_file, &(t->lock), "TT - Transmission finished
            , ending transmission thread\n");
97
98      ExitTransmissionThread(transmission_sk, t, 0);
99  }
```

Dopo aver stabilito la connessione con il server (creando la Socket di trasmissione) il programma entra in un ciclo. Il ciclo termina quando viene ricevuto un messaggio NO_TR dal Server (non ci sono file da inviare) oppure quando la variabile `running`, contenuta nella struttura condivisa dal main thread e dal transmission thread `thread_data`, viene impostata a zero dal main thread. La variabile è solamente gestita dal main thread e controlla essenzialmente il transmission thread. Il transmission thread usa invece la variabile `status`, contenuta nella stessa struttura, tramite la quale comunica eventuali errori, che vengono trattati dal main thread, come visto in sezione 5.1.

Il ciclo invia continuamente la primitiva SEND_NEXT al Server e agisce i base alla sua risposta. In caso di riposta WAIT il programma entra in un ciclo che ripete l'invio della primitiva dopo un ritardo di 100 ms fino a che non arriva una risposta diversa. Questo ciclo può essere interrotto sempre dalla variabile `running`.

Nel caso di riposta SENDING, il programma estrae le informazioni relative all'invio del file (sezione 4.5). Dopodiché viene controllata l'esistenza della cartella sul Server e, nel caso non esista, viene creato l'intero albero di cartelle grazie alla funzione `CreateTransmissionDir`, che si limita a chiamare diverse volte la funzione `MakePath` (sezione 3.1.2). La cartella viene nominata aggiungendo `-active` dopo il timestamp (nome della cartella) in modo che sia possibile individuare le cartelle relative a trasmissioni ancora in corso (la cartella verrà rinominata rimuovendo `-active` al termine della trasmissione). Dovrebbe comunque esserci solo una cartella attiva alla volta per come è stato progettato il protocollo di comunicazione fra Client e Server. Un esempio di albero di cartelle presente sul Client dopo aver effettuato alcune trasmissioni (delle quali una in corso) è riportato in figura 5.1.

```
DATA_DIR
├── 20230120-153000
├── 20230120-153143
├── 20230120-153356
├── 20230120-153503
└── 20230120-153811-active
```

Figura 5.1: Esempio di diverse cartelle relative a diverse sessioni di logging salvate nel Client

Nel codice può anche essere notato l'uso delle primitive di sincronizzazione (mutex) quando vengono copiati i dati condivisi dai due thread.

### 5.2.1 La funzione ReceiveAndDecompress

La ricezione del file compresso e la sua decompressione avviene tramite la funzione `ReceiveAndDecompress`, duale della funzione `CompressAndSend` del Server (sezione 4.5.1). L'implementazione della funzione è riportata nel codice 5.7.

Codice 5.7: Funzione ReceiveAndDecompress

```
1 int ReceiveAndDecompress (int sk, thread_data *t, buffer_tcp* b,
      const char file_path[], long int rec_len, long int *orig_len)
2 {
3     int i;
```

```
4       int ret;
5       char ch;
6       char rec[BUFSIZ+1];
7       Bytef* rec_buffer = NULL;
8
9       /* receive the compressed buffer */
10      rec_buffer = malloc(rec_len);
11      for (i = 0; i < rec_len; i++) {
12          rec_buffer[i] = tcp_binary_getchar_bs(sk, b, t);
13      }
14
15      if ((ret = tcp_readline(sk, b, rec)) == EOF || ret == 0) {
16          /* connection error */
17          PrintLog(t->log_file, &(t->lock), "ERROR - TT -
                Connection error while receiving %s\n", file_path);
18          ExitTransmissionThread(sk, t, -2);
19      }
20
21      if (strncmp(rec, "END", 3) != 0)
22          return -4;
23
24      if ((ret = decompressor(file_path, rec_buffer, rec_len,
            orig_len)) < 0) {
25          free(rec_buffer);
26          return ret;
27      }
28
29      free(rec_buffer);
30      return 0;
31  }
```

La funzione ha come argomenti l'identificativo della Socket, la struttura utile alla lettura del buffer `buffer_tcp`, il percorso dove scrivere il file ricevuto (`file_path`), la lunghezza del file compresso `rec_len` e un puntatore alla lunghezza originale del file. Si ricorda che le ultime due informazioni vengono trasmesse tramite la risposta SENDING e sono direttamente collegate alla funzione di compressione. Tutte le funzioni di compressione e decompressione saranno descritte in dettaglio nel capitolo 6.

La ricezione del buffer compresso avviene tramite un ciclo `for` che chiama la funzione `tcp_binary_getchar_bs` per ogni byte del buffer da ricevere. Quest'ultima funzione (codice 5.8) è una versione modificata della funzione `tcp_getchar_bs`, descritta in sezione 3.2.2.

Codice 5.8: Funzione tcp_binary_getchar_bs

```
1   int tcp_binary_getchar_bs (int sk, buffer_tcp* b, thread_data *t)
2   {
3       if (b->i >= b->dim)
4       {
5           /* reload the buffer */
6           if ((b->dim = tcp_binary_receive (sk, b->buffer)) == -1)
                {
7               /* connection error */
8               PrintLog(t->log_file, &(t->lock), "ERROR - TT -
                    Connection error while using
```

```
                tcp_binary_getchar_bs\n");
9               ExitTransmissionThread(sk, t, -2);
10          }
11          b->i = 0;
12      }
13
14      return b->buffer[b->i++];
15  }
```

La differenza è che il carattere di terminazione delle stringhe (`\0`) non viene più considerato come fine del buffer e viene chiamata la versione binaria della funzione `tcp_receive`. La funzione chiama inoltre direttamente la funzioni di uscita dal thread in caso di errore di connessione (impostando correttamente il valore di uscita).

Una volta ricevuto il buffer compresso viene controllato se c'è stato un errore di trasmissione. Viene infatti letta una linea dal buffer e se questa non corrisponde alla stringa END, inviata dal Server al termine dell'invio del buffer compresso (sezione 4.5.1), significa che c'è stato un errore di trasmissione.

La funzione `decompressor` verrà trattata in dettaglio in sezione 6.3.1, ma essenzialmente decomprime il buffer e scrive il risultato in un file (il quale percorso viene passato come argomento).

Al termine della funzione viene liberato il buffer dinamico di ricezione (usato dalla funzione `decompressor`).

I possibili valori di ritorno (della funzione `ReceiveAndDecompress`) sono:

- 0: successo della funzione
- -1, -2, -3: errore della funzione `decompressor` (sezione 6.3.1)
- -4: errore di ricezione

### 5.2.2 Il termine della trasmissione

Al termine della ricezione del file (con successo), il transmission thread invia la primitiva REMOVE, relativa al file appena scritto, al Server.

Nel caso il Client riceva una risposta EMPTY (insieme alle informazioni sulla cartella che non ha più file), procede a richiedere i file rimanenti al Server. Per questo motivo è stata definita la funzione `RequestAllRemainingFiles`, che si limita a chiamare la funzione `RequestFile` (codice 5.9) per ogni file mancante.

Codice 5.9: Funzione RequestFile

```
1  int RequestFile (int sk, buffer_tcp* b, const char rel_path[],
       const char data_dir[], thread_data *t)
2  {
3      int ret;
4      char cmd[MAXCMDLEN];
5      char buf[BUFSIZ+1];
6      char file_path[MAX_PATH_LEN];
7      char temp[MAX_PATH_LEN/2];
8
9      long int rec_len;
10     long int orig_len;
```

```
11
12      sprintf(cmd, "SEND_FILE %s\n", rel_path);
13      TcpSendReceiveTT(sk, b, cmd, buf, t);
14
15      if (!strncmp(buf, "SENDING REQUESTED", 17)) {
16
17          sscanf(buf, "%*s %*s %*s %ld %ld", &orig_len, &rec_len);
18          sscanf(rel_path, "%*[^/]%s", temp);
19          sprintf(file_path, "%s/%s", data_dir, temp);
20
21          return ReceiveAndDecompress(sk, t, b, file_path, rec_len,
                &orig_len);
22      }
23      else {
24          return -5;
25      }
26
27      return 0;
28  }
```

I primi due argomenti della funzione sono l'identificativo della Socket e la struttura `buffer_tcp`. La stringa `rel_path` rappresenta il percorso relativo del file sul Server, quindi a partire dalla cartella relativa al timestamp. Gli altri argomenti sono `data_dir`, cartella radice dei dati sul Client e la struttura `thread_data`.

La funzione invia la primitiva SEND_FILE insieme al percorso del file passato come argomento e in seguito chiama, come prima, la funzione `ReceiveAndDecompress` (sezione 5.2.1).

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1, -2, -3: errore della funzione `decompressor` (sezione 6.3.1)
- -4: errore di ricezione (errore della funzione `ReceiveAndDecompress`)
- -5: errore ritornato dal Server in seguito della primitiva SEND_FILE

Ricevuti correttamente tutti i file rimanenti, il Client richiede al Server di eliminare la cartella grazie alla primitiva REMOVE_DIR. L'ultima operazione è quella di rinominare la cartella locale (rimuovendo `-active`), grazie alla funzione standard `rename`.

Al termine del ciclo (quindi finite le cartelle sul Server che corrisponde alla risposta NO_TR o tramite comando dal main thread, grazie alla variabile condivisa `running`) il trasmission thread termina, chiamando la sua funzione d'uscita `ExitTransmissionThread` (codice 5.10).

Codice 5.10: Funzione ExitTransmissionThread

```
1  void ExitTransmissionThread (int sk, thread_data *t, int
       exit_code)
2  {
3       PrintLog(t->log_file, &(t->lock), "TT - CLOSING TRANSMISSION
           THREAD\n");
4
```

```
5       tcp_send(sk, "CLOSE\n");
6       PrintLog(t->log_file, &(t->lock), "TT - TRANSMISSION THREAD
            CLOSED\n");
7
8       t->status = exit_code;
9       pthread_exit(NULL);
10  }
```

La funzione imposta il suo valore di ritorno (0 siccome è un'uscita senza errori) nella variabile condivisa `status` e termina il thread.

## 5.3 Il sistema di I/O

Il sistema di Input/Output deve permettere all'utilizzatore dell'applicazione d'impartire i comandi, ad esempio l'inizio del sistema di logging o della trasmissione, in maniera semplice e affidabile. Un altro aspetto che deve garantire è un feedback, quindi deve mostrare le informazioni essenziali per avere un quadro completo dell'andamento del processo.

### 5.3.1 La libreria ncurses

La libreria usata è stata la libreria `ncurses` (*new curses*), implementazione della precedente libreria `curses`[1]. Essa fornisce un'interfaccia per la gestione dello schermo di un'applicazione su un terminale a caratteri. La stessa libreria viene anche usata per ottenere i comandi dall'utente. L'interfaccia grafica dell'applicazione sviluppata è minimale, e concerne solamente una piccola parte delle funzioni e delle potenzialità della libreria.

### 5.3.2 Funzioni preliminari del sistema di I/O

La funzione d'inizializzazione del sistema di I/O è stata chiamata `InitIO` e la sua implementazione è riportata nel codice 5.11.

Codice 5.11: Funzione InitIO

```
1  void InitIO ()
2  {
3       /* start curses mode */
4       initscr();
5
6       noecho();
7       nodelay(stdscr, TRUE);
8
9       printw("LIDAR Control System\n\n");
10      print_command_info(5, 0, 0);
11      refresh();
12  }
```

[1]La libreria ncurses implementa le stesse funzioni della libreria originale curses. La documentazione completa della libreria può essere trovata all'indirizzo internet:
https://pubs.opengroup.org/onlinepubs/7908799/xcurses/curses.h.html

La prima funzione, `initsrc`, serve a iniziare la modalità curses. La funzione `noecho` serve a non mostrare i caratteri immessi sulla tastiera nel terminale.

La funzione `nodelay`, con il secondo argomento `TRUE`, serve a rendere la funzione `getch` non bloccante. Grazie a questo fatto è quindi possibile ottenere immediatamente il valore digitato sulla tastiera.

La funzione `printw` serve a stampare i caratteri sul terminale mentre la funzione `print_command_info` serve a stampare le stringhe informative dei comandi disponibili sullo schermo (codice 5.14).

La funzione `refresh` è indispensabile e serve a riportare i risultati delle funzioni precedenti sullo schermo.

La funzione che termina la modalità curses, chiamata `EndIO`, si limita a chiamare la funzione di libreria `endwin`.

### 5.3.3 La funzione GetUserInput

L'input dell'utente è dato dalla pressione di determinati tasti della tastiera, in particolare i valori rilevanti sono i numeri da 1 a 5, come visto in sezione 5.1. La funzione usata per ottenere l'input è riportata nel codice 5.12.

Codice 5.12: Funzione GetUserInput

```
1  char GetUserInput ()
2  {
3      char ch;
4
5      /* get command */
6      if ((ch = getch()) != ERR) {
7          return ch;
8      }
9
10     return EOF;
11 }
```

La funzione `getch` non è bloccante (sezione 5.3.2), quindi viene ritornato il carattere se la funzione non ritorna il valore ERR. I possibili valori di ritorno sono infatti:

- ch: il carattere letto
- EOF: non c'è un carattere da leggere (la funzione non è bloccante)

### 5.3.4 La funzione PrintInfo

La funzione che mostra all'utente le informazioni relative al sistema (compone l'interfaccia utente) si chiama `PrintInfo` ed è riportata nel codice 5.13.

Codice 5.13: Funzione PrintInfo

```
1  void PrintInfo (int logger_status, int thread_status, struct
       tr_info tr)
2  {
3      move(2, 0);
4      clrtoeol();
```

```
5       refresh();
6
7       if (logger_status) {
8           mvprintw(2, 0, "LOGGER: ON");
9       }
10      else {
11          mvprintw(2, 0, "LOGGER: OFF");
12      }
13
14      if (thread_status == 1) {
15          move(3, 0);
16          clrtoeol();
17          mvprintw(3, 0, "TRANSMISSION:    %s    %d/%d", tr.d.first
                , tr.f.first, tr.f.last);
18      }
19      else {
20          move(3, 0);
21          clrtoeol();
22          mvprintw(3, 0, "TRANSMISSION: OFF");
23      }
24
25      refresh();
26  }
```

La funzione, in congiunzione con la stampa operata dalla funzione `InitIO`, crea l'interfaccia mostrata nel codice 5.14.

Codice 5.14: Esempio di interfaccia utente

```
LIDAR Control System

LOGGER: ON
TRANSMISSION:    20230218-190426    30/75

(1) : Start the logging system
(2) : Stop the logging system
(3) : Start the transmission
(4) : Stop the transmission
(5) : Quit
```

La prima informazione riguarda lo stato del logger e i suoi valori possono essere:

- ON: sistema di logging attivo
- OFF: sistema di logging non attivo
- DISCONNECTED: sistema disconnesso dal Server

La seconda informazione riguarda lo stato della trasmissione ed i suoi valori possono essere:

- OFF: non è in atto una trasmissione
- Informazioni della cartella e numero del file che sta venendo inviato rispetto all'ultimo file presente nella cartella

## 5.4 Il file di log

L'uso del terminale per l'interfaccia utente implica che non sia possibile usare le funzioni standard per le informazioni di log del programma (ad esempio la funzione `printf`). Per questo motivo è stato usato un file di log. Nel codice 5.15 è riportato il codice usato per l'apertura del file (estratto della funzione `main`).

Codice 5.15: Apertura del file di log

```
1  if ((t.log_file = fopen(LOG_FILE, LOG_FILE_MODE)) == NULL) {
2      fprintf(stderr, "Impossible to open the log file %s\n",
           LOG_FILE);
3      exit(EXIT_FAILURE);
4  }
5
6  /* set the log file to line buffered mode */
7  char log_buf[1024];
8  setvbuf(t.log_file, log_buf, _IOLBF, sizeof(log_buf));
```

Il file viene impostato in modalità bufferizzata a linea, in modo che i messaggi vengano scritti subito sul file. La funzione necessaria a scrivere sul file di log si chiama `PrintLog` ed è riportata nel codice 5.16.

Codice 5.16: Funzione PrintLog

```
1  int PrintLog (FILE *f, pthread_mutex_t *m, const char format[],
       ...)
2  {
3      va_list args;
4      va_start(args, format);
5      int ret;
6
7      char timestamp[26];
8      char buffer[MAX_LOG_FORMAT];
9
10     TimestampString(timestamp, "%Y-%m-%d %H:%M:%S");
11     snprintf(buffer, MAX_LOG_FORMAT, "%s - %s", timestamp, format
           );
12
13     pthread_mutex_lock(m);
14     ret = vfprintf(f, buffer, args);
15     pthread_mutex_unlock(m);
16
17     va_end(args);
18     return ret;
19 }
```

La funzione riceve come argomenti un file (il file di log), un mutex, una stringa di formato e un numero variabile di argomenti (come la funzione `printf`).

Entrambi i thread condividono il file di log, quindi l'operazione di scrittura deve essere protetta da accessi concorrenti mediante il mutex.

La funzione ottiene il timestamp attuale (funzione `TimestampString`) e inizia il messaggio con quest'ultimo, in modo da fornire un log molto preciso. Un estratto del risultato di una sessione è riportata nel codice 5.17.

Codice 5.17: Estratto del file di log

```
2023-02-21 13:06:14 - Starting the system
2023-02-21 13:06:14 - Trying to connect to the Server
2023-02-21 13:06:14 - Unable to connect to the Server, retrying
   in a bit
2023-02-21 13:06:17 - Successfully connected to the Server
2023-02-21 13:06:19 - Sending START command
2023-02-21 13:06:19 - R - Logging system STARTED
2023-02-21 13:06:20 - Starting the transmission thread
2023-02-21 13:06:20 - Transmission thread started
2023-02-21 13:06:20 - TT - Requesting the next file
2023-02-21 13:06:20 - TT - R - SENDING 20230221-130619 0 0 17 18
   505958 111654
2023-02-21 13:06:20 - TT - Receiving rec_data/20230221-130619-
   active/voxl-logger/log0000/run/mpa/tof/data_0.pcd, compressed
    size: 111654, original size: 505958
2023-02-21 13:06:20 - TT - Finished writing rec_data
   /20230221-130619-active/voxl-logger/log0000/run/mpa/tof/
   data_0.pcd
2023-02-21 13:06:20 - TT - Request to remove file 0, dir
   20230221-130619
2023-02-21 13:06:20 - TT - R - FILE 0, 20230221-130619 REMOVED
2023-02-21 13:06:20 - TT - Requesting the next file
2023-02-21 13:06:20 - TT - R - SENDING 20230221-130619 0 1 18 18
   504726 111078
2023-02-21 13:06:20 - TT - Receiving rec_data/20230221-130619-
   active/voxl-logger/log0000/run/mpa/tof/data_1.pcd, compressed
    size: 111078, original size: 504726
```

Le regole usate sono che il transmission thread inizia i propri messaggi sempre con la stringa `TT`. Inoltre i messaggi che provengono dal Server iniziano con la stringa `R`. I messaggi di errore iniziano con la stringa `ERROR`. Nell'estratto si può notare il protocollo in azione.

## 5.5 Gestione degli errori

In generale, quando accade un errore nel Client, oppure viene ricevuto un messaggio di errore dal Server, il programma Client deve terminare (causando anche la terminazione delle istanze del Server). Gli unici errori che andranno gestiti dal Client sono quelli di connessione, infatti nel caso di perdita di connessione (o di Server non attivo) verrà visualizzato un messaggio sull'interfaccia utente (`LOGGER: DISCONNECTED`). Il Client provvederà quindi a tentare nuovamente la connessione (a intervalli di tre secondi) fino a che non riesce nel suo scopo o il ciclo viene interrotta dall'utente.

### 5.5.1 Errori nel main thread

Il protocollo usato fra Client e Server prevede che a ogni messaggio del Client (invio di una primitiva) corrisponde una risposta del Server. Per questo motivo è stata scritta, per il main thread, la funzione `TcpSendReceiveMT` (codice 5.18).

Codice 5.18: Funzione SendReceiveMT

```
1  int TcpSendReceiveMT (int sk, buffer_tcp *b, char send[], char
       rec[], thread_data *t, pthread_t id) {
2
3      int ret;
4
5      if (tcp_send(sk, send) == 0) {
6          snprintf(rec, BUFSIZ+1, "ERROR, UNABLE TO WRITE\n");
7          PrintLog(t->log_file, &(t->lock), rec);
8          return -1;
9      }
10
11     if ((ret = tcp_readline(sk, b, rec)) == EOF || ret == 0) {
12         snprintf(rec, BUFSIZ+1, "ERROR, UNABLE TO READ\n");
13         PrintLog(t->log_file, &(t->lock), rec);
14         return -2;
15     }
16
17     /* check if the Server has send an error message. If so, exit
           */
18     if (strncmp(rec, "ERROR", 5) == 0) {
19         PrintLog(t->log_file, &(t->lock), "R - ERROR received
               from Server: %s\n", rec);
20         ExitMainThread(sk, t, id, EXIT_FAILURE);
21     }
22
23     /* print the result, only if it is not a STATUS response */
24     if (strncmp(rec, "STATUS", 6) != 0) {
25         PrintLog(t->log_file, &(t->lock), "R - %s\n", rec);
26     }
27
28     return 0;
29 }
```

Gli argomenti della funzione sono l'identificativo della Scoket e il suo buffer, una stringa che corrisponde al messaggio da inviare, un buffer dove salvare la risposta del Server, un puntatore alla struttura `thread_data` e l'identificativo del transmission thread.

La funzione invia la primitiva e controlla il risultato. La funzione chiamata dalle funzioni della `tcpsocketlib` in caso di errore, `error_handler`, non fa nulla, pertanto si utilizzano i codici di ritorno delle funzioni di trasmissione in modo da poter gestire gli errori di connessione. La funzione `tcp_send` ritorna il valore 0 se la Socket non è attiva (connessione non attiva). Si ricorda che il segnale SIGPIPE viene ignorato (sezione 5.1), infatti se non fosse ignorato la scrittura su una Socket chiusa causerebbe la terminazione del programma (come detto in sezione 5.1). Dopo la scrittura viene letta la risposta. La funzione `tcp_readline` ritorna il valore 0 se sono stati letti 0 byte di dati (connessione non attiva) e il valore EOF in caso di errore. L'impossibilità di scrivere o leggere la Socket implica che la connessione non è più presente, e il programma chiamante dovrà gestire la situazione nella maniera appropriata, in modo da ritentare la connessione.

La funzione controlla inoltre se il messaggio di risposta del Server inizia con la stringa ERROR. In questo caso si è in presenza di un errore nel Server, quindi il main thread deve terminare chiamando la funzione `ExitMainThread` (sezione 5.5),

causando anche l'eventuale chiusura del thread di trasmissione. I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore di scrittura sulla Socket
- -2: errore di lettura della Socket

Lo schema generale per il recupero della connessione è mostrato nel codice 5.19.

Codice 5.19: Schema generale per il recupero della connessione

```
1  do {
2      /* connessione al Server (tentativi multipli) */
3      active = Connect ()
4
5      while (active && !fine_client) {
6
7          ...
8
9          if (TcpSendReceiveMT(...) < 0) {
10             break;
11         }
12
13         ...
14
15     }
16 } while (!fine_client)
```

Il ciclo interno viene interrotto ogni volta che c'è un errore di connessione, in modo da tornare a ritentare la connessione. Lo schema è implementato nella funzione `main` del Client (codice 5.2).

### 5.5.2 Errori nel transmission thread

Anche nel caso del transmission thread è stata scritta una funzione per la gestione dell'invio dei messaggi e delle risposte del Server. La funzione si chiama `SendReceiveTT` ed è molto simile alla funzione scritta per il main thread `SendReceiveMT` (codice 5.18). La prima differenza è che le chiamate d'uscita della funzione utilizzano la funzione `ExitTransmissionThread` (codice 5.10) invece che la funzione `ExitMainThread`. La seconda differenza è che nel caso di errore di scrittura o lettura della Socket viene direttamente chiamata la funzione d'uscita, in quanto non c'è un meccanismo di recupero della connessione.

Prima di uscire, il transmission thread imposta lo stato della variabile `status` condivisa con il main thread (contenuta nella struttura `thread_data`). I possibili stati della variabile sono:

- 0: transmission thread inattivo
- 1: transmission thread attivo
- -1: errore generale nel transmission thread

- -2: errore di connessione nel transmission thread

Il main thread può quindi controllare in qualsiasi momento lo stato del thread di trasmissione e agire di conseguenza. In particolare (con riferimento al codice 5.2), le seguenti azioni sono effettuate dal main thread in base allo stato del transmission thread:

**-1:** viene chiamata la funzione `ExitMainThread`, si è in una condizione di errore non recuperabile

**-2:** la condizione viene controllata subito dopo aver stabilito la connessione con il Server, infatti la condizione -2 indica un errore di connessione nel transmission thread, quindi lo stesso va avviato (riattivato). Le condizioni di errore di connessione dei due thread sono quindi indipendenti

Riassumendo, nel caso la connessione venga persa durante la trasmissione, la condizione di errore di perdita della stessa (errore di scrittura o lettura della Socket) avviene indipendentemente nei due thread (e di conseguenza nelle due Socket). Il thread di trasmissione esce immediatamente impostando il suo stato a -2. Il main thread ritenta la connessione e, appena stabilita, controlla lo stato del transmission thread per poi ripristinarlo se necessario.

# Capitolo 6

# Utilizzo della compressione

## 6.1 La libreria zlib

La compressione dei dati è stata ottenuta grazie alla libreria `zlib`[1]. La libreria è la stessa usata nel celebre programma `gzip`.

### 6.1.1 Cenni sull'algoritmo di compressione

L'algoritmo di compressione `lossless` utilizzato dalla libreria si chiama DEFLATE, ed è una combinazione dell'algoritmo LZ77 e della codifica di Huffman. Lo schema di compressione è mostrato in figura 6.1.

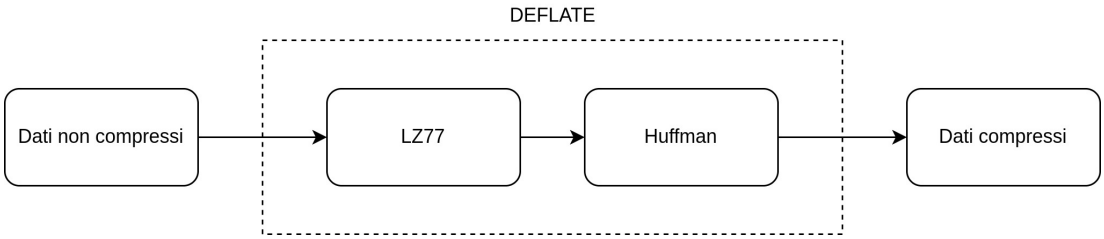


Figura 6.1: Schema dell'algoritmo DEFLATE

L'algoritmo LZ77 è una codifica dell'informazione senza perdita a dizionario. L'idea di base è quella di sostituire una sequenza di dati con un puntatore alla stessa sequenza già incontrata in precedenza. L'algoritmo LZ77 usa una *sliding window* per riconoscere sequenze di dati ripetute, quindi deve tenere traccia di una certa quantità di dati passati. Più è grande la *sliding window* più dati devono essere controllati dall'encoder (impattando le prestazioni ma possibilmente abbassando la dimensione dei dati compressi). Quando una corrispondenza viene trovata, essa viene sostituita con una *coppia lunghezza-distanza*. Essa indica che un certo numero di caratteri (lunghezza) sono ripetuti a una certa distanza indietro nel buffer non compresso.

La codifica di Huffman è un metodo di compressione statistico che permette di codificare dei simboli (sequenze di caratteri) in codici di lunghezza variabile. La lunghezza dei codici è dipendente dalla frequenza dei simboli, quindi simboli che

[1]La documentazione completa della libreria si può trovare all'indirizzo: https://www.zlib.net/

si ripetono spesso avranno una lunghezza minore rispetto ai simboli che sono meno frequenti. Nella pratica viene costruito un albero binario in cui ogni diramazione rappresenta l'aggiunta di un 1 o 0 al codice.

### 6.1.2 Le funzioni compress e uncompress

Le funzioni di libreria che sono state utilizzate nell'applicazione sono le funzioni `compress` e `uncompress`. Esse forniscono un metodo semplificato per comprimere dei buffer e internamente utilizzano le più complesse funzioni `deflate` e `inflate`. Il prototipo della funzione `compress` è riportato nel codice 6.1.

Codice 6.1: Prototipo della funzione compress

```
ZEXTERN int ZEXPORT compress OF(Bytef *dest, uLongf *destLen,
    const Bytef *source, uLong sourceLen)
```

Gli argomenti della funzione sono il buffer di destinazione (d tipo `Bytef`), un puntatore dove verrà salvata la dimensione del buffer compresso, il buffer d'ingresso e la sua lunghezza.

Un'altra funzione di libreria utile è la funzione `compressBound`, che serve a calcolare la dimensione massima del buffer d'uscita (compresso) data la dimensione del buffer d'ingresso. Grazie a questa funzione è possibile allocare dinamicamente i buffer prima della compressione. Le funzioni che useranno le funzioni di libreria saranno descritte nelle sezioni 6.2 e 6.3.

## 6.2 Le funzioni di compressione

### 6.2.1 La funzione compressor

La funzione che serve alla compressione di un file, e quindi a interfacciarsi con la libreria `zlib` si chiama `compressor` e la sua implementazione è riportata nel codice 6.2.

Codice 6.2: Funzione compressor

```
1  int compressor (const char filename[], long int *in_len, Bytef **
       out, long int *out_len)
2  {
3      Bytef *in = NULL;
4      int ret;
5
6      if ((*in_len = copy_file_in_buffer(filename, &in)) < 0) {
7          free(in);
8          return -1;
9      }
10
11     *out_len = compressBound(*in_len);
12     *out = malloc(*out_len);
13     if (*out == NULL) {
14         free(in);
15         return -2;
16     }
```

```
17
18      ret = compress(*out, out_len, in, *in_len);
19      if (ret != Z_OK) {
20          ret = -3;
21      }
22      else {
23          ret = 0;
24      }
25
26      free(in);
27      return ret;
28  }
```

Gli argomenti della funzione sono il nome del file da comprimere, un puntatore a `long int` dove verrà salvata la dimensione del file non compresso in byte, un puntatore a memoria dinamica che va gestito all'esterno della funzione (come mostrato nel codice 4.12) dove verrà salvato il buffer compresso e infine un puntatore a intero dove verrà salvata la dimensione del buffer compresso. Lo scopo della funzione è quello di, a partire da un file, fornire un buffer contenente il file compresso.

Come visto in sezione 6.1.2, le funzioni della libreria `zlib` lavorano con buffer, quindi il primo passo è quello di trasferire il contenuto del file d'ingresso in un buffer (allocato dinamicamente). La funzione, chiamata `copy_file_in_buffer` verrà trattata nella sezione 6.2.2.

Dopo aver copiato il contenuto del file in un buffer è sufficiente chiamare la funzione di libreria `compress`.

I possibili valori di ritorno (della funzione `compressor`) sono:

- 0: successo della funzione
- -1: errore nella copia del file nel buffer (errore di `copy_file_in_buffer`)
- -2: errore nell'allocazione della memoria dinamica
- -3: errore di compressione (della funzione `compress`)

### 6.2.2 La funzione copy_file_in_buffer

La funzione `copy_file_in_buffer` è riportata nel codice 6.3.

Codice 6.3: Funzione copy_file_in_buffer

```
1  long int copy_file_in_buffer (const char filename[], Bytef **buf)
2  {
3      FILE *f;
4      long int len;
5      if ((f= fopen(filename, "rb")) == NULL) {
6          return -1;
7      }
8
9      fseek(f, 0, SEEK_END);
10     len = ftell(f);
11     rewind(f);
12
13     *buf = malloc(len);
```

```
14      if(*buf == NULL) {
15          return -2;
16      }
17
18      if (len != 0 && fread(*buf, len, 1, f) != 1) {
19          return -3;
20      }
21
22      fclose(f);
23      return len;
24  }
```

Gli argomenti sono il nome del file e un puntatore a memoria dinamica (gestito dall'esterno) dove verrà copiato il contenuto del file.

Il file viene aperto in modalità binaria e per trovare la sua dimensione vengono usate le funzioni standard `fseek`, `ftell` e `rewind`. Questo metodo funziona solo con file che permettono la ricerca, e questo fatto non è vero in generale per i file binari. Nel caso dell'applicazione si sa che tutti i file sono file testuali, quindi il metodo appena descritto può essere usato.

Il file viene poi copiato in memoria grazie alla funzione standard `fread`.

I possibili valori di ritorno sono:

- len: lunghezza (in byte) del buffer
- -1: errore di apertura del file d'ingresso
- -2: errore di allocazione della memoria dinamica
- -3: errore nella copia del file (errore della funzione `fread`)

## 6.3 Le funzioni di decompressione

### 6.3.1 La funzione decompressor

La funzione `decompressor`, duale della funzione `compressor` è riportata nel codice 6.4.

Codice 6.4: Funzione decompressor

```
1  int decompressor (const char filename[], Bytef in[], long int
       in_len, long int *out_len)
2  {
3       Bytef *out = NULL;
4       int ret;
5
6       out = malloc(*out_len);
7       if (out == NULL) {
8           free(out);
9           return -1;
10      }
11
12      ret = uncompress(out, out_len, in, in_len);
13      if (ret != Z_OK) {
```

```
14          free(out);
15          return -2;
16      }
17
18      if (copy_buffer_in_file(filename, out, *out_len) < 0) {
19          free(out);
20          return -3;
21      }
22
23      free(out);
24      return 0;
25  }
```

Gli argomenti della funzione sono il nome del file di uscita, il buffer compresso da decomprimere, la lunghezza del file compresso e un puntatore a intero dove verrà salvata la lunghezza del file decompresso (già nota grazie alla trasmissione).

Dopo aver allocato la memoria dinamica necessaria alle operazioni, viene chiamata la funzione di libreria `uncompress` in modo da ottenere un buffer decompresso.

Dopodiché viene utilizzata la funzione `copy_buffer_in_file` per copiare il contenuto del buffer nel file specificato negli argomenti (la funzione verrà trattata in sezione 6.3.2).

I possibili valori di ritorno (della funzione `decompressor`) sono:

- 0: successo della funzione
- -1: errore nell'allocazione della memoria dinamica
- -2: errore di decompressione (della funzione `uncompress`)
- -3: errore nella copia del buffer nel file (errore di `copy_buffer_in_file`)

### 6.3.2 La funzione copy_buffer_in_file

La funzione `copy_buffer_in_file` è riportata nel codice 6.5.

Codice 6.5: Funzione copy_buffer_in_file

```
1  int copy_buffer_in_file (const char filename[], Bytef buf[], int
       len)
2  {
3      FILE *f;
4      if ((f = fopen(filename, "wb")) == NULL) {
5          return -1;
6      }
7
8      if (len != 0 && fwrite(buf, len, 1, f) != 1) {
9          return -2;
10     }
11
12     fclose(f);
13     return 0;
14 }
```

Gli argomenti della funzione sono il nome del file d'uscita, il buffer d'ingresso e la lunghezza del buffer.

La funzione apre il file di uscita in modalità binaria e in seguito copia il buffer nel file mediante la funzione standard `fwrite`.

I possibili valori di ritorno sono:

- 0: successo della funzione
- -1: errore di apertura del file d'uscita
- -2: errore nella scrittura del buffer nel file (errore della funzione `fwrite`)

## 6.4 Misura dell'impatto della compressione

L'effetto della compressione sulla quantità di dati trasmessa è stata misurata grazie al programma `iftop`[2], che permette di misurare l'uso della banda su un'interfaccia di rete. Le opzioni di lancio del programma sono le seguenti:

```
sudo iftop -i lo -f "src port 10000"
```

In questo modo è possibile misurare il traffico sulla porta 10000 dell'interfaccia di rete `loopback` (utile a testare i programmi di rete).

Il campione di dati usato per il test è composto in tutto da 4100 file, con un peso totale di 2 GB.

Il risultato della trasmissione completa è mostrato in figura 6.2.

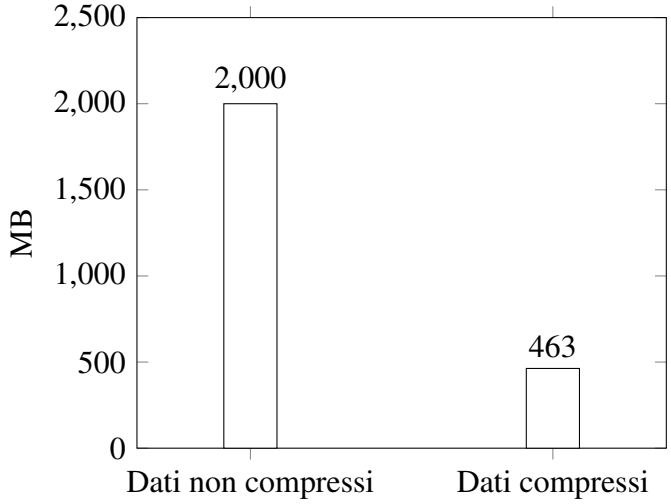


Figura 6.2: Confronto fra trasmissione di dati non compressi e compressi

Il risultato (trasmissione di 463 MB effettivi a fronte di 2000 MB) dimostra l'efficacia della compressione, infatti il tasso di riduzione è di circa il 77%, in accordo con le prove effettuate tramite il programma `zip` (sezione 2.6).

[2]La documentazione completa del programma può essere trovata all'indirizzo: https://linux.die.net/man/8/iftop

# Capitolo 7

# Conclusioni

Il risultato dello sviluppo dell'applicazione può essere considerato un successo, infatti tutti gli obiettivi posti nel capitolo 1 sono stati raggiunti senza particolari compromessi e l'uso della compressione ha notevolmente ridotto la quantità di dati trasmessi. Tuttavia, lo sviluppo dell'applicazione e il test della stessa è stato interamente effettuato utilizzando il programma di simulazione del `voxl-logger`, descritto in sezione 3.1. Per questo motivo l'applicazione è da considerarsi come un prototipo sperimentale. In futuro sarà quindi necessario portare il sistema (Server) sulla piattaforma reale (il drone) in modo da effettuare dei test accurati sul campo. Il programma dovrebbe comunque essere interamente compatibile, infatti il drone monta un sistema operativo Unix, come il computer usato per lo sviluppo e test.

Un altro aspetto dell'applicazione che può essere migliorato è la gestione degli errori, infatti ora gli unici errori che vengono gestiti sono quelli di connessione. Gli altri errori generici causano la terminazione del programma. In futuro si potrebbero implementare delle funzioni per la gestione di alcuni errori generici, al fine di recuperare la trasmissione.

Per quanto riguarda l'interfaccia utente, una possibile miglioria è quella di sviluppare una GUI (*graphical use interface*) utilizzando una libreria apposita (come `Qt` o `gtk`). L'interfaccia attuale è infatti molto minimale e mostra solo le informazioni essenziali.

Un'altra possibilità è quella di sfruttare il parallelismo lato Client (Multithreading) per separare il sistema di input/output dal resto dell'applicazione (usando uno o due thread appositi).

Uno degli sviluppi più interessanti è comunque quello di portare il sistema Client su un sistema embedded alimentato a batteria, in modo che non sia necessario avere un computer portatile per interfacciarsi con il drone. Una possibile famiglia di *single-board computer* da usare a tale scopo potrebbero essere le *Orange Pi* con una distribuzione Linux come sistema operativo. I controlli potrebbero essere dei semplici pulsanti e il sistema di output potrebbe diventare un piccolo schermo a basso consumo. Sarebbe necessario anche progettare e stampare, grazie a una stampante 3D, una scatola tale da contenere tutti i componenti descritti in precedenza.

# Bibliografia


[1] *Sito del produttore del drone*. https://cleorobotics.com/product.

[2] *Manuale voxl-logger*. https://docs.modalai.com/voxl-logger.

[3] *Libreria socketlib*. https://github.com/uThings/socketlib.

[4] *Guida all'uso del Multithreading*. https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html.

[5] *Manuale curses*. https://pubs.opengroup.org/onlinepubs/7908799/xcurses/curses.h.html.

[6] *Manuale zlib*. https://www.zlib.net.

[7] *Algoritmo di compressione zlib*. https://www.euccas.me/zlib.

[8] *Manuale iftop*. https://linux.die.net/man/8/iftop.

[9] *Codice sorgente completo dell'applicazione*. https://github.com/uThings/Study-and-development-of-a-LIDAR-remote-control-system-on-a-drone.

# Appendice A

# Istruzioni operative

I file sorgente dell'applicazione devono essere organizzati come mostrato in figura A.1.

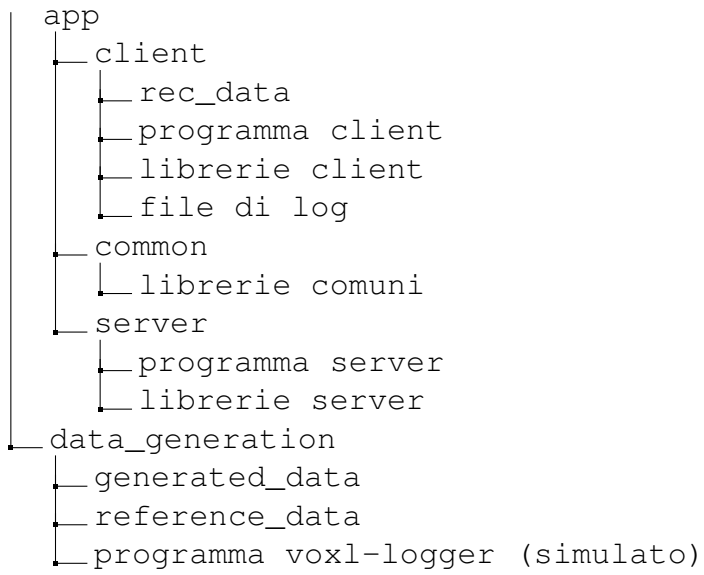


Figura A.1: Organizzazione dei file sorgente

Per compilare il programma `voxl-logger` non sono necessarie librerie, basta infatti spostarsi nella cartella `data_generation` e lanciare il comando:

```
$ gcc -o voxl-logger voxl-logger.c voxl-logger_lib.c
```

## A.1 Server

Le libreria da installare per il Server è la libreria `zlib`. Il comando da utilizzare in un sistema Linux Debian-based (come Ubuntu) è il seguente:

```
$ sudo apt install zlib1g-dev
```

Per quanto riguarda la compilazione, è stato scritto un `Makefile`, quindi è spostarsi nella cartella `server` e utilizzare il comando `make`.

Per lanciare il programma basta eseguire il comando seguente (supponendo di essere nella cartella `server`):

```
$ ./server 0.0.0.0 10000
```

In questo caso è stata usata la porta 10000.

## A.2 Client

Le librerie da installare per il Client sono le librerie `zlib` e `ncurses`. Il comando da utilizzare in un sistema Linux Debian-based (come Ubuntu) è il seguente:

```
$ sudo apt install zlib1g-dev libncurses5-dev
```

Anche in questo caso è stato scritto un `Makefile`, quindi per la compilazione è sufficiente spostarsi nella cartella `client` e lanciare il comando `make`.

Per lanciare il programma basta eseguire il comando seguente (supponendo di essere nella cartella `server`):

```
$ ./client 127.0.0.1 10000
```

In questo caso l'indirizzo ip utilizzato è quello di `loopback`, utile a testare i programmi di rete. La porta specificata è chiaramente la stessa scelta per il Server.

Non è necessario che il Server sia lanciato prima del Client, infatti quest'ultimo ha un meccanismo di recupero della connessione che permette di connettersi appena il Client è disponibile.

Il file di log si trova nella cartella del Client ed è nominato `lidar_control.log`.