Presentazione Server FTP

Sebastiano Marino 559083 - Basilio Jan Sansiveri 556419

August 29, 2025

Indice

Problem Statement

Obiettivo della Presentazione

Con questa presentazione andiamo a spiegare a grandi linee come realizzare un **server FTP concorrente** in linguaggio C e cosa è stato necessario utilizzare per implementarlo.

Nota

La rappresentazione completa dei vari script di codice, insieme alla descrizione dettagliata, è contenuta nella relazione scritta allegata alla repository GitHub al seguente indirizzo:
https://github.com/LabReSiD25-Unime/final-559083.git.

Problem Statement

Server Concorrente

Perché implementare un Server Concorrente?
A differenza di un server sequenziale, un server concorrente è progettato per gestire più richieste da parte di client contemporaneamente.
Ciò è reso possibile grazie all'utilizzo di chiamate bloccanti come **accept()** e all'impiego di **thread** (flussi di esecuzione interni a un processo).
Ogni thread gestisce quindi una connessione separata tra client e server, permettendo al server di rispondere simultaneamente a più richieste, migliorando così l'efficienza e la scalabilità del sistema.

Server Concorrente

Siamo tuttavia consapevoli del fatto che, per migliorare il nostro server e garantire una maggiore scalabilità, sarebbe stato preferibile utilizzare tecniche di I/O multiplexing (come **select()** o **epoll()**), che permettono di monitorare più file descriptor contemporaneamente, evitando la creazione di un thread per ogni connessione e riducendo così il carico sul sistema.

Stato Dell'Arte

Cos'è il protocollo FTP?

FTP è un protocollo di comunicazione che consente a un client di connettersi a un server remoto per scaricare o caricare file.
È stato progettato per facilitare lo scambio di file in modo semplice e strutturato come definito dal Documento ufficiale **RFC 959**.

RFC 959

Tale documento si occupa di rappresentare al meglio il protocollo FTP definendone le regole. Al suo interno vengono spiegati in maniera dettagliata i codici di risposta, codici numerici che permettono di informare i client sullo stato delle operazioni o nel caso in cui la connessione venga interrotta (codice 221).

Stato Dell'Arte

Codici di Risposta

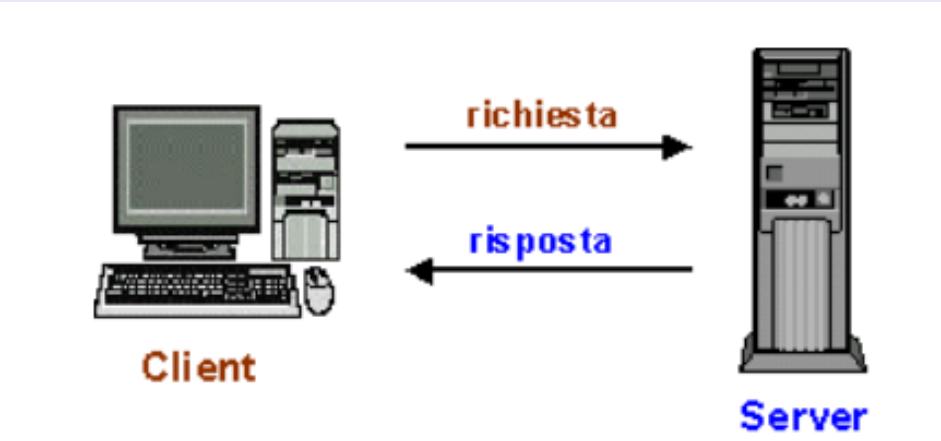
Di seguito riportiamo alcuni dei codici di risposta utilizzati lato Server:

- **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.
- **221** (Service Closing Control Connection) Connessione chiusa.

Architettura Client-Server

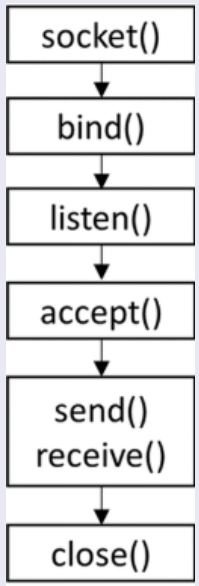
L'architettura Client-Server rappresenta il modello di comunicazione su cui si basa il protocollo FTP. I "protagonisti" principali sono due:

- Client: Si occupa di inoltrare delle richieste di implementazione al Server (inoltrazione comandi).
- Server: Si occupa di ricevere tali richieste, di implementarle e di restituire un output.



Socket

Banalmente un **Socket** rappresenta una interfaccia fornita dal Sistema Operativo che si occupa di effettuare operazioni di lettura e scrittura così da garantire una comunicazione bidirezionale tra Client e Server.



Stato Dell'Arte

TCP

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.

Metodologia e Implementazione

Comandi da Implementare

Il nostro server FTP dovrà essere in grado di gestire correttamente i comandi inoltrati dai client.
Di seguito vengono elencati i principali comandi che sarà necessario implementare:

- **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

Metodologia e Implementazione

ftp_root

Per implementare tali comandi però è stato necessario definire una directory virtuale su cui poi verranno effettuate le operazioni.
Abbiamo quindi deciso di dichiarare all'interno di un file header di nome **comandi.h** una macro "**FTP_ROOT**" che identifica la nostra directory virtuale che è stata creata manualmente all'interno della directory del server.

```
#define FTP_ROOT "./ftp_root" //definisce la directory virtuale
```

Metodologia e Implementazione

Sessione

All'interno dello stesso file siamo andati a definire 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.

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

Programma principale

All'interno del file **server.c** siamo andati a definire il programma principale main il quale si occuperà di richiamare due essenziali funzioni che sono state definite all'interno del file **funzioni_server.c**

```c
int main(){

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

    attesa_connessioni(server_fd);

    return 0;
}
```

Metodologia e Implementazione

Programma Principale

La prima funzione che viene richiamata è inizializzazione_server, la quale si occupa di creare il socket, configurarlo affinché il server entri in modalità di ascolto, e restituire il relativo FD.

Analizziamo dunque la funzione passo dopo passo :

Metodologia e Implementazione

Inizializzazione_server

```c
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;
    }
```

Metodologia e Implementazione

La funzione come si può notare restituisce un tipo di dato intero e non prende parametri in ingresso.

Inizialmente definiamo due variabili: la prima è un intero, chiamato server_fd, che rappresenta il file descriptor del server e verrà inizializzato successivamente tramite la funzione socket(). La seconda è una variabile di tipo sockaddr_in, chiamata indirizzo, che fa riferimento all'indirizzo di rete da configurare lato server.

Poi chiamiamo la funzione socket così da creare effettivamente il socket passando i relativi parametri.

Metodologia e Implementazione

Inizializzazione_server

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

Dopodichè configuriamo i campi relativi alla stuttura sockaddr_in.

Metodologia e Implementazione

Inizializzazione_server

```
    //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 più connessioni
    if(listen(server_fd,MAX_CONNESSIONI) < 0){
        perror("Errore listen");
        close(server_fd);
        return -1;
    }

    return server_fd;
}
```

Per concludere attraverso la funzione **bind()** andiamo 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 e ritorniamo il FD configurato.

Programma Principale

La seconda funzione ad essere richiamata è Attesa_Connessioni(), la quale prende come parametro in ingresso il FD restituito dalla funzione appena descritta.

Questa funzione attende e accetta connessioni in entrata. Conterrà quindi la chiamata bloccante **accept()** e si occuperà di restituire un socket (client_fd) che identificherà un canale di comunicazione privato tra il client appena connesso e il nostro server. Analizziamo anche questa funzione passo dopo passo.

Attesa_Connessioni

```c
//Funzione che prende in input la socket e permette a più client di connettersi
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));
```

Metodologia e Implementazione

Come si può notare, ci troviamo di fronte a una procedura di tipo void. All'interno di essa è presente un ciclo while infinito, che mantiene il server in ascolto delle connessioni in entrata. Abbiamo inoltre definito una variabile chiamata indirizzo_client, che rappresenta l'indirizzo di rete del client che tenterà di connettersi al server.

Successivamente viene chiamata la funzione accept(). A questo punto, il programma si blocca in attesa che un client si connetta. Una volta stabilita la connessione, accept() restituisce un socket che identifica il canale di comunicazione tra client e server

Metodologia e Implementazione

Attesa_Connessioni

```
//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
```

Procediamo allocando dinamicamente memoria alla struttura Sessione e nel caso in cui tutto proceda per il meglio inizializziamo i campi impostando al campo "client_fd" il socket restituito da accept mentre il campo "directory corrente" viene impostato a ftp_root attraverso la funzione strncpy.

Metodologia e Implementazione

Attesa_Connessioni

```
        pthread_t thread_id;
        //Ogni thread eseguirà 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 concludere creiamo un nuovo thread relativo alla comunicazione client-server a cui passiamo come parametro la funzione gestione_client (da eseguire) dentro la quale verranno ricevuti i comandi da parte del client che verranno poi implementati.
Una volta terminata l'esecuzione del thread all'interno della funzione eseguita verrà richiamata la funzione pthread_detach per liberare le risorse inerenti al thread.

Metodologia e Implementazione

Andiamo adesso ad analizzare la funzione eseguita da parte del thread.

Gestione_client

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

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

Metodologia e Implementazione

Inizialmente viene effettuato un cast dell'argomento in un tipo Sessione così da poter accedere ai campi.

Poi viene definito un buffer che conterrà al suo interno i comandi inoltrati da parte dei client e viene definita una variabile n di tipo ssize_t che identifica il numero di byte ricevuti.

Per concludere questa prima parte della funzione inoltriamo un messaggio di benvenuto al client con codice di risposta 220.

Metodologia e Implementazione

Gestione_client

```
//se l'autenticazione fallisce la sessione termina
if (!autentica_utente(sessione->client_fd)) {
    printf("Autenticazione fallita. Chiusura della connessione.\n");
    close(sessione->client_fd);
    free(sessione);
    pthread_exit(NULL);
} else {
    printf("Utente autenticato correttamente.\n");
}
```

Successivamente viene richiamata una funzione di nome autentica_utente che si occupa di verificare le credenziali immesse dall'utente per garantire o meno l'accesso al nostro server.

Abbiamo deciso di inserire questa funzione per uniformarci al meglio con il documento ufficiale e per garantire maggiore sicurezza al nostro applicativo.

Gestione_client

```
    //loop infinito per gestire i comandi del client finchè 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 è 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);
}
```

Metodologia e Implementazione

Per concludere viene definito un loop infinito per ricevere i comandi dal client. Dentro il ciclo richiamiamo la funzione **recv()** per ricevere il comando e inserirlo all'interno del buffer.

puliamo poi il nostro buffer da eventuali caratteri di newline così da avere il comando più "pulito" possibile e richiamiamo la funzione **gestisci_comando** la quale prenderà come parametri di ingresso il puntatore alla struttura sessione e il buffer contenente il comando inoltrato.

Sarà dunque tale funzione a implementare i comandi.

Metodologia e Implementazione

Siamo finalmente giunti alla funzione gestisci_comando, tale funzione rappresenta il fulcro del nostro server e date le sue dimensioni abbiamo voluto inserirla all'interno del file **"comandi.c"**

La logica relativa a questa funzione si basa su ripetuti blocchi if else in cui si ha un confronto tra il comando ricevuto nel buffer e la stringa che identifica il comando. Per fare ciò è stata utilizzata la funzione CMP, nel caso in cui il comando corrisponda allora verrà implementato

Gestisci_comando

```c
void gestisci_comando(Sessione *sessione, const char *comando){
    char copia[DIM_BUFFER];
    //copiamo il comando ricevuto nel buffer
    strncpy(copia,comando, DIM_BUFFER -1);
    copia[DIM_BUFFER - 1] = '\0';

    /*suddividiamo la stringa (cmd) in token in cui il primo token rappresenterà il comando vero e proprio
      mentre il secondo (arg) conterrà l'argomento se inserito es:directory..
    */

    char *cmd = strtok(copia, " ");
    char *arg = strtok(NULL,"");

    //Se il comando inoltrato non è riconosciuto o non esiste allora verranno stabiliti i relativi log
    if(!cmd){
        fprintf(stderr, "Errore: nessun comando ricevuto, Input: '%s'\n", comando);
        char msg[] = "500 Comando non riconosciuto.\n";
        send(sessione->client_fd, msg,strlen(msg),0);
        return;
    }
```

Metodologia e Implementazione

Per questa procedura si è partiti definendo un buffer di nome copia che conterrà il comando ricevuto sul buffer.

Utilizziamo dunque la funzione di copia (strncpy) per copiarlo e andiamo a richiamare la funzione strtok che permette di suddividere il nostro comando in un "token" relativo al comando ed in uno relativo all'argomento in quanto per molte operazioni (come CWD) l'utente sarà obbligato a specificare sia il comando che l'argomento.

Se il comando inoltrato non sarà riconosciuto allora verrà inoltrato un messaggio di errore con codice di risposta 500 al client.

Metodologia e Implementazione

Gestisci_comando

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

Adesso entriamo nella logica vera e propria.
Il primo comando che viene verificato è QUIT. Nel caso in cui cmd coindica con la stringa "QUIT" allora il server inoltra un messaggio di arrivederci al client con codice di risposta 221.

Il client in seguito si disconnetterà dal server.

Gestisci_comando

```
//Altrimenti, se il comando è CWD:
else if(strcmp(cmd,"CWD") == 0){
    //se non vi è argomento allora si inoltra al client la spiegazione d'uso
    if(arg == NULL){
        char msg[] = "501 Utilizzo comando: CWD <directory> \n";
        send(sessione->client_fd, msg, strlen(msg),0);
    }
    //altrimenti se si vuole risalire alla directory superiore utilizziamo il comando ".."
    else if (strcmp(arg, "..") == 0){
        //se la directory in cui ci troviamo è FTP_root non possiamo effettuare la risalita
        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);
            /*altrimenti risaliamo di una directory  eliminando l'ultimo '/' e tutto ciò che segue,
             lasciando solo il percorso della directory superiore.*/
```

Metodologia e Implementazione

Verifichiamo poi il comando CWD e nel caso in cui corrisponda effettuiamo un primo controllo sull'argomento e se l'argomento non viene specificato (dunque "NULL") inviamo una stringa al client contenente la dimostrazione di come utilizzare il comando.

Nel caso in cui però il client abbia specificato come argomento del comando i caratteri ".." vorrà dire che avrà intenzione di risalire da una directory figlia ad una directory padre superiore. In questo caso intanto verifichiamo che la directory corrente non sia ftp_root perchè non è possibile uscire dalla directory principale.

nel caso in cui lo sia allora inoltriamo un messaggio di errore con codice di risposta 550 al client.

Metodologia e Implementazione

Gestisci_comando

```
    }else{
        char *p = strrchr(sessione->directory_corrente, '/'); //cerca l'ultima occorrenza di /
        if(p != NULL)
            *p = '\0';
        if(strcmp(sessione->directory_corrente, "") == 0) //se la directory è vuota viene impostata a FTP_ROOT
            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);
    }

}
```

Metodologia e Implementazione

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.

Metodologia e Implementazione

Gestisci_comando

```
//Se l'argomento passato è un percorso assoluto passiamo al carattere successivo allo /, altrimenti rimarrà 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 è una directory
```

Metodologia e Implementazione

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.

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.

Metodologia e Implementazione

Gestisci_comando

```
        //se stat restituisce 0 allora il percorso esiste, la MACRO S_ISDIR controlla se il percorso è 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);
        }
    }
}
```

Metodologia e Implementazione

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.

Metodologia e Implementazione

Gestisci_comando

```
//Se il comando passato è LIST vengono ritornate le directory ed i relativi file
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 è 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);
```

Nel caso in cui il comando ricevuto fosse LIST è possibile elencare i file presenti all'interno della directory corrente.
Andiamo dunque ad aprire la directory mediante la funzione **opendir()** ed effettuiamo un controllo verificando che la directory sia stata trovata.

Se la directory viene aperta correttamente inoltriamo un messaggio con codice di risposta 150.

Metodologia e Implementazione

Gestisci_comando

```
    struct dirent *elemento;
    char lista[DIM_BUFFER];
    while((elemento = readdir(dir)) != NULL){//finchè vengono lette directory
        if(strcmp(elemento->d_name, ".") == 0 ||strcmp(elemento->d_name,"..") == 0 )//Se la directory è 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);
}
```

Viene poi dichiarato un puntatore alla struct dirent, chiamato elemento (che si occuperà di ricevere i vari elementi presenti all'iterno della directory), e un buffer chiamato lista che conterrà i vari elementi. 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.

Metodologia e Implementazione

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.

Metodologia e Implementazione

Gestisci_comando

```
//Se il comando ricevuto è RETR viene scaricato un file inoltrato dal client
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);
```

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.

Metodologia e Implementazione

Gestisci_comando

```
    }else{
            char percorso[PATH_MAX];//salvo il path
            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 modalità '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);
        }

}
```

Metodologia e Implementazione

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.

Metodologia e Implementazione

Gestisci_comando

```
//Se il comando è STOR viene caricato un file inoltrato dal client
else if(strcmp(cmd,"STOR") == 0){
    if (arg == NULL) {//se non vi è alcun file viene specificato il caso d'uso
        char msg[] = "501 Uso: STOR <nome_file>\n";
        send(sessione->client_fd, msg, strlen(msg), 0);
```

L'ultimo comando verificato è STOR.

Nel caso in cui il comando corrisponda, viene (come per RETR) effettuato un controllo relativo all'argomento specificato dall'utente su cui voler caricare dati.

Nel caso in cui l'argomento non venga specificato viene inoltrato un messaggio con codice di risposta 501 in cui viene spiegato come utilizzare correttamente il comando.

Gestisci_comando

```
        } 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 modalità 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) { //finchè ci sono dati da leggere
                buffer_dati[lunghezza] = '\0';
                if (strncmp(buffer_dati, "<EOF>", 5) == 0) //se il client ha inviato <EOF> interrompiamo il programma
                    break;
                fwrite(buffer_dati, 1, lunghezza, file);//in seguito scriviamo i dati ricevuti
            }
            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);
    }
}
```

Metodologia e Implementazione

Altrimenti se il nome del file su cui voler caricare i dati è stato fornito, definiamo un nuovo array di nome percorso e costruiamo il path attraverso la funzione **snprintf()**.
Successivamente apriamo il percorso specificato in modalità di scrittura. Questa modalità permette di creare automaticamente il file se non è già presente nella directory; se invece il file esiste, sarà possibile effettuare normali operazioni di scrittura.

Se l'apertura del file fallisce (ossia se il puntatore al file è NULL), interrompiamo il programma segnalando l'errore. Altrimenti, inviamo un messaggio all'utente per indicare l'inizio della fase di download, che dovrà terminare la scrittura sul file utilizzando la stringa EOF.

Autentica_Utente

```c
//Funzione per garantire l'accesso da parte di un utente
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)
                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;
}
```

Metodologia e Implementazione

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. 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.

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.

Makefile

```
1  CC = gcc
2  CFLAGS = -Wall -Wextra -pthread
3
4  CLIENT_DIR = client
5  SERVER_DIR = server
6
7  CLIENT_SRCS = $(CLIENT_DIR)/client.c $(CLIENT_DIR)/funzioni_client.c
8  SERVER_SRCS = $(SERVER_DIR)/server.c $(SERVER_DIR)/funzioni_server.c $(SERVER_DIR)/comandi.c
9
10 CLIENT_EXEC = $(CLIENT_DIR)/client.out
11 SERVER_EXEC = $(SERVER_DIR)/server.out
12
13 all: $(CLIENT_EXEC) $(SERVER_EXEC)
14
15 $(CLIENT_EXEC):
16     $(CC) $(CFLAGS) -o $@ $(CLIENT_SRCS)
17
18 $(SERVER_EXEC):
19     $(CC) $(CFLAGS) -o $@ $(SERVER_SRCS)
20
21 clean:
22     rm -f $(CLIENT_EXEC) $(SERVER_EXEC)
```

MakeFile

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.

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
```

Risultati

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:
```

OUTPUT CWD

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

Risultati

OUTPUT LIST

```
 Inserisci comando: LIST
150 Inizio elenco--
file.txt
226 Fine elenco.
Comandi disponibili: CWD,LIST,RETR,STOR,QUIT
 Inserisci comando:
```

OUTPUT RETR

```
 Inserisci comando: RETR file.txt
150 Inizio download...
Messaggio di output
226 Download completato.
Comandi disponibili: CWD,LIST,RETR,STOR,QUIT
 Inserisci comando:
```

Risultati

OUTPUT STOR

```
 Inserisci comando: STOR newfile.txt
150 Inizio caricamento, invia dati e termina con <EOF> su una linea.
Output comando STOR
<EOF>
226 caricamento completato.
Comandi disponibili: CWD,LIST,RETR,STOR,QUIT
 Inserisci comando:
```

OUTPUT QUIT

```
 Inserisci comando: QUIT
221 Arrivederci.
seby@seby-Inspiron-15-3525:~/Documenti/ProgettoReti/client$
```

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.

Conclusione e Sviluppi Futuri

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 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.