



SAPIENZA
UNIVERSITÀ DI ROMA

SmartBerry Smart Home using Raspberry Pi.

Laureando
Antonio Grieco

Relatore
Giorgio Grisetti



SAPIENZA
UNIVERSITÀ DI ROMA



SAPIENZA
UNIVERSITÀ DI ROMA

SmartBerry Smart Home using Raspberry Pi.

Facoltà di Ingegneria dell'informazione, informatica e statistica
Dipartimento di Ingegneria informatica automatica e gestionale Antonio Ruberti
Corso di laurea in Ingegneria informatica ed automatica

Antonio Grieco
Matricola 1715585

Relatore
Giorgio Grisetti

A.A. 2020-2021

Stay Hungry, Stay Foolish.

Steve Jobs.



Ringraziamenti

Prima di iniziare con l'elaborato intendo ringraziare alcune persone fondamentali nel mio percorso. Un grazie speciale va al mio relatore, il Professore Giorgio Grisetti per avermi dato l'opportunità di realizzare questo progetto. Ringrazio poi i miei genitori, per essere stati sempre presenti, nei momenti felici e in quelli meno felici, per avermi supportato sempre, nonostante le difficoltà. Ringrazio Marco, mio fratello, per tutte le serate trascorse a spronarmi, a dirmi di non mollare, che un giorno le soddisfazioni sarebbero arrivate. Ringrazio la mia fidanzata, Aurora, per avermi supportato, ma soprattutto sopportato in questo percorso, trovando sempre una parola di conforto che mi ha aiutato a raggiungere questo importante traguardo. Ringrazio infine i miei colleghi universitari e gli amici tutti per le serate spensierate trascorse insieme, senza le quali non avrei trovato

la forza e la volontà di andare avanti. L'ultimo ringraziamento va banalmente a me, per averci creduto fino in fondo, per non aver mai mollato, nonostante tutto. **GRAZIE.**

Indice

Introduzione	11
1. Raccolta dei requisiti.....	13
2. Tecnologie e metodologie utilizzate.....	14
2.1 Tecnologie utilizzate	14
2.1.1 RaspberryPi 3B+	14
2.1.2 Gpio.....	15
2.1.3 WiringPi	15
2.1.4 IEEE Standard C	16
2.2 Metodologia utilizzata ed organizzazione del lavoro	17
3. Analisi del sistema	18
3.1 Funzionamento dei file codice e file header del Client	19
3.1.1 Client.c.....	19
3.1.2 Client.h	23
3.2 Funzionamento dei file codice e file header del Server	24
3.2.1 Server.c	24
3.2.2 Server.h	29
3.2.3 Temperature.c	29
3.2.4 Temperature.h	32
3.2.5 UltrasonicDetection.c.....	33
3.2.6 UltrasonicDetection.h	35
3.2.7 Blink.c	36
3.2.8 Blink.h.....	36
3.2.9 PWMLed.c	37

3.2.10 PWMLed.h.....	38
3.3 Funzioni comuni	38
3.3.1 Le famose send e recv	38
3.3.2 Gestione degli errori e di debug.....	39
3.3.3 Signal Handler	40
4. Compilazione del sistema.....	41
4.1 Compilazione lato Client	41
4.1.1 Makefile del client	41
4.2 Compilazione lato Server	42
4.2.1 Makefile del server.....	42
5. Realizzazione del sistema	43
5.1 Il progetto nel dettaglio.	43
6. Dispiegamento e validazione.....	48
6.1 Distribuzione ed installazione dell'applicazione.....	48
6.1.1 Il primo avvio, abilitazione SSH.	48
6.1.2 Installazione di WiringPi.....	49
Conclusione	50

Introduzione

Il lavoro di tesi proposto riguarda la progettazione e lo sviluppo di un sistema che emula un dispositivo Smart per la gestione domotica della casa. La scelta per l'elaborazione della relazione tecnica finale è ricaduta su questo argomento perché racchiude al suo interno diversi argomenti affrontati durante il corso di studi ma allo stesso tempo permette di approfondirli e integrarli con diverse tecnologie. L'idea di progetto è nata per mostrare come sfruttando un'architettura Server-Client sia possibile creare un'infinità di soluzioni e progetti completamente diversi fra loro, l'unico limite è la fantasia. In questo caso specifico tratteremo come costruire una rudimentale centrale IoT per rendere domotica la propria casa. Nel corso del primo capitolo daremo una descrizione generale dell'idea di progetto in sé e delle azioni che l'utente può compiere. Nel secondo capitolo tratteremo le tecnologie e metodologie utilizzate e dei componenti hardware essenziali per la realizzazione del progetto. Il terzo capitolo tratterà in maniera più approfondita il funzionamento dei vari file che compongono la struttura del sistema stesso, quindi i vari codici scritti per reggere e supportare la struttura del progetto. Nel quarto capitolo verrà mostrato come rendere autonomo il processo di compilazione ed esecuzione del sistema complessivo attraverso la struttura del `Makefile`. Il quinto capitolo, invece, tratterà, in maniera approfondita le varie operazioni presenti nel sistema e di come esse funzionano. L'ultimo capitolo tratta la fase preliminare alla realizzazione del progetto, come l'installazione di pacchetti fondamentali per il corretto funzionamento del progetto.

1. Raccolta dei requisiti

Il progetto di tesi svolto consiste nella progettazione e nello sviluppo di un sistema che emula il comportamento di un device IoT, il tutto basato sulla comunicazione tra client e server. Il progetto può essere scaricato e testato se si è in possesso degli strumenti hardware necessari. E' di fatto necessario un computer a scheda singola ovvero la Raspberry Pi ed alcuni componenti da collegare ai Pin della stessa. Per il progetto la scheda svolgerà un ruolo di particolare importanza, infatti gli verrà assegnato il compito di fare da server. Quest'ultimo una volta avviato si troverà in attesa ciclica e infinita di connessioni provenienti dall'esterno. Qualsiasi altro dispositivo in possesso dell'indirizzo IP del server sarà capace di connettersi per usufruire dei servizi offerti. L'utente una volta connesso si troverà davanti una lista di operazioni selezionabili attraverso la riga di comando. (Si è esplicitamente scelto di non utilizzare nessun tipo di interfaccia grafica in modo da lavorare a più basso livello). A livello di implementazione, tutte le operazioni selezionabili saranno gestite lato server, per cui dalla raspberry stessa. Questa è una parte fondamentale del progetto poiché le varie operazioni non sono appoggiate al client, nel senso che non andranno a sfruttare le risorse possedute dai vari utenti, evitando l'aspetto dei requisiti minimi. Ragion per cui gli unici controlli effettuati lato client sono controlli su ciò che possiamo inviare al server per evitare di creare possibili malfunzionamenti o bug. Il server sarà in grado di eseguire diverse azioni, come l'accensione o il lampeggiamento di un Led, la rilevazione della temperatura e dell'umidità dell'ambiente ed avviare un rudimentale sistema di allarme basato su un modulo ad ultrasuoni che rileva costantemente la distanza, nel caso in cui si verifichi un cambiamento nella rilevazione l'utente sarà informato tramite un messaggio ed attraverso una e-mail inviata automaticamente dalla raspberry.

2. Tecnologie e metodologie utilizzate

2.1 Tecnologie utilizzate

2.1.1 RaspberryPi 3B+



Fig1. Common RaspberryPi 3B+

Il nucleo, a livello hardware, di questo progetto, è questo fantastico computer a scheda singola di nome RaspberryPi. Si tratta praticamente di un mini computer della grandezza di pochi centimetri. La scheda è stata progettata per ospitare sistemi operativi basati sul kernel Linux e per la quantità enorme di feedback positivi vi è stato concepito un sistema operativo appositamente dedicato chiamato Raspberry Pi OS. Quindi questa scheda possiede tutto ciò che un pc ha; Monta un processore, una scheda video, memoria (SDRAM) per questa versione 1GB, quattro porte usb da 2.0 e tante altre peculiarità. Per farla funzionare, basta una semplice scheda SD, che avrà la funzione di boot e di storage, ed un alimentatore di alta qualità da 2.5 Ampere collegato alla scheda tramite un triviale cavo micro USB. La scheda è oltretutto dotata di porta HDMI per il collegamento in modo da essere utilizzata sfruttando la carina interfaccia grafica integrata del SO. Precisamente il modello utilizzato è il 3B+, ma il progetto è estendibile anche ad altre versioni.

2.1.2 Gpio

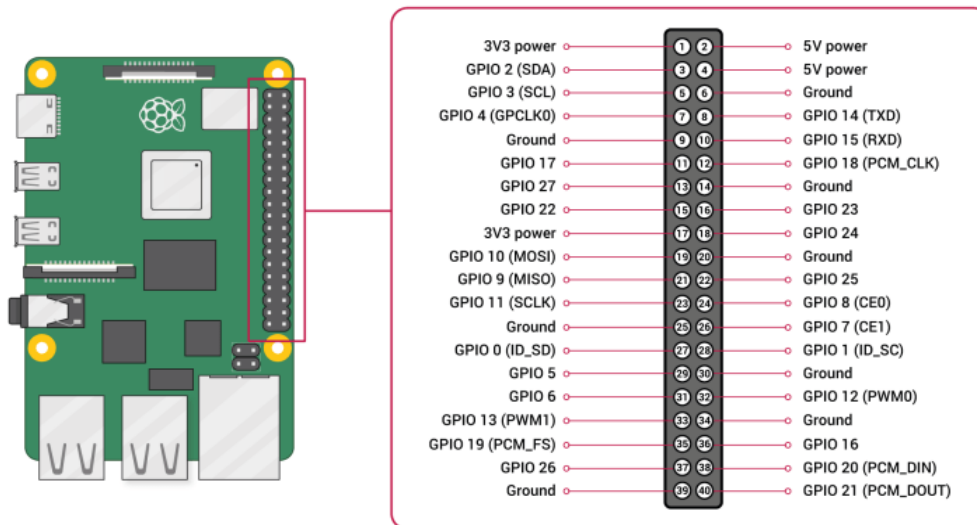


Fig2. Pin Board

L'intero progetto si basa sui Pin di cui è dotata la raspberry, sono montati su di essa di diverse tipologie, come possiamo vedere dalla figura in alto, i pin 3.3V e 5V forniscono una tensione ed una corrente massima in uscita, utilizzata per alimentare altri dispositivi. I moduli GPIO (General Purpose INPUT OUTPUT) servono per interfacciare la scheda con il mondo esterno, offrendo la possibilità di leggere e scrivere dati. Alcuni di questi ultimi possono essere utilizzati attraverso una tecnologia chiamata PWM (Pulse Width Modulation) che ci permette di controllare i componenti lavorando sulla modulazione dell'ampiezza degli impulsi. Troviamo infine i pin di tipo Ground che fungono come dei collegamenti a massa per il controllo della tensione minima.

2.1.3 WiringPi

WiringPi è una libreria che ci permette di accedere ai PIN GPIO scritta in linguaggio C per le schede BCM2835, BCM2836 e successive, montate sulla Raspberry Pi. Offre una valida alternativa alla tecnologia proprietaria BCM della casa produttrice per la maggiore comodità d'uso e per la semplicità con cui gestisce i canali di comunicazione fra la scheda e i dispositivi esterni. Per installare questa libreria sul nostro mini computer ci occorre digitare alcuni semplici comandi da terminale, come per qualsiasi altro SO

Linux. Una cosa che occorre subito far notare è che WiringPi utilizza un ordinamento dei pin differente da quello BCM(proprietario),bisogna per cui prestare attenzione a definire nel codice il corretto numero di Pin che si intende utilizzare,essendo differente dall'ordine che si legge fisicamente sulla scheda.Ci offre un metodo veloce ed efficace per il controllo dei moduli GPIO installati sulla Raspberry. Mette a disposizione diverse funzioni utilizzabili per creare dei canali di input ed output,per creare un canale digitale utilizzato per controllare i dispositivi collegati sfruttando la tecnologia PWM,funzioni per "pulire" i pin dopo l'utilizzo e molto altro.Dispone oltretutto di una comoda interfaccia a linea di comando denominata GPIO per il controllo e la gestione dei vari pin e per leggere o scrivere su un determinato canale direttamente da riga di comando,è possibile anche avere un prospetto dei pin e della numerazione adottata,digitando "**gpio readall**" ci verrà restituita una schermata che mostra tutti i pin,con le diverse numerazioni adottate.

Per utilizzare le funzioni di libreria offerte è necessario aggiungere nel nostro codice l'istruzione "*#include <wiringPi.h>*".

2.1.4 IEEE Standard C

Tutto ciò che è racchiude questo progetto è stato interamente pensato, scritto e sviluppato tramite il linguaggio C. Il linguaggio C sviluppato da Dennis Ritchie è stato creato con lo scopo della stesura del sistema operativo Unix. Dopo la pubblicazione di quest'ultimo si è decisi di standardizzarlo e da lì è diventato uno dei linguaggi di programmazione basilari per le tecnologie odierne. E' rinomato per la sua efficienza e si è imposto come linguaggio di riferimento per la realizzazione di software di sistema su gran parte delle piattaforme hardware moderne. Si può definire il linguaggio C come il padre di linguaggi che ultimamente stanno crescendo di importanza come C++ Java e C#. Per questa sua grandissima importanza storica è stato deciso di sviluppare interamente il lavoro con questo linguaggio di programmazione.

2.2 Metodologia utilizzata ed organizzazione del lavoro

La progettazione e lo sviluppo del sistema SmartBerry è stata svolta interamente da me e visionata dal relatore. Alcune porzioni di codice per la comunicazione Client-Server sono state prese da un vecchio progetto universitario per il corso di Sistemi Operativi, sviluppato insieme a due amici e colleghi universitari. Durante lo sviluppo si è utilizzata la shell nativa della Raspberry Pi, tramite SSH. Esso è un protocollo che permette di stabilire una sessione remota cifrata tramite interfaccia a riga di comando con un altro host di una rete informatica, operazione facilmente eseguibile se ci si trova nella stessa rete come nel nostro caso, è vivamente consigliato l'uso di una connessione cablata nella fase di sviluppo per una maggiore stabilità. Il sistema con il quale si è effettuato il collegamento è una macchina virtuale con su installata una distribuzione Linux, con SO Ubuntu(64-bit), ma è possibile utilizzare qualsiasi Sistema Operativo; nel caso in cui il protocollo SSH non fosse integrato è possibile stabilire una connessione SSH sfruttando Software esterni disponibili gratuitamente. Il lavoro è stato quasi interamente svolto utilizzando Visual Studio Code, installato su una macchina virtuale Linux, per una maggiore comodità nella gestione dei diversi file. Si sconsiglia infatti di lavorare direttamente sulla scheda in quanto la memoria è limitata e si rischierebbe di incappare in fastidiosi rallentamenti. Per leggere modifiche e per necessità di compilazione e testing dell'applicazione sulla Raspberry Pi si è usato l'IDE Geany, molto leggero e compatto, ideale quando non si dispone di tanta potenza computazionale. Inoltre durante lo sviluppo è stato sfruttato il sistema dei repository che offre GitLab. Di fatto il codice dell'applicazione è completamente disponibile su GitLab al seguente link: <https://gitlab.com/Grieco96/smartberryhome>.

3. Analisi del sistema

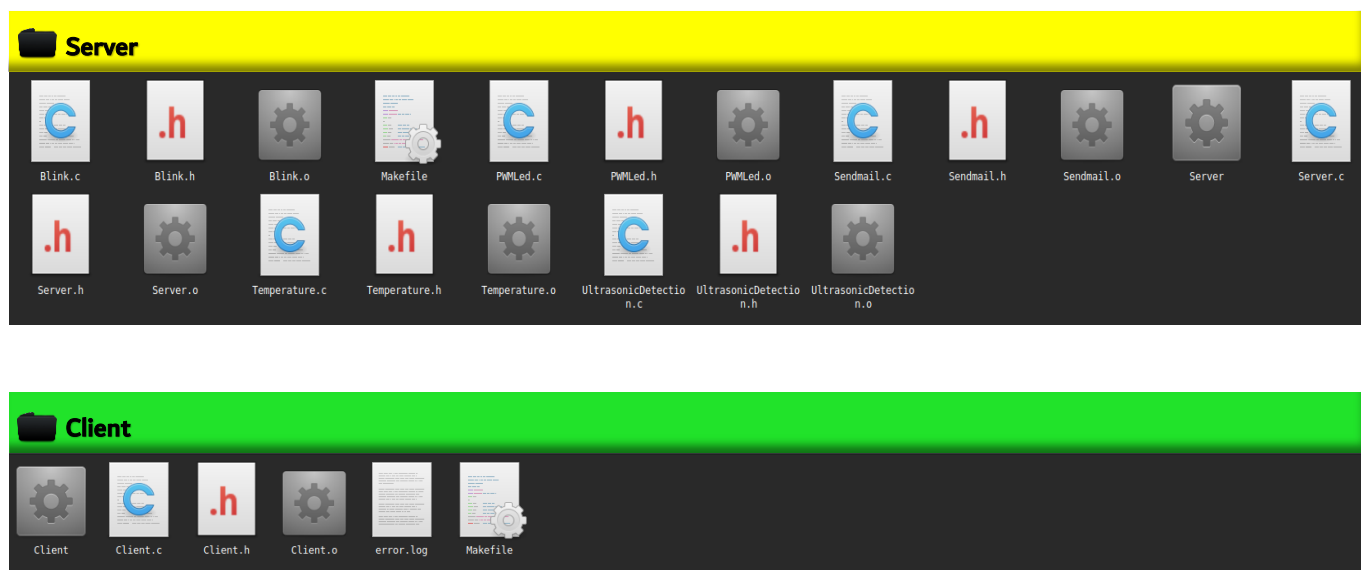


Fig3. File-System Client & Server

Il progetto è costituito da due cartelle differenti, la cartella server racchiude tutte le funzioni necessarie per la gestione dei vari collegamenti oltre ai metodi per il controllo e l'esecuzione dei vari dispositivi smart. Il client invece presenta una struttura molto più leggera, in quanto lui si occupa solo di inviare richieste al server ed attenderne l'esito. Si è optato per questa scelta per ottenere una maggiore portabilità, trattandosi di una smart home l'obiettivo è di rendere la vita più semplice, di conseguenza una maggiore adattabilità del codice ci porta ad un maggiore beneficio in termini di utenza raggiunta. Questo rende idealmente possibile l'esecuzione del codice anche da uno smartphone, utilizzando i giusti software, in quanto il client non necessita di installare alcuna libreria particolare, se non le librerie standard offerte dal linguaggio C.

Da qui in avanti andremo ad analizzare le porzioni di codice più interessanti, spiegando nel dettaglio come si è arrivati alla conclusione, il perché si è deciso di implementare in questo modo, analizzando diversi aspetti, come ad esempio la gestione della memoria, delle connessioni e così via.

3.1 Funzionamento dei file codice e file header del Client

3.1.1 Client.c

Il file client.c presenta una semplice ed importante proprietà: non ha bisogno di nessuna particolare libreria per poter funzionare, sfrutta solo librerie standard. Esso è costituito dal classico main nel quale andremo a gestire tutta la connessione con il server e nel quale inseriremo alcuni controlli per controllare l'inserimento dei dati. Nel codice abbiamo inizializzato diverse variabili, alcune di tipo locali e altre di tipo globale. Quelle globali sono necessarie per rendere i valori del descrittore del socket e dei file visibili da tutte le funzioni del sistema. La porzione di codice più interessante del client è quella legata al collegamento con il server:

```
// Variables for handling a socket

struct sockaddr_in server_addr = {0};

// Create a socket

socket_desc = socket(AF_INET, SOCK_STREAM, 0);
ERROR_HELPER(socket_desc, "Could not create socket");

// Set up parameters for the connection

server_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(SERVER_PORT);

// Initiate a connection on the socket
ret = connect(socket_desc, (struct sockaddr*) &server_addr, sizeof(struct sockaddr_in));
ERROR_HELPER(ret, "Could not create connection");
if (DEBUG)
    fprintf(stderr, "Connection established!\n");
```

Infatti per potersi connettere ad un server il client deve creare un socket attraverso la funzione `socket()`, che restituirà il descrittore dello stesso. I parametri passati a questa funzione riguardano rispettivamente: la famiglia di indirizzi che dovrà interpretare, il paradigma di connessione ed il valore 0 che corrisponderà al tipo di protocollo da utilizzare, in questo caso di default. Successivamente sono stati settati i parametri della struct `server_addr`, struttura dati che ci permette di manipolare gli indirizzi Ip nel seguente modo:

- Tramite `inet_addr`, funzione che converte la stringa passata come parametro nella notazione decimale standard dell' IPv4
- `AF_INET` la famiglia di indirizzi Ip

I principali controlli effettuati dal client sono per lo più collegati alla forma del tipo di messaggi che sceglie di inviare al server. Ovvero viene effettuato un controllo sulla forma della parola scritta da parte dell'utente. Di seguito il codice che mostra come i precedenti controlli vengono messi in atto:

```
//MAIN LOOP
while(1){
do{
// Printf option

if(control)
printf("Ops... \nInsert valid option, please.\n");
control = FALSE;
printf("-Blink\n-PwmLed\n-Temperature\n-Ultrasonic\n-Help\n-Quit\nInsert option: "); // <----- INSERIRE FUNZIONI RASPY!!!!

// Clear buf

memset(buf, 0, buf_len);
if(DEBUG)
fprintf(stderr, "buf: %s\n", buf);

// Read from stdin(Keyboard Input)

if (fgets(buf, sizeof(buf), stdin) != (char*)buf) {
fprintf(stderr, "Error while reading from stdin, exiting...\n");
exit(EXIT_FAILURE);
}
msg_len = strlen(buf);
buf[--msg_len] = '\0'; // Remove '\n' from the end of the message for checking
if(DEBUG)
fprintf(stderr, "buf: %s\n", buf);

// To lower case

for ( i = 0; i < msg_len; ++i){
buf[i] = tolower(buf[i]);
}
control = TRUE;

// MAIN OPTION WHILE
}while(!(msg_len == quit_command_len && !memcmp(buf, quit_command, quit_command_len)) && //not QUIT
!(msg_len == BLINK_LEN && !memcmp(buf, BLINK, BLINK_LEN)) &&
!(msg_len == PWMLED_LEN && !memcmp(buf, PWMLED, PWMLED_LEN)) &&
!(msg_len == TEMP_LEN && !memcmp(buf, TEMP, TEMP_LEN)) &&
!(msg_len == HELP_LEN && !memcmp(buf, HELP, HELP_LEN)) && // <---INSERIRE QUI FUNZIONI RASPY
!(msg_len == ULTRASONIC_LEN && !memcmp(buf, ULTRASONIC, ULTRASONIC_LEN))

); // Checking request
```

Il seguente codice cerca di ottenere l'input dell'utente grazie alla funzione *fgets()*, che si occupa di catturare la stringa immessa dall'utente da tastiera tramite lo standard input(stdin).Successivamente verrà eliminato il terminatore di stringa,un carattere speciale che indica la fine di una parola,viene poi invocata la funzione *tolower()* per trasformare tutti i caratteri presenti nel buffer in minuscolo.Entrambi questi controlli vengono eseguiti per rendere più semplice l'immissione dei comandi,non essendo case-sensitive, e per avere un riscontro più veloce sulla correttezza.Infine viene confrontato con gli identificatori delle operazione attraverso delle espressioni con gli operatori logici;praticamente,viene confrontata sia la lunghezza della stringa che i bytes in memoria sfruttando la funzione *memcmp()*,per avere un doppio controllo in modo da evitare fastidiosi bug.Quando la parola inserita corrisponderà ad una delle operazioni selezionabili si entrerà in una delle possibili scelte del costrutto IF-ELSE. La successiva porzione di codice che tratteremo riguarda il comportamento per lo scambio dei messaggi fra il Server ed il Client,visti dal lato del secondo;Per prima cosa viene liberato il buffer,subito dopo ci si mette in attesa del messaggio sfruttando la funzione *recv()*,viene letto e stampato a schermo,se necessario chiameremo la funzione *fgets()* per immettere ciò che ci viene richiesto e si invia la risposta al Server,a questo punto ci si metterà in attesa della risposta,che può avere esito positivo o meno. Si riporta in basso solo una porzione di codice,più precisamente quella legata alla funzione Blink,essendo tutte molto simili fra loro:

```

else if (msg_len == BLINK_LEN && !memcmp(buf, BLINK, BLINK_LEN)){

    //Clear buf

    memset(buf , 0 , buf_len);
    if(DEBUG)
        fprintf(stderr, "buf: %s\n", buf);

    //Receive Blink Mess

    while ( (msg_len = recv(socket_desc, buf, buf_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        ERROR_HELPER(-1, "Cannot read from socket");
    }

    printf("%s", buf);
    //Clear buf
    memset(buf , 0 , buf_len);
    if(DEBUG)
        fprintf(stderr, "buf: %s\n", buf);

    memset(buf , 0 , buf_len);
    if (fgets(buf, sizeof(buf), stdin) != (char*)buf) {
        fprintf(stderr, "Error while reading from stdin, exiting...\n");
        exit(EXIT_FAILURE);
    }

    msg_len = strlen(buf);
    buf[--msg_len] = '\0'; // remove '\n' from the end of the message
    if(DEBUG)
        fprintf(stderr, "buf: %s\n", buf);

    // Send BlinkONOFF to server
    while ( (ret = send(socket_desc, buf, msg_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        ERROR_HELPER(-1, "Cannot write to socket");
    }

    //Clear buf

    memset(buf , 0 , buf_len);
    if(DEBUG)
        fprintf(stderr, "buf: %s\n", buf);

    // waiting for "Well done" message

    memset(buf , 0 , buf_len);
    while ( (msg_len = recv(socket_desc, buf, buf_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        ERROR_HELPER(-1, "Cannot read from socket");
    }
}

```

Quando l'utente avrà terminato le richieste per il server e vorrà chiudere la connessione verrà eseguito il codice per chiudere il socket:

```
// Close the socket
}
ret = close(socket_desc);
ERROR_HELPER(ret, "Cannot close socket");

if (DEBUG){
    ret = close(fd);
    ERROR_HELPER(ret, "Could not close error.log");
}

printf("Exiting...\n");

exit(EXIT_SUCCESS);
```

3.1.2 Client.h

Questo file rappresenta il file header del corrispettivo .c .Al suo interno troviamo le macro per la gestione degli errori,molto utili in fase di revisione.Vi troviamo poi definite alcune costanti,come l'indirizzo Ip della Raspberry,fondamentale per stabilire la connessione,troviamo poi la definizione delle varie operazioni e relativa lunghezza,ricavata sfruttando la funzione *strlen()*,tutto ciò per aumentare la leggibilità del codice.

3.2 Funzionamento dei file codice e file header del Server

3.2.1 Server.c

E' il codice portante di questo progetto, al suo interno è contenuto il codice che mostra come il server gestisce le connessioni, come garantisce la mutua esclusione sulle risorse per impedire a diversi utenti di richiamare la stessa funzione insieme e creare malfunzionamenti, ed infine, come vengono gestite le chiamate alle funzioni e l'implementazione degli algoritmi necessari al funzionamento delle stesse. Il server sfrutta il paradigma multithread, il che permette la connessione simultanea di più utenti, poiché ognuna delle quali è gestita da un thread differente, che si occuperà di processare la richiesta. Il codice si compone dalla funzione main, necessaria alla realizzazione del costrutto multithread. Di fatto è stata creata una struttura dati personalizzata che si occuperà della creazione dei thread.

```
//Semaphore
sem_t Rasputin;

//struct thread arg
typedef struct handler_args_s{
    int socket_desc;
    struct sockaddr_in *client_addr;
} handler_args_t;
```

Nel main si trova l'inizializzazione del semaforo, inizializzato come variabile globale, esso costituirà l'elemento fondamentale per poter effettuare una operazione di mutua esclusione per evitare affollamenti su risorse non condivisibili. Come per il client abbiamo anche qui la fase iniziale di preparazione del socket. Si può vedere che viene usata la struttura dati `sockaddr_in` per le socket TCP/IP, essa prevede le stesse variabili passate come parametri come nel caso del client. In più abbiamo abilitato il `reuseaddress` per avere un riavvio del nostro server più performante in caso di crash. Bisogna poi associare la porta di connessione al socket, quindi costruire l'indirizzo completo utilizzando la funzione `Bind()`, ci si mette infine in attesa di eventuali connessioni.


```

//Variables
int ret;
int socket_desc, client_desc;
struct sockaddr_in server_addr = {0};
int sockaddr_len = sizeof(struct sockaddr_in);

//Semaphore initialized

ret=sem_init(&Rasputin,0,1);
ERROR_HELPER(ret,"Failed to initialize semaphore");

// Initialize socket for listening

socket_desc = socket(AF_INET, SOCK_STREAM, 0);
ERROR_HELPER(socket_desc, "Could not create socket");

server_addr.sin_addr.s_addr = INADDR_ANY; // Want to accept connections from any interface
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);

// Enable SO_REUSEADDR to quickly restart server after a crash

int reuseaddr_opt = 1;
ret = setsockopt(socket_desc, SOL_SOCKET, SO_REUSEADDR, &reuseaddr_opt, sizeof(reuseaddr_opt));
ERROR_HELPER(ret, "Cannot set SO_REUSEADDR option");

// Bind address to socket
ret = bind(socket_desc, (struct sockaddr *)&server_addr, sockaddr_len);
ERROR_HELPER(ret, "Cannot bind address to socket");

// Start listening
ret = listen(socket_desc, MAX_CONN_QUEUE);
ERROR_HELPER(ret, "Cannot listen on socket");

// Allocate client_addr dynamically and initialize it to zero
struct sockaddr_in *client_addr = calloc(1, sizeof(struct sockaddr_in));

```

Per gestire le connessioni in entrata è stato sfruttato un loop infinito che resta in attesa di possibili utenti, non appena qualcuno richiede una connessione viene generato un nuovo thread per gestirla, si accetta la connessione, viene creata una istanza della struttura dati **handler_args** e le si passano alcuni argomenti utili che verranno poi trasferiti al thread attraverso la chiamata alla funzione *pthread_create()*. Come ultima istruzione del ciclo troviamo la creazione di un nuovo buffer per la l'indirizzo del client, siccome un semplice reset dei campi non basta in questo caso.

```
// Loop to manage incoming connections spawning handler threads

while (1){
    // Accept incoming connection

    client_desc = accept(socket_desc, (struct sockaddr *)&client_addr, (socklen_t *)&sockaddr_len);
    if (client_desc == -1 && errno == EINTR) continue; // Check for interruption by signals
    ERROR_HELPER(client_desc, "Cannot open socket for incoming connection");

    if (DEBUG)
        fprintf(stderr, "Incoming connection accepted...\n");

    pthread_t thread;

    // Put arguments for the new thread into a buffer
    handler_args_t *thread_args = malloc(sizeof(handler_args_t));
    thread_args->socket_desc = client_desc;
    thread_args->client_addr = client_addr;

    ret = pthread_create(&thread, NULL, connection_handler, (void *)thread_args);
    PTHREAD_ERROR_HELPER(ret, "Could not create a new thread");

    if (DEBUG)
        fprintf(stderr, "New thread created to handle the request!\n");






    ret = pthread_detach(thread); // I won't pthread_join() on this thread
    PTHREAD_ERROR_HELPER(ret, "Could not detach the thread");

    // Can't just reset fields: we need a new buffer for client_addr!

    client_addr = calloc(1, sizeof(struct sockaddr_in));
}

exit(EXIT_SUCCESS); // this will never be executed
```

All'interno della porzione di codice relativa alla funzione `connection_handler()` possiamo vedere effettivamente il processo di gestione delle operazioni. Come prima cosa ovviamente troviamo anche qui l'inizializzazione delle variabili necessarie al funzionamento e alla gestione del buffer di invio e ricezione dei messaggi, variabili temporanee utilizzate per le chiamate alle funzioni e così via. Il server come prima operazione invia il messaggio di benvenuto al client, mettendosi subito dopo in una attesa ciclica, ideata per lo scambio di messaggi continui con i vari client, situazione ricreata sfruttando il costrutto `while`. Nella sequenza logica di funzionamento di una connessione, nel momento in cui entriamo nella funzione `connection_handler()` il server aspetterà per la ricezione di un messaggio, che per i controlli effettuati dal client sarà per forza di cose corretto, ne legge il contenuto e sfruttando lo stesso algoritmo del client, cioè controllando la lunghezza del messaggio e l'ID dello stesso, entra nella voce corrispondente della struttura IF-ELSE ed esegue l'operazione desiderata in mutua esclusione. Le operazioni selezionabili sono 5:

-  Blink
-  PWMLed
-  Temperature
-  Ultrasonic
-  Quit

e saranno illustrate in dettaglio successivamente. Andremo ora ad analizzare il codice di una delle operazioni, come fatto per il client, dal punto di vista del server. In questa porzione di codice viene anche mostrata la gestione e l'accesso alle risorse in mutua esclusione, sfruttando la definizione di semaforo e utilizzando le funzioni `sem_wait()` & `sem_post()`, così facendo tutto il codice compreso fra esse potrà essere eseguito da un solo utente alla volta, evitando fastidiosi errori in grado di causare malfunzionamenti nel server.

```

else if(recv_bytes==TEMP_LEN && !memcmp(buf,TEMP,TEMP_LEN)){

    puts("SONO IN TEMP");

    // Clear buf

    memset(buf,0,buf_len);
    if(DEBUG)
        fprintf(stderr,"buf: %s\n",buf);

    // Wait on semaphore
    ret = sem_wait(&Rasputin);
    ERROR_HELPER(ret, "Wait on semaphore Rasputin");
    if(DEBUG)
        fprintf(stderr , "ON Semaphore\n");
    /*******
    *
    *   TEMPERATURE FUNCTION   *
    *
    *   *****/
    temperature(temperature_array);

    //Post on semaphore

    ret = sem_post(&Rasputin);
    ERROR_HELPER(ret, "Wait on semaphore Rasputin");

    // Initialize buf

    sprintf(buf,TEMP_MESS);
    msg_len=strlen(buf);
    if(DEBUG)
        fprintf(stderr,"buf: %s\n",buf);

    // Send TEMP mess
    while ((ret = send(socket_desc, buf, msg_len, 0)) < 0){
        if (errno == EINTR) continue;
        ret = close(socket_desc);
        // Free buffer
        free(args->client_addr);
        free(args);
        //Exit
        pthread_exit(NULL);
    }
}

```

Infine, quando il client terminerà la connessione, il server eseguirà il codice relativo alla chiusura del socket, al rilascio degli spazi di memoria allocati e alla terminazione del thread:

```
// Close socket
ret = close(socket_desc);
ERROR_HELPER(ret, "Cannot close socket for incoming connection");
if (DEBUG)
    fprintf(stderr, "Thread created to handle the request has completed.\n");

//Free buffer
free(args->client_addr);
free(args);

//Exit
pthread_exit(NULL);
//END OF CONNECTION_HANDLER FUNCTION
```

3.2.2 Server.h

Questo file rappresenta il file header del relativo file .c . Al suo interno vi sono definite le macro per la manipolazione degli errori, causati da particolari operazioni. Inoltre abbiamo settato alcuni parametri necessari al funzionamento del server come le variabili di identificazione delle operazioni che adotta il server e dei messaggi preconfigurati per aumentare la leggibilità del codice.

3.2.3 Temperature.c

Il codice che si trova all'interno di questo file ci illustra in dettaglio come la raspberry si interfaccia con il dispositivo DHT11 utilizzato per la rilevazione di temperatura e dell'umidità. Inizialmente il Server chiama la funzione principale *Temperature()* situata in questo file, che a sua volta chiamerà un'altra funzione, *read_data_dht()*, che si occuperà di avviare la routine di lettura dei dati, assicurandosi che effettivamente i dati siano validi, in caso contrario verrà restituita una stringa per avvertire l'utente di ciò e automaticamente verrà effettuata un'altra lettura. Come prima cosa si vanno ad includere alcune librerie standard, oltre a wiringPi che ci consentirà di creare un canale di I/O con il dispositivo, successivamente verranno eseguite le seguenti operazioni in questo ordine:

Arriva la chiamata alla funzione *Temperature()* da parte del server e viene eseguita la seguente porzione di codice:

```
void* temperature(int*array){

    printf( "Reading Temperature...\n" );
    int i;

    if ( wiringPiSetup() == -1 )
        exit( 1 );

    while ( 1 )
    {
        read_dht11_dat();          //Read data from DHT11

        if(ret[0] !=0 && ret[2] !=0) {
            printf( "Humidity = %d.%d %% Temperature = %d.%d C \n",ret[0],ret[1],ret[2],ret[3]);          //Check result
            for(i=0;i<4;i++){
                array[i]=ret[i];          //Filling result array
            }

            break;
        }

        delay( 1000 );
    }
}
```

Viene eseguita la funzione `wiringPiSetup()` che prepara i pin eseguendo una funzione di routine,in caso di esito positivo si entra nel loop e cosa prima cosa troviamo una chiamata alla funzione `read_dht11_data()` che esegue inizialmente il seguente codice:

```
void read_dht11_dat(){

    uint8_t laststate    = HIGH;
    uint8_t counter      = 0;
    uint8_t j            = 0, i;

    dht11_dat[0] = dht11_dat[1] = dht11_dat[2] = dht11_dat[3] = dht11_dat[4] = 0;

    //Setting pins

    pinMode( DHTPIN, OUTPUT );
    digitalWrite( DHTPIN, LOW );
    delay( 18 );
    digitalWrite( DHTPIN, HIGH );
    delayMicroseconds( 40 );
    pinMode( DHTPIN, INPUT );
}
```

La prima cosa da notare è che le variabili vengono definite sfruttando delle primitive e non le più classiche int,long ecc,questo a causa del fatto che avremo bisogno successivamente di esguire alcune operazioni di shift.Si trova inoltre la definizione del vettore dht11_dat,in cui verranno salvati i risultati della lettura.Subito dopo troviamo il settaggio dei Pin,inserendo fra le varie chiamate alle funzioni alcune attese per evitare di avere malfunzionamenti nella macchina.Si procede poi con il ciclo di lettura:

```
// Cycle for get data from DHT11

for ( i = 0; i < MAXTIMINGS; i++ )
{
    counter = 0;
    while ( digitalRead( DHTPIN ) == laststate )
    {
        counter++;
        delayMicroseconds( 1 );
        if ( counter == 255 )
        {
            break;
        }
    }
    laststate = digitalRead( DHTPIN );

    if ( counter == 255 )
        break;

    if ( (i >= 4) && (i % 2 == 0) )
    {
        dht11_dat[j / 8] <<= 1;
        if ( counter > 50 )
            dht11_dat[j / 8] |= 1;
        j++;
    }
}
```

In questa fase avviene l'effettiva lettura da parte del dispositivo, eseguendo all'interno di un ciclo for ripetute letture fino ad ottenerne una soddisfacente incrementando ad ogni iterazione il contatore j. Non appena il contatore avrà raggiunto un valore sufficientemente grande si avrà l'effettivo salvataggio della lettura nel vettore:

```
if ( (j >= 40) &&
    (dht11_dat[4] == ( (dht11_dat[0] + dht11_dat[1] + dht11_dat[2] + dht11_dat[3]) & 0xFF) ) )
{
    for(i=0;i<4;i++){
        ret[i]=dht11_dat[i];
    }
}
else {
    printf( "Wait a few seconds please...I'm reading data...\n" );
}
}
```

Con questa operazione termina la funzione read_dht11_data() e di conseguenza il puntatore delle operazione torna alla funzione Temperature(), che effettua dapprima un ulteriore controllo sulla correttezza dei dati e poi trasferisce gli stessi al server attraverso la variabile array che la funzione prende in input.

3.2.4 Temperature.h

File header del relativo file .c . Al suo interno abbiamo i prototipi delle funzioni che utilizzeremo nel file codice Temperature.c . I prototipi permettono al compilatore di produrre un codice oggetto che può essere facilmente unito con quello della relativa libreria in futuro, anche senza avere la libreria disponibile al momento, oltre a permettere di aggiungere facilmente funzioni al progetto senza impattarne la struttura.

3.2.5 UltrasonicDetection.c

Andiamo adesso ad analizzare il codice relativo alla funzione che emula un sistema di allarme sfruttando un modulo ad ultrasuoni per rilevare la distanza, non si tratta di un vero e proprio sistema di videosorveglianza bensì di un rilevamento continuo della distanza, che nel caso sia differente in due rilevazioni successive, invia una email all'utente informandolo che è stato rilevato del movimento. Ho scelto di sviluppare il sistema in questo modo perché lo scopo finale del progetto è di creare un Hub IoT a basso costo, introdurre una videocamera sarebbe stato decisamente più costoso. Quando il server invoca la funzione Ultrasonic(), viene eseguita la porzione di codice per la creazione di un apposito thread che si occuperà di gestire il loop di rilevazione continua della distanza, non è possibile eseguire la funzione senza sfruttare il costrutto multithread perché trattandosi di un ciclo infinito la funzione non termina mai, a meno che l'utente non lo decida esplicitamente inviando al server il segnale di terminazione, questo porterebbe il server a rimanere in attesa del ritorno della funzione per un tempo indefinito, invalidando completamente la sua funzione. Ecco il codice che mostra la creazione del thread:

```
int UltrasonicDetection(int On_Off){  
    int ret;  
    TEST=On_Off;  
  
    ret=pthread_create(&On_thread,NULL,LoopControl,NULL);    // Create loop thread  
    PTHREAD_ERROR_HELPER(ret, "Could not create the thread");  
  
    ret = pthread_detach(On_thread);    // Detach for reuse memory  
    PTHREAD_ERROR_HELPER(ret, "Could not detach the thread");  
}
```

Il thread appena creato si occuperà di lanciare la funzione LoopControl(), che si occuperà di lanciare effettivamente la routine di rilevazione continua, facendo uso di altre funzioni, ecco il codice in dettaglio:

```

void* LoopControl(void* arg){
    int i=0;
    wiringPiSetup();
    pinMode(trigPin,OUTPUT);
    pinMode(echoPin,INPUT);
    float distance = getSonar();
    float old_distance;
    while(TEST==1){
        i++;
        old_distance=distance;

        distance = getSonar();

        if(distance!=old_distance){
            puts("ALLARME");
            SendMail();                // Send an alarm email to the owner
            puts("DONE");
            pthread_exit(NULL);
        }
        delay(100);
    }
    pthread_exit(NULL);
}

```

In questa parte della funzione vengono inizializzati i pin utilizzati dal dispositivo, uno in Input e l'altro in Output, la variabile per la rilevazione della distanza usando la funzione getSonar() fatta in questo modo:

```

float getSonar(){                // Get the measurement result of ultrasonic module in cm
    long pingTime;
    float distance;
    digitalWrite(trigPin,HIGH); // Send 10us high level to trigPin
    delayMicroseconds(10);
    digitalWrite(trigPin,LOW);
    pingTime = pulseIn(echoPin,HIGH,timeOut); // Read plus time of echoPin
    distance = (float)pingTime * 340.0 / 2.0 / 10000.0; // Calculate distance with sound speed 340m/s
    return distance;
}

```

Questa funzione si occupa in poche parole di elaborare i dati del sensore, calcolando la distanza in centimetri. Lo fa misurando il tempo che impiega il suono a trovare un ostacolo e a ritornare indietro, il primo segnale viene inviato dal Trigger, l'Echo invece viene passato alla funzione pulseIn() che regola il tempo trascorso fra l'invio ed il

ritorno del suono,viene poi calcolata l'effettiva distanza fra gli oggetti attraverso la formula che si vede sopra.

Tornando alla funzione principale di questa sezione,LoopControl(),viene creato un ciclo while,che controlla se la variabile Test è settata ad uno,questa variabile viene passata direttamente dal server,0 se l'utente sceglie di spegnere il modulo,1 se desidera accenderlo,si è scelto questo metodo perché è quello meno dispendioso in termini di risorse.All'interno del ciclo viene controllato se la distanza è cambiata dall'ultima rilevazione,in caso affermativo,viene inviata una e-mail all'utente attraverso la funzione Sendmail(),che molto banalmente scrive su un file le istruzioni da eseguire da terminale e le fa eseguire attraverso il comando system(cmd),di seguito il codice:

```
void SendMail(){  
  
    char cmd[100]; // To use the command  
    char to[] = "typeyourmailhere@test.com"; // Email id of the owner  
    char body[] = "          -ATTENTION-\n      Movement revealed by Ultrasonic Alarm!!!"; // Alarm message  
    char alarmFile[100]; // Name of alarmFile.  
  
    strcpy(alarmFile,tempnam("/tmp","sendmail")); // Generate alarm file name  
  
    FILE *fp = fopen(alarmFile,"w"); // Open alarmFile for writing  
    fprintf(fp,"%s\n",body); // Write alarm message to it  
    fclose(fp); // Close alarm file  
  
    sprintf(cmd,"sendmail %s < %s",to,alarmFile); // Prepare sendmail command to send email  
    system(cmd); // Execute command on shell  
}
```

3.2.6 UltrasonicDetection.h

File header del file code UltrasonicDetection.c , nel quale sono definite le macro per la gestione degli errori e vi sono definite le funzioni utilizzate nel file.

3.2.7 Blink.c

Il file .c relativo a questa funzione è molto breve, in quanto il suo unico scopo è quello di attivare o disattivare un led, il codice utilizzato per fare ciò è molto semplice, in quanto inizializza i Pin attraverso la funzione `wiringPiSetup()`, crea un canale di output sul Pin al quale è collegato il Led ed in base a quello che l'utente desidera fare viene spento/acceso scrivendo attraverso la funzione `digitalWrite()` un valore basso/alto. Di seguito il codice per implementare queste semplici operazioni:

```
#include <wiringPi.h>
#include <stdio.h>

#define ledPin 0 // Define the led pin number

void Blink(int ret){

    wiringPiSetup(); //Initialize wiringPi

    pinMode(ledPin, OUTPUT); //Set the pin mode

    if(ret==0){
        digitalWrite(ledPin,HIGH);
        printf("LED TURNED ON \n");
    }
    else if(ret==1){
        digitalWrite(ledPin,LOW);
        printf("LED turned OFF \n");
    }

}
```

3.2.8 Blink.h

File header del relativo .c, contiene la definizione dell'unica funzione utilizzata, `Blink()`.

3.2.9 PWMLed.c

Nel file PwmLed andremo ad analizzare il codice relativo all'attivazione del lampeggiamento dei Led sfruttando la tecnologia del PWM. In questo caso la situazione è più delicata della precedente in quanto bisogna avviare un Loop che scriva valori crescenti sul canale di comunicazione e successivamente decrescenti per simulare la comune situazione di una lampadina che lampeggia, per fare ciò sfruttiamo come nel caso della funzione Ultrasonic la struttura multithread, in modo da avere l'esecuzione del thread assegnato alla funzione in parallelo a quella del programma principale. Andiamo a vedere in dettaglio il corpo della funzione che si occupa della gestione dei thread:

```
void PWMLed(int set){
    int ret;

    tester=set;

    ret=pthread_create(&On_thread,NULL,pwm_go,NULL);    // Crete thread for loop pwm
    PTHREAD_ERROR_HELPER(ret, "Could not create the thread");

    ret = pthread_detach(On_thread);                    // Detach for reuse allocated memory
    PTHREAD_ERROR_HELPER(ret, "Could not detach the thread");

}
```

La situazione è analoga all'altra funzione che sfrutta questo paradigma, si inizializza la funzione tester per avere un controllo sullo spegnimento dei led, si crea il thread e viene invocata la funzione pthread_detach() per ottimizzare le risorse, rendendole riutilizzabili. Viene passata al nostro thread la funzione pwm_go() che presenta la seguente struttura:

```
#define ledPin    1

pthread_t On_thread;

int tester;

void* pwm_go(void* arg){

    wiringPiSetup();
    softPwmCreate(ledPin,0,100);
    int i;
```

Nella prima parte della funzione, come si può vedere, vengono definite le variabili utili, si inizializzano i Pin e si crea il canale di comunicazione che sfrutta la tecnologia PWM, successivamente troviamo un ciclo while, che controlla costantemente se la variabile tester è settata ad 1. Se il precedente controllo va a buon fine si entra nel corpo del ciclo che presenta due semplici cicli for che incrementano e decrementano rispettivamente un contatore, passato come variabile alla funzione softPwmWrite(), che scrive quel valore sul pin, rendendo più o meno luminoso il Led, in mezzo alla funzione troviamo diverse chiamate alla funzione delay(), sfruttate per regolare la velocità con cui si svolge l'operazione, di conseguenza passando un diverso valore si avrà un effetto diverso.

```
while(tester==1){  
    for(i=0;i<100;i++){           // Make the led brighter  
        softPwmWrite(ledPin, i);  
        delay(10);                // Delay for control speed of blinking  
    }  
    delay(100);  
    for(i=100;i>=0;i--){          // Make the led darker  
        softPwmWrite(ledPin, i);  
        delay(10);  
    }  
    delay(100);                  // Delay after a complete cycle  
}  
pthread_exit(NULL);              // Exit when tester go to 0  
}
```

3.2.10 PWMLed.h

File header del relativo file .c, contiene le macro per la gestione degli errori e l'inizializzazione delle due funzioni utilizzate.

3.3 Funzioni comuni

3.3.1 Le famose send e recv

Il server ed il client comunicano tra di loro attraverso due funzioni chiavi. Queste funzioni nel programma si ripetono in continuazione sia nei codici del server che del client. Questo paragrafo descriverà l'uso personale di queste funzioni all'interno del progetto.

- send: avvia la trasmissione di un messaggio dal socket specificato al suo peer

```
while ((ret = send(socket_desc, buf, msg_len, 0)) < 0){
    if (errno == EINTR)
        continue; ret =
    close(socket_desc);
    //free buffer
    free(args->client_addr);
    free(args);
    //exit
    pthread_exit(NULL);
}
```

- recv: riceve il messaggio dal socket di una connessione

```
while ((recv_bytes = recv(socket_desc, buf, buf_len, 0)) < 0){
    if (errno == EINTR)
        continue; ret =
    close(socket_desc);
    //free buffer
    free(args->client_addr);
    free(args);
    //exit
    pthread_exit(NULL);
}
```

3.3.2 Gestione degli errori e di debug

Durante la programmazione è molto frequente trovarsi davanti degli errori, sia in fase di compilazione che in fase di esecuzione. Per questo si è scelto di implementare nel progetto delle istruzioni necessarie a gestire eventuali errori dello standard error. Non solo, vengono utili anche variabili di debug per riuscire a capire tramite l'output la posizione degli stessi, in questo modo si risparmi tantissimo tempo in fase di testing, quando anche dei piccoli errori di battitura possono far perdere un sacco di tempo.

```

#define GENERIC_ERROR_HELPER(cond, errCode, msg) do {
    if (cond) {
        fprintf(stderr, "%s: %s\n", msg, strerror(errCode));
        exit(EXIT_FAILURE);
    }
} while(0)

#define ERROR_HELPER(ret, msg)          GENERIC_ERROR_HELPER((ret < 0), errno, msg)
#define PTHREAD_ERROR_HELPER(ret, msg) GENERIC_ERROR_HELPER((ret != 0), ret, msg)

```

Per la gestione degli errori abbiamo implementato la seguente macro, presente nei file header del client e del server, oltre che negli header di alcuni file relativi alle operazioni della raspberry, in modo tale da richiamarla immediatamente all'occorrenza, con una semplice riga di codice.

Parecchie volte nella lettura del codice noteremo queste due istruzioni ripetersi.

```

if(DEBUG)
    fprintf(stderr, "buf: %s\n", buf);

```

Questa funzione è collegata ad una variabile settata appositamente con lo scopo di funzionare come variabile di debug, essa è servita durante la costruzione del progetto per informarci sul tipo di messaggio che il server o il client stavano processando riuscendo a capire così la posizione nel codice e l'eventuale presenza di errori di comunicazione.

3.3.3 Signal Handler

Come per qualsiasi programma ci deve essere una combinazione di tasti per interrompere il processo. In informatica vengono chiamati segnali, cioè impulsi asincroni trasmessi da un processo ad un altro per poter comunicare tra loro. Di conseguenza nel Client è stata implementata la funzione di gestione dei segnali speciali. Per far capire il tipo di segnale bisogna sfruttare la libreria del linguaggio **signal.h**. E ai fini del progetto si è scelti di poter catturare il segnale di interruzione SIG_INT(ctrl + c). Per implementarlo è sufficiente inserire la seguente riga di codice:

```

(void) signal(SIGINT, signal_handler);

```

Da questo momento il programma è capace di catturare il segnale e di processarlo attraverso la funzione signal_handler, implementata in questo caso per chiudere eventuali descrittori di socket rimasti aperti.

4. Compilazione del sistema

La compilazione è l'operazione che consente di trasformare il codice sorgente del programma in un file eseguibile da parte di un computer. Per automatizzare la compilazione dei file sorgente l'ambiente UNIX mette a disposizione il comando make. In ogni directory che contiene sorgenti è sufficiente creare un file di testo, chiamato Makefile, che contiene le istruzioni per la compilazione. Una volta creato basterà digitare make sul terminale per avviare la compilazione. Una particolarità del make è che esso controlla se qualche file fra i prerequisiti è stato modificato dall'ultima compilazione, ed in caso affermativo ricompila i file modificati in modo automatico.

4.1 Compilazione lato Client

4.1.1 Makefile del client

Il Makefile del client è abbastanza semplice dato che non richiede nessun file sorgente esterno da cui dipendere ma solamente dal client.h, non essendo definite altre funzioni in questa cartella. Per questo il makefile di questa directory è composto da una manciata di righe.

```
CC = gcc

all:      Client final

Client:    Client.c Client.h
    $(CC) -c Client.c

final:
    $(CC) -o Client Client.o

clean:
    rm -f *.o Client

run:
    ./Client
```

Il funzionamento è semplice, si dichiarano alcune variabili che possono essere richiamate all'interno del pattern \$(). Per poter compilare i programmi si cerca di completare il documento seguendo questa struttura:

target : dipendenza1.c dipendenza2.c
comando per generare il target

4.2 Compilazione lato Server

4.2.1 Makefile del server

Il Makefile del server invece, presenta molte dipendenze da file sorgenti, in quanto tutte le operazioni sono gestite lato Server. Di seguito è riportata l'intera struttura del file, che presenta una struttura leggermente diversa, per consentire al file di compilare integrando le librerie `lpthread` e `wiringPi`, aggiungendo le rispettive regole per permettere il collegamento.

```
CC = gcc

THREAD = -lpthread

WIRINGPI= -lwiringPi

all:      Server Blink Temperature PWMLed UltrasonicDetection Sendmail final

Server:    Server.c Server.h
$(CC) -c Server.c

Temperature: Temperature.c Temperature.h
$(CC) -c Temperature.c

Blink:     Blink.c Blink.h
$(CC) -c Blink.c

PWMLed:    PWMLed.c PWMLed.h
$(CC) -c PWMLed.c

UltrasonicDetection: UltrasonicDetection.c UltrasonicDetection.h
$(CC) -c UltrasonicDetection.c

Sendmail:  Sendmail.c Sendmail.h
$(CC) -c Sendmail.c

final:
$(CC) -o Server Server.o Temperature.o PWMLed.o Blink.o UltrasonicDetection.o Sendmail.o $(THREAD) $(WIRINGPI)

.PHONY:clean

clean:
rm -f Server *.o
```

5. Realizzazione del sistema

5.1 Il progetto nel dettaglio.

Per rendere la lettura più semplice, si è scelto di optare per una suddivisione a colori, riportando una legenda iniziale.

● Server

● Client

- Inizialmente il server si prepara alla comunicazione, istanzia alcune variabili e le strutture dati per la socket, lo crea ed esegue il bind dell'indirizzo sulla porta e si mette in ascolto.
 - Il client come operazione iniziale crea un file log per eventuali errori, crea il socket e setta i parametri necessari per effettuare una connessione. Dichiarare alcune variabili importanti all'interno del main per poter gestire i messaggi che scambia con il server e si mette in attesa di ricevere, tramite la `recv()`, il messaggio di benvenuto da parte del server.
- Il server accetta la connessione in entrata, incaricando uno specifico thread di gestirla, da questo momento verrà trattata la fase di gestione della singola connessione. Vengono istanziate le variabili necessarie per le operazioni e successivamente si precarica nel buffer designato per la comunicazione, il messaggio di benvenuto da inviare al client, subito dopo lo spedisce attraverso la funzione `send()`. Successivamente in un loop infinito, utile per tornare al menu iniziale al completamento di qualsiasi operazione, si mette in attesa di ricevere l'id dell'operazione che l'utente ha scelto.
 - Il client riceve e stampa il messaggio di benvenuto per poi mettersi all'interno di un costrutto che si ripete all'infinito, per permettere all'utente di eseguire continuamente le operazioni che offre il server. Esso ci mostra le attività che può eseguire il server e ci chiede di inserire tramite shell,

l'operazione che vogliamo eseguire, che verrà salvata all'interno del buffer : `char buf[]` e poi inviato al server attraverso la `send()`. Verrà effettuato un controllo dell'input dell'utente, nel caso in cui l'ultima parola digitata non possa essere associata a nessun identificativo di operazione del server il client ritornerà a mostrare le attività di scelta e richiederà all'utente di digitare nuovamente per eseguire un'operazione. Si uscirà da questa fase solo se verrà inviato un messaggio corretto oppure con il segnale di interruzione `SIG_INT`.

Le operazioni selezionabili sono:

-Blink

-PWMLed

-Temperature

-Ultrasonic

-Help

-Quit

- In contemporanea, server e client, dopo che quest'ultimo ha inviato la stringa scritta dall'utente, effettuano un controllo sul tipo di operazione scelta, attraverso dei costrutti **if-else** (Esempio: *se codice_id_operazione_1 = stringa_utente esegui operazione_1 altrimenti se codice_id_operazione_2 = stringa_utente esegui operazione_2*). Verranno adesso discusse le singole operazioni.

Operazione Blink

- Il server come prima operazione libera il buffer e lo inizializza con il messaggio relativo alla funzione blink, per chiedere all'utente se si intende accendere o spegnere il led e lo si invia attraverso la funzione `send()`.

- Il client riceve il messaggio attraverso la funzione `recv()`, lo stampa a schermo ed aspetta che l'utente inserisca una risposta, rimuove il terminatore di stringa e lo invia attraverso la funzione `send()`, mettendosi poi in attesa di ricevere il messaggio di conferma.
- Il server riceve il messaggio, esegue un controllo sulla correttezza dei dati per impedire di richiamare una funzione con dei dati sbagliati rischiando di danneggiare i componenti, e chiede l'accesso alle risorse al semaforo tramite la `sem_wait()`, quando lo ottiene, entra nella zona critica e invoca la funzione `Blink()`, passandogli come parametro il valore ricevuto precedentemente dall'utente. Viene eseguita la funzione `Blink()`, si esce dalla zona critica tramite la `sem_post()`, si inizializza il messaggio di conferma e lo si invia al client che era rimasto in attesa di una risposta.

Operazione PWMLed

- Il processo per questa operazione è molto simile a quello visto precedentemente, la principale differenza la troviamo nella gestione della funzione, che avendo una natura multithread necessita di un ulteriore controllo, lo scambio di messaggi è pressoché identico essendo entrambe le operazioni sfruttate per controllare un led, precedentemente in modo classico e adesso tramite la tecnologia del PWM, per cui passeremo direttamente alla operazione successiva.

Operazione Ultrasonic

- Questa operazione è sicuramente la più interessante, in quanto emula un sistema di videosorveglianza ad un costo davvero irrisorio. Lo scambio dei messaggi è uguale ai precedenti, la differenza sta nella funzione che viene chiamata, `UltrasonicDetection()`.
- Il server dopo lo scambio di messaggi e dopo aver ottenuto l'accesso alle risorse condivise chiamerà la funzione `UltrasonicDetection()`, passandogli come parametro il valore ottenuto

dall'utente. La funzione chiamata avvierà il sistema di rilevamento ad ultrasuoni,che continuerà a rilevare la distanza in Loop,la particolarità sta nel fatto che quando noterà una differenza in due rilevazioni consecutive avvierà un procedura che chiamerà un'altra funzione,SendMail(),che avrà lo scopo di informare tramite email il proprietario di aver rilevato una differenza nelle misurazioni,il tutto automaticamente,tramite degli appositi comandi stampati su un file temporaneo e poi fatti eseguire a riga di comando tramite cmd(system).Dopo aver inviato l'email stamperà a schermo "ALLARME",rendendo più immediato l'avvertimento nel caso in cui ci si trovi in prossimità della Raspberry Pi.Anche questa funzione necessita del costrutto multithread,in quanto deve essere eseguita in background.

Operazione Temperature

- Per questa operazione ci saranno delle differenze anche nello scambio dei messaggi da parte,in quanto invocando l'operazione temperature non necessiteremo di ricevere un ulteriore valore da parte del client,ma potremo avviare direttamente la procedura che porta alla rilevazione.A differenza di quello che dice il nome questa funzione oltre alla temperature rileva anche l'umidità dell'ambiente sfruttando un modulo DHT11 da collegare al Raspberry Pi.
- Il server dopo aver ricevuto da parte del client il comando temperature,chiederà l'accesso alla zona critica, una volta ottenuto chiamerà la funzione relativa passandogli come parametro temperature_array[], un vettore che ci sarà utile per ottenere il risultato delle rilevazioni.Nel corpo della funzione Temperature() troviamo diverse valutazioni sulla correttezza dei dati,nel caso in cui tali operazioni vadano a buon fine otterremo un risultato che verrà salvato nel vettore risultato,che è lo stesso che abbiamo dato alla funzione inizialmente,se i controlli non vengono superati si ripete l'operazione fino ad ottenere una rilevazione soddisfacente. Adesso il server si occuperà di inizializzare la variabile TEMP_MESS all'interno del buffer,tale variabile conterrà i risultati dell'ultima rilevazione,prendendo i valori dalle varie posizione del vettore temperature_array[],oltre a

delle stringhe per indicare quale valore riguarda la temperature e quale l'umidità. Invierà poi i risultati all'utente, rimasto in attesa di un risultato, tramite la funzione `Send()`.

- Il client, una volta ricevuti i dati con le misurazioni tramite la funzione `recv()`, si occuperà di stamparli a schermo sfruttando la funzione `printf()` e passandogli come parametro il buffer `buf[]`.

Operazione Help

- Questa operazione è presente solo nel file client, in quanto viene sfruttato per informare l'utente sul funzionamento delle varie operazioni. Una volta entrati in questa fase ci verrà chiesto di digitare il nome dell'operazione della quale necessitiamo di conoscere maggiori dettagli, dopo aver digitato il nome di una delle operazioni verrà restituita a schermo una stringa di poche righe che descriverà il funzionamento dell'operazione scelta. L'implementazione di questo comando è molto semplice, essendo basata su un costrutto `if-else`, analogo a quello del corpo principale della funzione di gestione delle operazioni, allo stesso tempo aumenterà di molto la Quality of Life, avendo a portata di mano tutte le informazioni di cui si necessita.

Operazione Quit

- Quando l'utente digiterà questa operazione verrà avviata la procedura di Logout, verrà chiuso il socket utilizzato per la comunicazione, verranno eliminati eventuali descrittori rimasti aperti e verranno liberate eventuali variabili preallocate, per pulire la memoria da eventuali occupazioni. Queste operazioni verranno effettuate da entrambi i lati ma si noti bene che il comando `quit` non cesserà l'attività del Server, ma terminerà semplicemente la sessione corrente, essendo il Server progettato per gestire diverse connessioni simultaneamente.

6. Dispiegamento e validazione

6.1 Distribuzione ed installazione dell'applicazione

In questa sezione tratteremo come poter replicare il progetto, le operazioni preliminari da fare e vari consigli sulle modalità da utilizzare.

Tips & Tricks : per installare il sistema operativo consiglio di usare l'installer Raspberry Pi Imager scaricabile dal sito ufficiale della Raspberry il quale permette di installare su una scheda SD il sistema operativo personalizzato della Raspberry o altre distribuzioni unix-like.

6.1.1 Il primo avvio, abilitazione SSH.

Una volta avviata la raspberry è necessario eseguire i seguenti comandi da terminale per aggiornare i vari pacchetti del nostro mini-pc:

```
$ sudo apt-get update  
$ sudo apt-get upgrade  
$ sudo rpi-update  
$ sudo reboot
```

Successivamente prepariamo la raspberry ad abilitare sia il servizio ssh, che sfrutteremo per connetterci da remoto.

```
$ sudo raspi-config
```

Si aprirà una GUI :

- selezionare Advanced Option, premere il tasto Invio, andare sull'opzione "A4 SSH", premere invio, e nuovamente invio su "Yes" per abilitare il servizio. Una volta abilitato sarà sufficiente aprire una qualsiasi shell e collegarsi alla Raspberry tramite il seguente comando.

```
$ pi@192.168.x.x //indirizzo ip della raspberry
```

Default password : raspberry

N.B: Per rendere effettivi i cambiamenti è necessario un riavvio (*sudo reboot*).

6.1.2 Installazione di WiringPi

Per scaricare questa libreria, indispensabile ai fini di questo progetto, avremo bisogno di digitare alcuni comandi da terminale, eccoli nel dettaglio:

```
$ sudo apt-get install wiringpi
$ gpio -v //Se compare qualcosa l'installazione è andata a buon fine!
```

6.1.3 Installazione di SendMail

Necessitiamo di installare Sendmail per poter inviare le email da terminale nella funzione UltrasonicDetection(), è una operazione molto semplice infatti basteranno pochissimi passaggi.

```
$ sudo apt-get install sendmail
```

Successivamente andremo a configurare il file contenente gli host eseguendo questi comandi:

```
$ hostname //Per trovare l'hostname
$ sudo nano /etc/hosts //Per modificare il file hosts
```

A questo punto dovremo cercare la riga che inizia con 127.0.0.1 ed aggiungere a questa stringa localhost "hostname" precedentemente trovato.

Adesso dovremo lanciare il file di configurazione di sendmail ed accettare tutto digitando "Y" dopo aver eseguito il seguente comando:

```
$ sudo sendmailconfig
```

A questo punto avremo terminato e di conseguenza potremo inviare email a chi vogliamo tramite terminale semplicemente digitando:

```
$ sendmail -v sheldon@example.com //Inserire la mail del destinatario
$ From: leonard@example.com //Inserire la propria mail
$ Subject: Divertiamoci con le bandiere //Inserire l'oggetto della mail
$ "My personal message" //Inserire il messaggio e premere invio
```

Conclusione

Questo progetto nasce dall'idea di sviluppare un sistema per controllare da remoto un led, un po' per gioco, un po' dalla passione maturata nel corso degli anni per questo ramo dell'informatica, un po' come sfida personale. Come si è potuto notare quell'idea si è concretizzata, pezzo per pezzo, aggiungendo sempre più funzionalità e con la possibilità di aggiungerne sempre di nuove. Si sente sempre più spesso parlare di casa domotica, di IoT, di accessori smart e così via, fa spesso sorridere quando si pensa che con la tecnologia attuale sia possibile controllare da remoto praticamente qualsiasi dispositivo, da una lampadina ad un frigorifero, il tutto nasce proprio da questo pensiero, sfruttando la Raspberry Pi, che ha un costo davvero irrisorio e comprando qualche banale modulo programmabile è possibile fare letteralmente di tutto. Questo progetto presenta come base solida l'architettura Server-Client e sfrutta questo paradigma per emulare il comportamento di una centrale IoT, comportandosi come un hub centrale, raggiungibile da diversi device contemporaneamente. Il nucleo di questo sviluppo risiede nel fatto che tutti sono dotati ormai di una connessione Wi-Fi e la maggior parte di noi possiede un vecchio pc inutilizzato, bene, con queste sole due cose è possibile iniziare a configurare una sorta di casa domotica, certo, molto rudimentale, trattandosi di sviluppi amatoriali e senza nessuna cura per il design, ma funzionale. I possibili sviluppi futuri di questa applicazione sono tantissimi, come l'aggiunta di una videocamera per avere un sistema di sorveglianza più solido, l'aggiunta di uno schermo LCD per ottenere il rilevamento della temperatura direttamente a schermo, l'installazione di un modulo buzzer che suona quando vengono rilevati dei movimenti, un servo motore per cambiare l'angolo di rilevazione della distanza e tanti, tanti altri.