Progetto Finale Server FTP

Sebastiano Marino 559083 - Basilio Jan Sansiveri 556419

Indice

1 Problem Statement

Il presente progetto si propone di realizzare un server FTP concorrente in linguaggio C che gestisca i seguenti comandi:

- **CWD** (Change Working Directory) Cambia la directory di lavoro corrente
- **LIST** (Directory Listing) Elenca i file e le directory presenti
- **RETR** (Retrieve File) Scarica un file dal server
- **STOR** (Store File) Carica un file sul server
- **QUIT** (Session Termination) Termina la sessione FTP

Per lo sviluppo dell'intero applicativo è stato fondamentale consultare l'RFC 959, documento ufficiale che definisce il File Transer Protocol (FTP) disponibile al seguente indirizzo:

https://datatracker.ietf.org/doc/html/rfc959

Questo documento ha lo scopo di standardizzare il trasferimento di file tra host su reti TCP/IP, inoltre, (anche se non richiesto) abbiamo deciso di implementare una procedura di login per l'accesso da parte dell'utente, al fine di garantire una maggiore sicurezza e una migliore conformità alle specifiche definite nel documento RFC.

1.1 Codici di Risposta

L'FTP, utilizza un sistema di codici di risposta a tre cifre per comunicare lo stato delle operazioni tra client e server, inoltre, questi codici sono standardizzati e permettono al client di sapere se un comando è stato eseguito correttamente o se c'è stato un errore. Di seguito riportiamo alcuni codici di risposta utilizzati all'interno del programma:

- **220** (Service Ready) Il server FTP è pronto a ricevere comandi.
- **230** (User Logged In) Login effettuato correttamente, l'utente è autenticato.
- **530** (Not Logged In) Login fallito, nome utente o password errati.
- **150** (File Status Okay) Pronto a trasferire i dati.
- **226** (Closing Data Connection) Trasferimento dati completato con successo.
- **500** (Syntax Error) Comando non riconosciuto o sintassi non valida.
- **501** (Syntax Error in Parameters) Sintassi errata nei parametri o negli argomenti del comando.
- **502** (Command Not Implemented) Il comando non è supportato dal server.
- **550** (File Unavailable) File o directory non disponibile, può indicare "file non trovato" o "permesso negato".
- **221** (Service Closing Control Connection) Connessione chiusa.

2 Stato Dell'Arte

Il protocollo FTP si basa sull'architettura **client-server**, un modello di comunicazione in cui un client invia richieste, e il server attende e risponde a tali richieste. In questo modello, il server resta in ascolto su una porta di rete predefinita (nel caso di FTP, la porta 21) e stabilisce una connessione TCP ogni volta che un client desidera interagire.

La comunicazione tra client e server avviene tramite l'utilizzo di **socket**, un'interfaccia di programmazione fornita dal sistema operativo che permette ai processi di scambiarsi dati, il server crea un socket "passivo" in ascolto su una determinata porta, mentre il client, crea un socket "attivo" e cerca di connettersi a quello del server e una volta stabilita la connessione, entrambi i socket permettono di leggere e scrivere dati in maniera bidirezionale.

Per la realizzazione del server FTP, nel nostro progetto abbiamo fatto uso del protocollo **TCP** (Transmission Control Protocol), il quale garantisce l'affidabilità della trasmissione, assicurando che non ci siano perdite di pacchetti durante la comunicazione ed evitando che arrivino in disordine. Inoltre il server non si limita a rispondere alle richieste dei client, è lui a coordinare le operazioni principali, ad elaborare le istruzioni e a garantire che ogni comando venga eseguito correttamente, dunque i client hanno il compito di inviare input e richieste, ma il punto di forza del sistema risiede nel server, che ne rappresenta il fulcro.

Si è poi pensato a come rendere il nostro server concorrente, a tal proposito abbiamo deciso di utilizzare i **thread** (flussi di esecuzione interni al processo), che permettono di gestire più client contemporaneamente. In questo caso ogni volta che un client si connette, il server crea un nuovo thread dedicato esclusivamente a quella connessione, così le operazioni di trasferimento file avvengono in parallelo e in modo indipendente.
Ricordiamo inoltre che la **concorrenza** si riferisce all'interlacciamento delle istruzioni di più processi o thread, che condividono le risorse del sistema, in pratica, il sistema operativo alterna l'esecuzione di questi processi in modo da farli sembrare "quasi simultanei", anche se vengono eseguiti uno alla volta su una singola CPU.

3 Metodologia e Implementazione

L'implementazione del server FTP è iniziata con la definizione e la scrittura delle principali funzioni necessarie al suo funzionamento, all'interno del file **funzioni_server.c**.

Per una migliore organizzazione e modularità del codice, è stato creato un file header dedicato, **server.h**, contenente le dichiarazioni delle funzioni e delle costanti utilizzate.

```c
#define PORT 21 //Porta di ascolto server FTP

int inizializzazione_server(); //funzione principale server creazione socket ecc

void attesa_connessioni(int server_fd);//funzione che accetta le varie connessioni in
    entrata

void *gestione_client(void *arg); //funzione eseguita da un thread

int autentica_utente(int client_fd); //funzione di login
```

Successivamente, è stato realizzato il file **server.c**, che rappresenta il punto di ingresso del programma. In questo file, ci siamo limitati a richiamare le funzioni principali definite precedentemente, come l'inizializzazione del socket e la gestione delle connessioni concorrenti.

```c
int main(){

    int server_fd = inizializzazione_server();
    printf("server␣FTP␣in␣ascolto␣sulla␣porta␣:␣%d\n", PORT);

    attesa_connessioni(server_fd);

    return 0;
}
```

Abbiamo poi deciso di dichiarare all'interno di un file header di nome **comandi.h** mediante la definizione di una macro "**FTP_ROOT**" la nostra directory virtuale, creata manualmente all'interno della directory del server. Grazie a quest'ultima sarà possibile effettuare le varie operazioni già citate.

In seguito abbiamo creato una struttura di nome Sessione, composta da:

- **client_fd**: file descriptor del socket associato al client (essenziale per leggere o scrivere sul socket).
- **directory_corrente**: una stringa che tiene traccia della cartella corrente del client.

```c
#define FTP_ROOT "./ftp_root"

typedef struct {
    int client_fd;
    char directory_corrente[PATH_MAX];
}Sessione;
```

Dunque, ogni thread lato server, che gestisce un client, avrà una propria Sessione distinta, così da poter tenere separati i dati inerenti a ciascun client.

3.1 Inizializzazione_server()

Come si può notare il primo passo implementativo è stato quello di creare il socket di riferimento lato server e configurare l'indirizzo di rete.
Abbiamo perciò richiamato la funzione inizializzazione_server che prevede come valore di ritorno il relativo file descriptor. Di seguito riportiamo tutta la logica implementativa relativa a questa funzione mediante il seguente script di codice:

```
int inizializzazione_server(){
    int server_fd;
    struct sockaddr_in indirizzo; //struttura che contiene info di rete (ipv4,porta...)

    //Funzione socket per la creazione del socket TCP
    if((server_fd = socket(AF_INET,SOCK_STREAM,0)) == -1){
        perror("Errore creazione socket");
        return -1;
    }

    //configurazione indirizzo
    indirizzo.sin_family = AF_INET;
    indirizzo.sin_addr.s_addr = INADDR_ANY;//accetta connessioni da qualsiasi IP
    indirizzo.sin_port = htons(PORT);

    //Funzione per associare indirizzo e porta alla socket
    if(bind(server_fd,(struct sockaddr *)&indirizzo,sizeof(indirizzo)) < 0){
        perror("Errore bind");
        close(server_fd);
        return -1;
    }

    //Funzione per mettere in ascolto la nostra socket su piu' connessioni
    if(listen(server_fd,MAX_CONNESSIONI) < 0){
        perror("Errore listen");
        close(server_fd);
        return -1;
    }

    return server_fd;
}
```

All'interno delle funzione abbiamo dichiarato la variabile intera relativa al socket e la struttura che identifica l'indirizzo del server.

Dopodichè viene creato il socket mediante la funzione **socket()** e configurato l'indirizzo di rete.

Attraverso la funzione **bind()** andiamo poi a legare il socket a un determinato indirizzo e numero di porta.

Infine, utilizziamo **listen()** per mettere in ascolto il nostro socket, permettendo a un massimo di 3 connessioni di rimanere in attesa di essere accettate.

3.2 Attesa_connessioni()

La prossima funzione ad entrare in gioco è proprio attesa_connessioni che come parametro prende proprio il socket appena creato e configurato.

```
void attesa_connessioni(int server_fd){
    //loop infinito
    while(1){
        struct sockaddr_in indirizzo_client;
        socklen_t lunghezza = sizeof(indirizzo_client);
        //funzione accept che ritorna la socket client per garantire la comunicazione
        int client_fd = accept(server_fd,(struct sockaddr *)&indirizzo_client,&lunghezza);

        if(client_fd < 0){
            perror("errore accept");
            continue;
        }

        printf("Nuova connessione da %s : %d\n",inet_ntoa(indirizzo_client.sin_addr),ntohs(
            indirizzo_client.sin_port));

        //Allocazione dinamica di memoria per la sessione tra client-server
        Sessione *sessione = malloc(sizeof(Sessione));
        if(!sessione){
            fprintf(stderr, "Errore malloc\n");
            close(client_fd);
            continue;
        }
        //impostazione campi sessione...
        sessione->client_fd = client_fd;
        strncpy(sessione->directory_corrente,FTP_ROOT,PATH_MAX); //Impostiamo la directory
            corrente a ftp_root

        //Creazione di un nuovo thread per gestire il client
        pthread_t thread_id;
        //Ogni thread eseguira' la funzione gestione_client per eseguire i vari comandi
        if(pthread_create(&thread_id,NULL,gestione_client,(void *)sessione) != 0){
            perror("Errore creazione thread");
            close(client_fd);
            free(sessione);
            continue;
        }

        pthread_detach(thread_id); //libera le risorse inerenti al thread dopo la sua
            esecuzione.

    }
}
```

Per mantenere sempre il server attivo pronto ad accettare nuove connessioni abbiamo definito un loop while infinito all'interno del quale viene creata una variabile indirizzo di tipo sockaddr_in che verrà riempita con le informazioni (IP e porta) del client che si connette.

Viene poi chiamata la funzione **accept()** per accettare una connessione in arrivo e ritornando la socket relativa al client generando un vero e proprio canale di comunicazione separato tra client e server. Ricordiamo che tale funzione è bloccante e dunque il programma tenderà a interrompersi in attesa di una connessione da parte di un client.

Successivamente allochiamo dinamicamente memoria per una struttura "Sessione" che servirà a mantenere informazioni sulla sessione con questo client (come il socket e la directory corrente), se malloc fallisce, chiudiamo subito il socket e ricominciamo il ciclo per accettare nuove connessioni, in seguito inizializziamo i campi relativi alla sessione assegnando al socket il file descriptor appena ottenuto mentre la directory corrente viene

impostata come ftp_root (la nostra directory virtuale).

A questo punto, creiamo un nuovo thread dedicato alla gestione della comunicazione con il client appena connesso, facciamo dunque utilizzo della funzione **pthread_create()** per generarlo e passiamo come parametri l'identificatore del thread (thread_id), la relativa funzione da eseguire **gestione_client** ed il puntatore alla struttura sessione, liberiamo poi le risorse relative al thread una volta terminato mediante la funzione **pthread_detach()**.

3.3 Gestione_client()

La seguente funzione, come già accennato, verrà eseguita da ciascun thread creato. Al suo interno include due parti fondamentali, una funzione **autentica_utente()**, che si occupa di verificare le credenziali dell'utente per effettuare l'accesso, e una funzione **gestisci_comando()**, che consente di eseguire i vari comandi richiesti dall'utente.

Andiamo ora a vederla nel dettaglio per comprendere meglio come queste due funzioni collaborano nella gestione della comunicazione tra server e client.

```
void *gestione_client(void *arg){
    //Cast dell'argomento in tipo Sessione
    Sessione *sessione = (Sessione *)arg;
    char buffer[DIM_BUFFER];//buffer che conterra' i comandi
    ssize_t n;//numero di byte ricevuti

    char *benvenuto = "220 Benvenuto nel server FTP\n";
    send(sessione->client_fd,benvenuto,strlen(benvenuto),0);

    if (!autentica_utente(sessione->client_fd)) {
        printf("Autenticazione fallita. Connessione chiusa.\n");
        close(sessione->client_fd);
        free(sessione);
        pthread_exit(NULL);
    } else {
        printf("Utente autenticato correttamente.\n");
    }


    //loop infinito per gestire i comandi del client finche' non si disconnette
    while(1){
        memset(buffer,0,DIM_BUFFER);
        //Popolazione buffer mediante recv()
        n = recv(sessione->client_fd,buffer,DIM_BUFFER -1,0);

        //se la connessione e' stata chiusa (=0) o ci sono stati errori (<0) il client si
            disconnette
        if(n <= 0){
            printf("Disconnessione client : %d", sessione->client_fd);
            break;
        }

        //Pulizia di eventuali terminatori di riga
        buffer[strcspn(buffer, "\n")] = '\0';
        printf("Comando ricevuto  %d: %s\n", sessione->client_fd, buffer);

        //Richiama la funzione per la gestione dei comandi
        gestisci_comando(sessione,buffer);
    }
    close(sessione->client_fd);
    free(sessione);
    pthread_exit(NULL);
}
```

All'interno della funzione, il primo passo è quello di eseguire il cast del parametro ricevuto in ingresso, ovvero il puntatore generico di tipo void, in un puntatore alla struttura Sessione. Questo permette di accedere ai campi come il file descriptor del client (client_fd) e la directory corrente associata alla sessione.

Successivamente viene preparato un buffer per contenere i comandi inviati dal client e subito dopo, viene inviato un messaggio di benvenuto al client mediante la funzione **send()**.

A questo punto chiamiamo la funzione **autentica_utente()**, passando il file descriptor del client per verificare che l'utente sia autorizzato. Se l'autenticazione fallisce, chiudiamo la connessione e liberiamo la memoria usata dalla sessione, se invece va tutto bene, siamo pronti a ricevere comandi dal client.

Per concludere, entriamo in un ciclo infinito finché il client non si disconnette e dentro questo ciclo:

- Puliamo il buffer con **memset()** per assicurarci che non ci siano dati residui
- Riceviamo i dati dal client con **recv()** e li salviamo nel buffer
- Passiamo il comando alla funzione **gestisci_comando()**, che lo analizzerà ed eseguirà

Quando il client si disconnette, chiudiamo il socket con **close()**, liberiamo la memoria con **free()** e facciamo terminare il thread con **pthread_exit()**.

3.4 Gestisci_comando()

Questa funzione è fondamentale nel nostro progetto in quanto contiene al suo interno la logica necessaria per eseguire i relativi comandi inoltrati dai client, essendo più complessa abbiamo deciso di inserirla all'interno di un file apposito di nome **comandi.c**.

La funzione di riferimento non restituisce alcun valore, si occupa di ricevere come parametri un puntatore alla struttura Sessione così da poter accedere ai campi già disccussi e una stringa di nome comando che identifica il comando inoltrato dal client.

Come primo passo implementativo abbiamo generato un buffer denominato copia e utilizzato la funzione **strncpy()** per copiare dal comando originale tutti i caratteri fino a un massimo di DIM_BUFFER - 1. In questo modo lasciamo sempre spazio per il carattere di terminazione, che aggiungiamo manualmente per essere certi che la nuova stringa sia correttamente terminata.

Abbiamo scelto di lavorare su questo buffer in quanto, nel passo successivo, dobbiamo utilizzare la funzione **strtok()** per suddividere il comando ricevuto in più parti (token). Questo ci permette di distinguere il comando vero e proprio dall'eventuale argomento (ad esempio, nel comando CWD nome_percorso il comando è CWD e l'argomento è nome_percorso), tale funzione non può aggire sulla stringa originale in quanto essendo una costante non può essere in alcun modo modificata.

```
1  void gestisci_comando(Sessione *sessione, const char *comando){
2      char copia[DIM_BUFFER];
3      //copiamo il comando ricevuto nel buffer
4      strncpy(copia,comando, DIM_BUFFER -1);
5      copia[DIM_BUFFER - 1] = '\0';
6      char *cmd = strtok(copia, " ");
7      char *arg = strtok(NULL,"");
8      //Se il comando inoltrato non e' riconosciuto o non esiste allora verranno stabiliti i
           relativi log
9      if(!cmd){
10         fprintf(stderr, "Errore: nessun comando ricevuto, Input: '%s'\n", comando);
11         char msg[] = "500 Comando non riconosciuto.\n";
12         send(sessione->client_fd, msg,strlen(msg),0);
13         return;
14     }
```

3.5 QUIT

Come primo accertamento, abbiamo verificato se il comando ricevuto fosse QUIT. Nel caso in cui lo fosse, viene inoltrato un messaggio di risposta al client con codice 221, che indica la chiusura della connessione da parte del server.

```
//Se il comando ricevuto e' QUIT allora il client si disconnette
   if(strcmp(cmd, "QUIT") == 0){
       char msg[] = "221 Arrivederci. \n";
       send(sessione->client_fd,msg,strlen(msg),0);
   }
```

3.6 CWD

In questa parte del codice viene gestito il comando CWD, che serve per cambiare directory.
Per prima cosa, controlliamo se non è stato passato alcun argomento, in tal caso, il server risponde al client con un messaggio che spiega come utilizzare correttamente il comando, indicando che bisogna specificare una directory (messaggio con codice 501).

```
   else if(strcmp(cmd,"CWD") == 0){
       if(arg == NULL){
           char msg[] = "501 Utilizzo comando: CWD <directory> \n";
           send(sessione->client_fd, msg, strlen(msg),0);
       }
```

Se invece l'argomento è .., significa che il client vuole risalire alla directory superiore. In questo caso verifichiamo che la directory corrente sia già la directory principale FTP_root, se lo è, non è consentito uscire da questa cartella per ragioni di sicurezza e viene inviato un messaggio di errore al client (codice 550).

```
else if (strcmp(arg, "..") == 0){
            if(strcmp(sessione->directory_corrente, FTP_ROOT) == 0){
                char msg[] = "550 Non puoi uscire dalla directory principale FTP_root";
                send(sessione->client_fd, msg, strlen(msg),0);
```

Se invece non siamo nella directory principale, si risale effettivamente di un livello andando a cercare con la funzione **strrchr()** l'ultima occorrenza del carattere / nel percorso corrente e tagliando da lì in poi. Se, dopo questa operazione, la directory corrente risultasse vuota, la si reimposta a FTP_ROOT. Infine, viene costruito un messaggio che conferma al client l'avvenuto cambio di directory, includendo nel testo il nuovo percorso corrente, e questo messaggio viene inviato tramite la socket associata al client.

```
}else{
        char *p = strrchr(sessione->directory_corrente, '/');//cerca l'ultima occorrenza di
            /
        if(p != NULL)
        *p = '\0';
        if(strcmp(sessione->directory_corrente, "") == 0)
        strncpy(sessione->directory_corrente, FTP_ROOT,PATH_MAX);
        char msg[DIM_BUFFER];
        int len = snprintf(msg, DIM_BUFFER, "250 Directory cambiata a %s\n", sessione->
            directory_corrente);
        if (len < 0) {
        perror("snprintf error");
        }else if (len >= DIM_BUFFER) {
        fprintf(stderr, "Warning: messaggio troncato in msg (DIM_BUFFER=%d)\n", DIM_BUFFER);
        }
```

```
        send(sessione->client_fd,msg,strlen(msg),0);
        }
    }
```

In questa parte del codice, viene gestito il caso in cui l'utente voglia cambiare directory specificando un nome diverso da **".."** . Si tratta quindi del ramo else che entra in azione quando l'argomento passato non è nullo e non indica di tornare indietro.

```
        //Se l'argomento passato e' un percorso assoluto passiamo al carattere successivo
            allo /, altrimenti rimarra' invariato
        else{
            char pulisci_argomento[PATH_MAX];
            if(arg[0] == '/'){
                snprintf(pulisci_argomento, PATH_MAX, "%s", arg +1);
            } else{
                snprintf(pulisci_argomento,PATH_MAX,"%s",arg);
            }
            char nuovoPercorso[PATH_MAX];
            int len = snprintf(nuovoPercorso, PATH_MAX, "%s/%s", sessione->
                directory_corrente, pulisci_argomento);
            if (len < 0) {
                perror("snprintf error");
            } else if (len >= PATH_MAX) {
                fprintf(stderr, "Warning: path troncato in nuovoPercorso (PATH_MAX=%d)\n",
                    PATH_MAX);
            }
            //se stat restituisce 0 allora il percorso esiste, la MACRO S_ISDIR controlla se
                 il percorso e' una directory
            struct stat info;
            if(stat(nuovoPercorso,&info) == 0 && S_ISDIR(info.st_mode)){
                strncpy(sessione->directory_corrente,nuovoPercorso,PATH_MAX);
                char msg[DIM_BUFFER];
                int len = snprintf(msg, DIM_BUFFER, "250 Directory cambiata a : %s\n",
                    sessione->directory_corrente);
                if (len < 0) {
                    perror("snprintf error");
                } else if (len >= DIM_BUFFER) {
                    fprintf(stderr, "Warning: messaggio troncato in msg (DIM_BUFFER=%d)\n",
                        DIM_BUFFER);
                }
                send(sessione->client_fd,msg,strlen(msg),0);
            }
            //altrimenti se il percorso non esiste
            else{
               fprintf(stderr, "Errore  directory non trovata , Input : %s",comando);
               char msg[] = "550 Directory non trovata\n";
               send(sessione->client_fd,msg,strlen(msg),0);
            }
        }
    }
```

La prima cosa che viene fatta è dichiarare un array chiamato pulisci_argomento, di dimensione PATH_MAX, che servirà a contenere una versione "ripulita" dell'argomento. Se l'argomento inizia con il carattere /, si copia in pulisci_argomento la stringa a partire dal carattere successivo, per evitare problemi nella costruzione del percorso. Se invece non inizia con /, l'argomento viene copiato così com'è in pulisci_argomento. Questo permette all'utente di specificare la directory sia con che senza lo slash iniziale.

A questo punto, viene dichiarato un array chiamato nuovoPercorso, che conterrà il percorso completo.
Per costruirlo, si usa la funzione snprintf, che combina la directory corrente e l'argomento ripulito (pulisci_argomento), separandoli con uno /. Dopo questa operazione, vengono effettuati dei controlli per verificare che tutto sia corretto.

Una volta costruito il nuovo percorso, il codice deve verificare che esso esista realmente sul filesystem e che corrisponda a una directory. Il filesystem, in questo contesto, è la struttura logica utilizzata dal sistema operativo per organizzare e gestire file e cartelle.
Per effettuare questa verifica, viene dichiarata una variabile di tipo struct stat chiamata info, e si utilizza la funzione **stat()** passando come argomento il percorso appena costruito.

- Se stat() restituisce 0 tutto sarà avvenuto con successo e abbiamo la conferma che il percorso è una directory, e l'operazione può procedere.
- Se invece stat() fallisce o il percorso non è una directory, si conclude che la directory specificata non esiste o non è valida.

Nel caso in cui la directory esista, viene aggiornata la directory corrente della sessione copiandovi dentro il nuovo percorso. Poi, viene preparato un messaggio di conferma (codice 250) che dice al client che la directory è stata cambiata con successo, anche qui si controlla se snprintf ha dato errori o ha troncato la stringa, e si invia il messaggio al client tramite la funzione send.

Se invece la directory non esiste, viene scritto un messaggio di errore, specificando l'input ricevuto, e poi si invia al client un (messaggio con codice 550) che indica che la directory richiesta non è stata trovata.

3.7 LIST

```
else if(strcmp(cmd,"LIST") == 0){
       DIR *dir = opendir(sessione->directory_corrente); //Apre la directory grazie alla
           funzione opendir() dalla libreria dirent.h
       if(dir == NULL){//se non e' stata trovata alcuna directory
           fprintf(stderr, "Errore: impossibile leggere la directory corrente. Input: '%s'\
               n", comando);
           char msg[] = "550 Impossibile leggere la directory. \n";
           send(sessione->client_fd,msg,strlen(msg),0);
           return;
       }
       char msg_inizio[] = "150 Inizio elenco-- \n";
       send(sessione->client_fd,msg_inizio,strlen(msg_inizio),0);
       struct dirent *elemento;
       char lista[DIM_BUFFER];
       while((elemento = readdir(dir)) != NULL){//finche' vengono lette directory
           if(strcmp(elemento->d_name, ".") == 0 ||strcmp(elemento->d_name,"..") == 0 )//Se
               la directory e' figlia (.) oppure padre (..) continua a leggere
               continue;
           snprintf(lista, DIM_BUFFER, "%s\n", elemento->d_name);
           send(sessione->client_fd, lista, strlen(lista), 0);
       }
       closedir(dir);//chiudiamo la directory
       char msg_fine[] = "226 Fine elenco.\r\n";
       send(sessione->client_fd, msg_fine, strlen(msg_fine), 0);
   }
```

Nel caso in cui il comando ricevuto è LIST sarà possibile elencare i file presenti all'interno della directory corrente.

Per fare ciò viene confrontata inizialmente la stringa ricevuta con "LIST" usando come sempre la funzionre **strcmp()**, se il confronto avviene con successo la directory corrente viene aperta attraverso la funzione **opendir()**.Se la directory non viene trovata, cioè se **opendir()** restituisce **NULL**, viene stampato un messaggio di errore, specificando che non è stato possibile leggere la directory corrente. In questo caso il server invia al client un messaggio FTP (codice 550) che indica l'impossibilità di accedere alla directory, dopodichè che termina l'esecuzione del comando con return.

Se invece la directory viene aperta correttamente, il server invia al client un messaggio (con codice 150) per segnalare l'inizio dell'elenco dei file. Viene poi dichiarato un puntatore alla struct dirent, chiamato elemento (che si occuperà di ricevere i vari elementi presenti all'interno della directory), e un buffer chiamato lista.

Succesivamente il server entra in un ciclo while che continua finché **readdir()** restituisce elementi validi dalla directory. All'interno del ciclo, vengono ignorati gli elementi "." e ".." che rappresentano rispettivamente la directory corrente e quella superiore, tramite un controllo con strcmp.

Per ogni altro elemento valido, il nome del file viene scritto nel buffer lista usando snprintf, seguito da un carattere di newline, e poi inviato al client con la funzione send.

Una volta terminata la lettura di tutti gli elementi, la directory viene chiusa con closedir, e il server invia al client un messaggio (con codice 226) che indica la fine dell'elenco.

3.8 RETR

```
else if(strcmp(cmd,"RETR") == 0){
       if(arg == NULL){//se non viene fornito alcun argomento si spiega l'uso del comando
           char msg[] = "501 Uso: RETR <nome_file>\n";
           send(sessione->client_fd, msg, strlen(msg), 0);
       }else{
               char percorso[PATH_MAX];
               int len = snprintf(percorso, PATH_MAX, "%s/%s", sessione->directory_corrente
                   , arg); //inserisce il path formato dalla dir corrente e l'argomento
                   passato
               if (len < 0) {
                   perror("snprintf error");
               }else if(len >= PATH_MAX){
                   fprintf(stderr, "Warning: path troncato in percorso (PATH_MAX=%d)\n",
                       PATH_MAX);
               }
               FILE *file = fopen(percorso,"rb");//apriamo il file in modalita' "rb"
               if(file ==NULL){
                   char msg[] = "550 File non trovato.\n";
                   send(sessione->client_fd, msg, strlen(msg), 0);
                   return;
               }
               char msg_inizio[] = "150 Inizio download...\n";
               send(sessione->client_fd, msg_inizio, strlen(msg_inizio), 0);
               char dati[DIM_BUFFER];
               size_t file_letti;
               while ((file_letti = fread(dati, 1, DIM_BUFFER, file)) > 0) {//con fread
                   leggiamo i file e salviamo il contenuto in dati
                   send(sessione->client_fd, dati, file_letti, 0);
               }
               fclose(file);
               char msg_fine[] = "\n226 Download completato.\n";
               send(sessione->client_fd, msg_fine, strlen(msg_fine), 0);
           }
   }
```

Per la gestione del comando RETR, il server confronta la stringa ricevuta con "RETR" usando strcmp, se il confronto ha successo, verifica se è stato fornito un argomento, cioè il nome del file da scaricare.

Se l'argomento è assente, il server invia al client un messaggio FTP (con codice 501) che indica l'uso corretto del comando. Se invece l'argomento è presente, il server dichiara un array chiamato percorso, che servirà a contenere il percorso completo del file.

Per costruire il percorso viene fatto uso della funzione snprintf cosi da unire la directory corrente della sessione e il nome del file passato come argomento, se restituisce un valore negativo, viene stampato un messaggio di errore, mentre se il valore restituito è maggiore o uguale a PATH_MAX, significa che il percorso è stato troncato e viene stampato un avviso con fprintf.

Una volta costruito il percorso, il server tenta di aprire il file in modalità binaria con fopen usando il flag "rb", se il file non viene trovato o non è accessibile, il server invia al client un messaggio FTP (con codice 550) che indica che il file non è stato trovato, e termina l'esecuzione del comando con return. Se invece viene aperto correttamente, il server invia al client un messaggio (con codice 150) per segnalare l'inizio del download.

Viene dichiarato un buffer chiamato dati e una variabile di nome file_letti che conterrà il numero di byte letti dal file. Successivamente il server entra in un ciclo while che continua finché la funzione fread riesce a leggere blocchi di dati dal file ed ogni blocco letto viene inviato al client tramite la funzione send.

Quando la lettura del file è completata, il server chiude il file con fclose e invia al client un messaggio (con codice 226) che conferma il completamento del download.

3.9 STOR

```
else if(strcmp(cmd,"STOR") == 0){
        if (arg == NULL) {//se non vi e' alcun file viene specificato il caso d'uso
            char msg[] = "501 Uso: STOR <nome_file>\n";
            send(sessione->client_fd, msg, strlen(msg), 0);
        } else {
                char percorso[PATH_MAX];//si identifica il percorso
                int len = snprintf(percorso, PATH_MAX, "%s/%s", sessione->directory_corrente
                    , arg);//inserisce il path formato dalla dir e arg
                if(len < 0){
                    perror("snprintf error");
                }else if(len >= PATH_MAX){
                    fprintf(stderr, "Warning: path troncato in percorso (PATH_MAX=%d)\n",
                        PATH_MAX);}
                FILE *file = fopen(percorso,"wb");//apriamo il file in modalita' wb
                if(file == NULL){
                        char msg[] = "550 Impossibile creare il file.\n";
                        send(sessione->client_fd, msg, strlen(msg), 0);
                        return;
                }
                char msg_inizio[] = "150 Inizio caricamento, invia dati e termina con <EOF>
                    su una linea.\n";
                send(sessione->client_fd, msg_inizio, strlen(msg_inizio), 0);
                char buffer_dati[DIM_BUFFER];//buffer per ricevere i dati dal client
                int lunghezza;
                while ((lunghezza = recv(sessione->client_fd, buffer_dati, DIM_BUFFER - 1,
                    0)) > 0) { //finche' ci sono dati da leggere
                buffer_dati[lunghezza] = '\0';
                if (strncmp(buffer_dati, "<EOF>", 5) == 0)
                    break;
                fwrite(buffer_dati, 1, lunghezza, file);
            }
            fclose(file);
            char msg_fine[] = "226 caricamento completato.\n";
            send(sessione->client_fd, msg_fine, strlen(msg_fine), 0);

        }
    }else{
            char msg[] = "502 Comando non implementato.\n";
            send(sessione->client_fd, msg, strlen(msg), 0);
    }
}
```

Il comando STOR, serve al client per caricare un file sul server. Inizialmente controlliamo se il comando ricevuto è proprio "STOR". Se dovesse essere così, si verifica se il client ha fornito anche il nome del file da salvare. Se il nome manca, il server risponde con un messaggio di errore che spiega come utilizzare il comando, in quanto senza nome non può sapere dove salvare i dati.

Se invece il nome del file è stato fornito, viene definito un array di nome percorso per costruire il percorso completo dove salvare il file. Per farlo, come in altre occasioni viene utilizzata la funzione snprintf, la quale concatena la directory corrente con il nome del file, separandoli con uno slash.

Questo percorso viene poi salvato nel buffer prima definito, se la lunghezza supera PATH_MAX, il server stampa un avviso perché il percorso potrebbe essere stato troncato.

Una volta ottenuto il percorso, il server prova ad aprire il file in modalità scrittura binaria ("wb") e se non riesce ad aprirlo, magari per problemi di permessi o perché la directory non esiste, invia al client un messaggio di errore: "550 Impossibile creare il file."

Altrimenti se il file viene aperto correttamente, il server invia al client un messaggio di inizio caricamento dati con terminazione mediante la stringa EOF così da poter terminare il caricamento.

A questo punto, è necessario caricare i dati, dunque il server entra in un ciclo dove riceve i dati dal client, un pezzo alla volta, usando un buffer chiamato buffer_dati e ogni volta che riceve dati, controlla se il contenuto ricevuto presenta la stringa EOF, che indica la fine del trasferimento. Se la trova, esce dal ciclo. Altrimenti, scrive i dati ricevuti nel file usando fwrite.

Quando il client ha finito di inviare i dati e il server ha scritto tutto nel file, chiude il file con fclose e invia un messaggio finale al client: "226 caricamento completato." Questo conferma che il file è stato ricevuto e salvato correttamente.

Infine,(essendo STOR l'ultimo comando implementato all'interno della funzione) se il comando ricevuto non è STOR né uno degli altri previsti, il server risponde con: "502 Comando non implementato".

3.10 Autentica_Utente

Come già anticipato, abbiamo deciso di inserire questa funzione per garantire una maggiore sicurezza, implementando un login di accesso con credenziali (username e password) dedicate al nostro server FTP. Questa scelta è stata fatta di nostra iniziativa, anche se non era espressamente richiesta, con l'obiettivo di seguire al meglio le indicazioni del documento RFC 959 proprio per questo abbiamo deciso di esporla per ultima.

```
int autentica_utente(int client_fd) {
    char username[64] = {0};
    char password[64] = {0};
//Ricezione di username e password dal client
    recv(client_fd, username, sizeof(username), 0);
    username[strcspn(username, "\r\n")] = '\0';

    recv(client_fd, password, sizeof(password), 0);
    password[strcspn(password, "\r\n")] = '\0';
//Apertura file in lettura contenente le coppie username:password
    FILE *fp = fopen("Autentica.txt", "r");
    if (!fp) {
        perror("Errore apertura file utenti");
        return 0;
    }
//lettura del file riga per riga e confronto di username e password con strcmp
    char riga[128];
    while (fgets(riga, sizeof(riga), fp)) {
        char file_user[64], file_pass[64];
        if (sscanf(riga, "%[^:]:%s", file_user, file_pass) == 2) {
```

```
            if (strcmp(file_user, username) == 0 && strcmp(file_pass, password) == 0) {
                fclose(fp);
                send(client_fd, "230 Login riuscito.\n", 21, 0);
                return 1;
            }
        }
    }
//Se username e password non corrispondono login fallito
    fclose(fp);
    send(client_fd, "530 Login fallito.\n", 19, 0);
    return 0;
}
```

Come si può notare, questa funzione inizia ricevendo dal client le credenziali inserite a terminale (username e password). Successivamente apre in lettura un file di testo chiamato Autentica.txt, che contiene al suo interno diverse coppie di username e password separate da due punti.

Il file viene quindi letto riga per riga e i dati ricevuti dal client vengono confrontati con quelli presenti nel file così da poter autorizzare l'accesso nel caso in cui corrispondano o negarlo.

3.11 MakeFile

Per semplificare il processo di compilazione del progetto e garantire una gestione più ordinata dei file sorgente e degli eseguibili, abbiamo deciso di utilizzare un Makefile.

```
    CC = gcc
CFLAGS = -Wall -Wextra -pthread

CLIENT_DIR = client
SERVER_DIR = server

CLIENT_SRCS = $(CLIENT_DIR)/client.c $(CLIENT_DIR)/funzioni_client.c
SERVER_SRCS = $(SERVER_DIR)/server.c $(SERVER_DIR)/funzioni_server.c $(SERVER_DIR)/comandi.c

CLIENT_EXEC = $(CLIENT_DIR)/client.out
SERVER_EXEC = $(SERVER_DIR)/server.out

all: $(CLIENT_EXEC) $(SERVER_EXEC)

$(CLIENT_EXEC):
    $(CC) $(CFLAGS) -o $@ $(CLIENT_SRCS)

$(SERVER_EXEC):
    $(CC) $(CFLAGS) -o $@ $(SERVER_SRCS)

clean:
    rm -f $(CLIENT_EXEC) $(SERVER_EXEC)
```

Questo Makefile definisce in modo strutturato:

- Il compilatore da utilizzare (gcc);
- Le opzioni di compilazione (-Wall -Wextra -pthread) che includono la visualizzazione di tutti gli avvisi e il supporto al multithreading;
- I percorsi e i file sorgente specifici sia per il client che per il server;
- I nomi dei file eseguibili da generare.

La regola all, viene lanciata quando andiamo ad inviare il comando make. Questo richiama le regole per la compilazione sia del client che del server.

Le due regole:

```
$(CLIENT_EXEC):
    $(CC) $(CFLAGS) -o $@ $(CLIENT_SRCS)

$(SERVER_EXEC):
    $(CC) $(CFLAGS) -o $@ $(SERVER_SRCS)
```

servono per compilare rispettivamente il client e il server, utilizzando i file sorgente definiti in precedenza.

Infine,clean ci è utile per eliminare i file eseguibili generati durante la compilazione, permettendo di pulire il progetto eseguendo semplicemente make clean.

4 Risultati

Nel nostro progetto abbiamo scelto di implementare il server FTP utilizzando la porta 21, che è la porta standard riservata per il protocollo FTP secondo il documento RFC 959.

È importante sottolineare che, su sistemi Unix/Linux, tutte le porte con numero inferiore a 1024 sono considerate come privilegiate e dunque non direttamente accessibili da normali processi.

Per questo motivo, per avviare correttamente il nostro server sulla porta 21, è stato necessario eseguire l'eseguibile con i permessi di amministratore.

```
seby@seby-Inspiron-15-3525:~/Documenti/ProgettoReti/server$ sudo ./server.out
[sudo] password di seby:
server FTP in ascolto sulla porta : 21
```

4.1 Accesso Utente

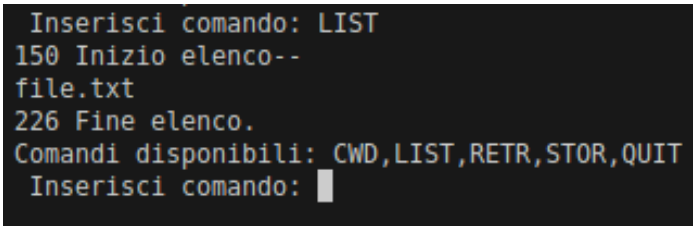
Di seguito mostriamo l'accesso effettuato da parte di un client ed i relativi comandi che potranno essere effettuati:

```
seby@seby-Inspiron-15-3525:~/Documenti/ProgettoReti/client$ ./client.out
Connessione al server FTP riuscita
220 Benvenuto nel nostro server FTP
Username: Seby
Password: 1234
Server: 230 Login riuscito.
Comandi disponibili: CWD,LIST,RETR,STOR,QUIT
 Inserisci comando:
```

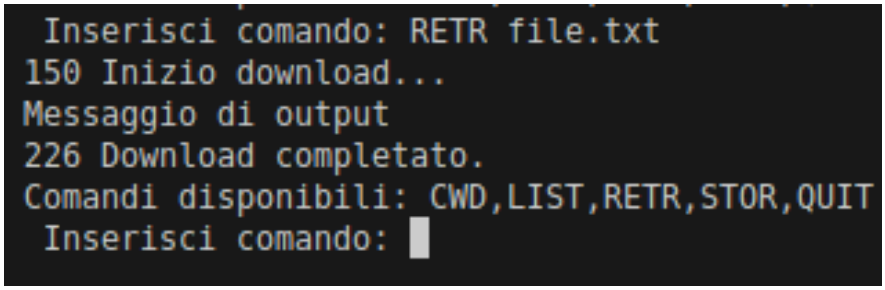
4.2 OUTPUT CWD

```
 Inserisci comando: CWD /Documenti
250 Directory cambiata a : ./ftp_root/Documenti
Comandi disponibili: CWD,LIST,RETR,STOR,QUIT
 Inserisci comando:
```

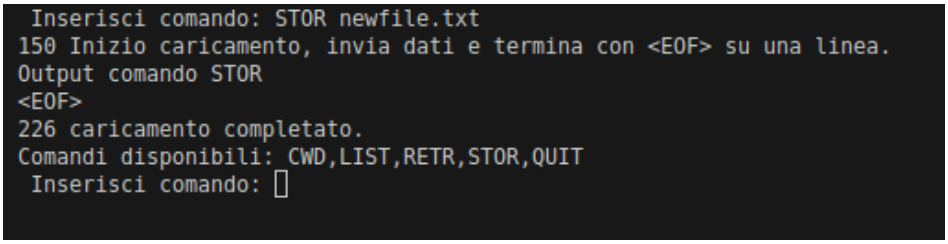
4.3 OUTPUT LIST



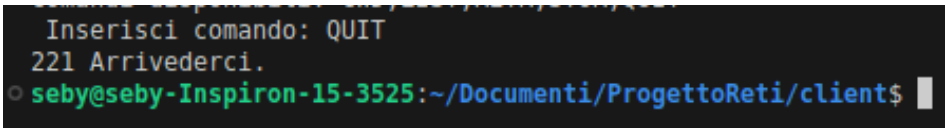
4.4 OUTPUT RETR



4.5 OUTPUT STOR



4.6 OUTPUT QUIT



5 Conclusione e Sviluppi Futuri

Lo sviluppo del nostro server FTP concorrente ci ha permesso di approfondire diversi concetti fondamentali legati alla programmazione di rete e alla gestione della concorrenza.

Durante il progetto, è stato implementato un sistema capace di accettare più connessioni simultanee, consentendo a diversi client di interagire con il server ed eseguire i comandi previsti.

Siamo tuttavia consapevoli che l'applicativo, nella sua versione attuale, può essere ulteriormente migliorato. Di seguito proponiamo alcuni possibili sviluppi futuri che potrebbero renderlo più completo e sicuro:

- **Sistema di autenticazione avanzato**: attualmente, l'autenticazione avviene tramite inserimento diretto di username e password nel terminale, visibili in chiaro. Uno sviluppo importante consisterebbe nell'introdurre un sistema di login più sicuro, che permetta di nascondere la password durante l'inserimento e di salvare le credenziali in modo cifrato
- **Crittografia e sicurezza**: per aumentare il livello di sicurezza nella trasmissione dei dati, si potrebbe integrare un protocollo di crittografia come FTPS, in modo da proteggere le informazioni da accessi non autorizzati.
- **Interfaccia grafica o web**: lo sviluppo di una semplice interfaccia web renderebbe l'applicazione più user-friendly, permettendo anche a utenti meno esperti di interagire con il server in modo più intuitivo, senza dover utilizzare esclusivamente il terminale.

Con gli sviluppi futuri proposti, il progetto potrebbe diventare più completo e utile anche in contesti reali, sarebbe così possibile offrire uno strumento facile da usare, sicuro e applicabile in situazioni concrete.