

Grid-FTP Client/Server Application

2018/2019

Contents

Introduzione	4
Obiettivo	4
Flusso dei dati	4
Registrazione	4
Autenticazione.....	4
Upload file	4
Donwload	5
Specifiche client.....	6
Client.....	6
Registrazione	6
Autenticazione.....	7
Upload file	8
Download	8
Specifiche metadata server	8
Specifiche data repository.....	9
Specifiche funzioni client.....	10
BlockList.....	10
CheckMD5	10
CreateSocket.....	10
DieWithError.....	10
MD5	10
PrintBlock.....	10
PrintList.....	11
Put.....	11
ReadFromSocket.....	11
WriteBlock	11
WriteToSocket	12
Specifiche funzioni metadata server	12
AcceptTCPConnection	12
CheckAvaliableUserName	12
CheckCredentials	12
CheckDoubleDR	13
CheckNewFile	13
CreateNewFolder.....	13
CreateSocket.....	13

CreateTCPServerSocket	13
DieWithError.....	13
DRStatus	13
GenrateKey	14
GetFileKey.....	14
HandleTCPClient	14
NotifyDR	18
ReadFromSocket.....	18
Registration	18
RemoveFile	18
WriteToFile	18
WriteToSocket	19
Specifiche funzioni data repository.....	19
AcceptTCPConnection	19
CreateSocket.....	19
CheckNewFile	19
CheckNewUser	19
CountFiles	20
CreateDRTCPServerSocket	20
CreateFolder	20
DieWithError.....	20
Get	20
HandleDRTCPClient	20
MD5	22
PrintBlock.....	22
ReadFromSocket.....	22
RemoveAuthFolder.....	23
RemoveFile	23
WriteBlock	23
WriteToFile	23
WriteToSocket	23
Note	24

Introduzione

Obiettivo

L'obiettivo del progetto è quello di creare un applicazione client/server GRID-FTP.

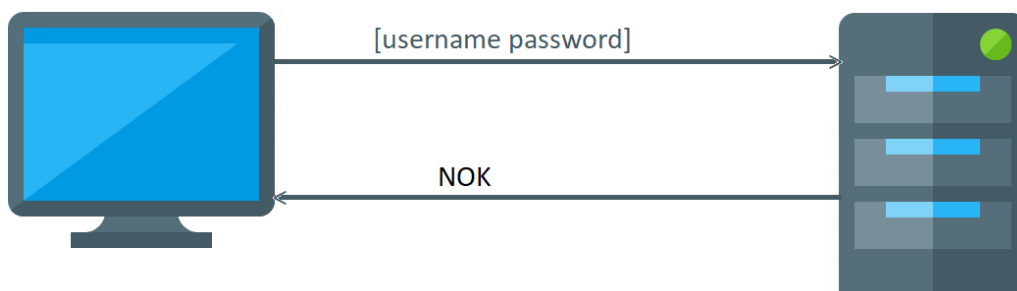
I componenti del progetto sono tree, clients, metadata server (MDS) e data repositories (DR) dove il metadata server è uno solo mentre i client e i repository possono essere più di uno.

Le operazioni possibili in questo sistema sono la registrazione, l'autenticazione dei client e upload e download dei file.

Flusso dei dati

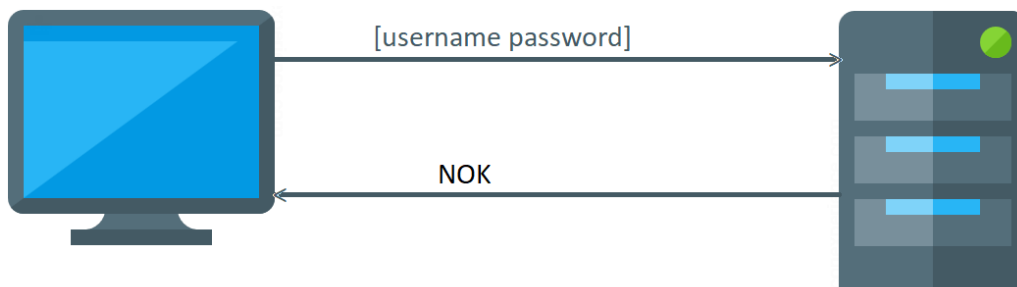
Registrazione

Il client invia username e password di propria scelta e il metadata server controlla se lo username scelto è disponibile, nel caso fosse disponibile registra l'utente in caso contrario manda un messaggio di errore.



Autenticazione

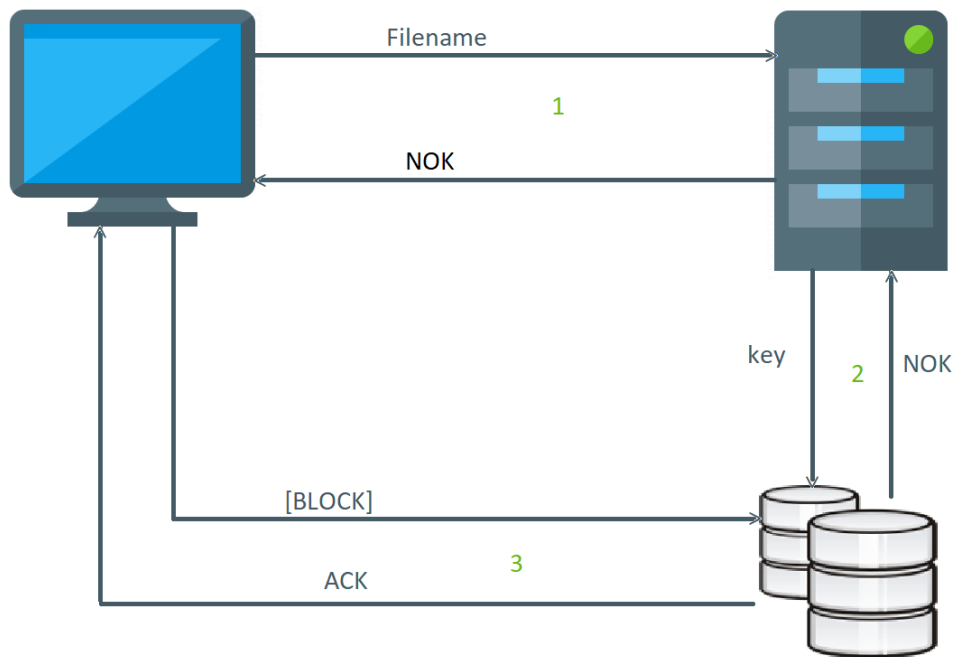
Il client invia username e password con cui si è precedentemente registrato e il metadata server provvede ad autenticarlo, in caso contrario invia un messaggio di errore.



Upload file

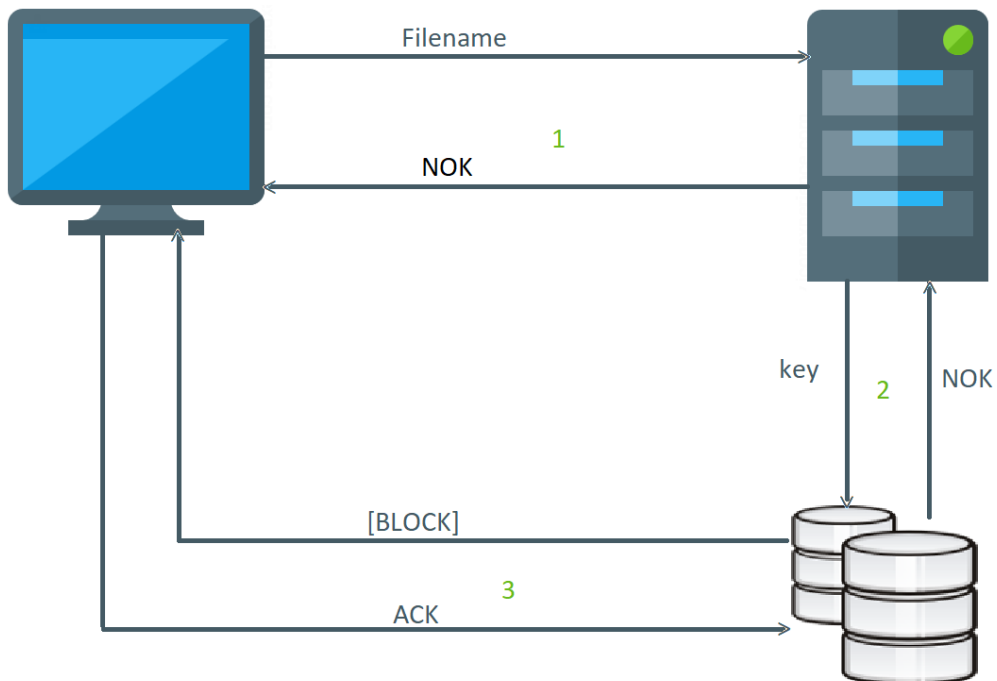
Il client sceglie il file da fare upload, se il file non esiste viene avvisato subito ancora prima di iniziare qualsiasi interazione con i server.

Se il file esiste viene mandata una richiesta al metadata server, esso controlla che il file non esista già, in caso contrario avvisa il client dei data repository da contattare per prelevare il file e nel frattempo avvisa i data repository dell'arrivo di una connessione.



Download

Il cliente sceglie il file da scaricare e invia il nome del file al metadata server, esso avvisa i data repository dell'arrivo di una nuova connessione e invia ip e porta dei data repository al client che potrà connettersi per scaricare i file.



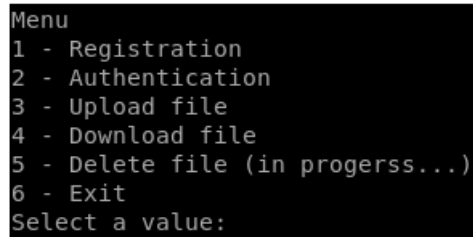
Specifiche client

Client

Il client permetterà di eseguire le seguenti operazioni:

- 1) Registrazione
- 2) Autenticazione
- 3) Upload file
- 4) Download file

Ad ognuna di queste operazioni corrisponde un numero che l'utente deve inserire.



```
Menu
1 - Registration
2 - Authentication
3 - Upload file
4 - Download file
5 - Delete file (in progress...)
6 - Exit
Select a value:
```

Il file è strutturato come una cascata di IF ed ELSE dove ad ogni IF si controlla l'opzione scelta e in base ad esso si eseguono delle operazioni.

Per il funzionamento del client c'è il bisogno del metadata server che sia in esecuzione, la prima operazione che esegue il client è quella di connettersi al metadata server e solo allora mostra il menu.

Il client tiene traccia se l'utente è attualmente autenticato o meno.

Per eseguire il client basta eseguire il file binario **./client**, bisogna conoscere a monte l'indirizzo IP e la porta del metadata server e modificare adeguatamente client.h dove ci sono le configurazioni per la connessione al metadata server.

Il client ha un bug dove alla fine di ogni operazione mostra il menu due volte anziché una.

Durante il download e l'upload di file binary c'è un bug dove per eseguire il comando successivo bisogna cancellare dei caratteri dal terminale finché si può e poi inserire il numero del comando da eseguire.

Registrazione

Opzione registrazione: 1

Per la registrazione si richiede all'utente di inserire un username e password, lo username deve contenere caratteri alfa numerici, altri caratteri speciali non sono ammessi e nel caso venisse richiesto di inserire il nome utente finché non viene rispettata la regola, mentre per la password non ci sono restrizioni.

```

Menu
1 - Registration
2 - Authentication
3 - Upload file
4 - Download file
5 - Delete file (in progerss...)
6 - Exit
Select a value: 1

User name (MAX 20): jafor
>> valid = 1
Password (MAX 20): jafor
Username: jafor, Password: jafor
> unamelen: 5, pwrlen: 5
> cmd: register jafor jafor

reading...

> Registration succeeded.

```

Entrambe le stringhe possono essere di una lunghezza massima di 20 caratteri.

Una volta inserite le due stringhe il client manda le informazioni al server creando un comando apposito riconoscibile dal server. Il comando per la registrazione è in questo formato **“register <username> <password>”**, il comando è formato dalla parola “command” e seguito dal nome utente e password scelti dall’utente divisi dallo spazio e senza le parentesi angolari.

Per inviare il comando viene utilizzata la funzione **writetosocket(int socket, char* string, int size)**.

Lo stato della registrazione viene mostrato a video sia che abbia avuto successo o meno.

Per qualche ragione dopo la registrazione non si riesce a fare l’autenticazione anche se la registrazione va a buon fine, in questo caso basta riavviare il client e procedere con l’autenticazione.

Autenticazione

Opzione autenticazione: 2

Durante il processo di autenticazione viene chiesto all’utente di inserire le proprie credenziali.

```

Menu
1 - Registration
2 - Authentication
3 - Upload file
4 - Download file
5 - Delete file (in progerss...)
6 - Exit
Select a value: 2

User name (MAX 20): jafor
Password (MAX 20): jafor
Username: jafor, Password: jafor
> unamelen: 5, pwrlen: 5
> cmd: auth jafor jafor

reading...

> Successfully authenticated.

```

La prima cosa che viene fatto è un controllo se l'utente è stato precedentemente autenticato o meno, nel caso non lo fosse chiede di inserire username e password che possono essere lunghe massimo venti caratteri, nel caso l'utente si è già autenticato avvisa con un messaggio e stampa di nuovo il menù.

Una volta inserite le credenziali per l'autenticazione viene creato il comando per il metadata server e il comando è in questo formato **"auth <username> <password>"**, il comando è composto dalla parola "auth" e le credenziali divisi dallo spazio senza parentesi angolari.

Lo stato dell'autenticazione viene mostrato a video.

Upload file

Opzione upload file: 3

Durante la fase di autenticazione viene chiesto all'utente di inserire il nome del file.

Il passo è quello di calcolare la dimensione del file che sarà poi utilizzata dal metadata server per suddividere i blocchi nel modo più equo possibile tra i vari data repositories.

Nel caso il file inserito non esistesse viene mostrato un messaggio di errore e mostra di nuovo il menù così da poter ricominciare da capo.

Una volta ottenuto il peso del file viene creato un comando per il metadata server per mandare una richiesta di upload mandando un messaggio nel seguente formato **"put <filename> <filesize>"**, il comando è formato dalla parola "put", nome del file e dalla dimensione del file separati con lo spazio senza parentesi angolari.

Se è andato tutto bene dal lato server verrà inviato come primo messaggio al client la chiave del file che verrà poi utilizzata presso i data repositories per autenticarsi temporaneamente.

Appena ricevuta la chiave il client crea il comando da inviare al data repository nel seguente formato **"clntinf <username> <filename> <key>"**, il comando è formato dalla parola "clntinf", nome dell'utente che esegue l'operazione, nome del file da uploadare e la chiave che serve per autenticarsi.

Dopo di che il client riceverà in sequenza le informazioni riguardanti i data repositories di cui IP, PORT e il numero di blocchi da inviare nel seguente formato **"<IP> <PORT> <#blocks>"** dove IP e PORT sono del data repository e #blocks è il numero di blocchi che il client deve inviare al data repository.

Per ogni data repository che riceve il client legge tanti pezzi (chunks) quanti specificati dal metadata server (#blocks), e crea una lista di blocchi con la funzione **struct blocks* blocklist(struct blocks* blst, int blockindex, char* key, char* chunk, char* md5)**, creata la lista viene passata alla funzione **int put(char* drip, char* drport, struct blocks* blst, char* clntinf)**, che provvederà ad inviare correttamente i blocchi alla repository dedicata.

In caso di errore si chiude il client.

Download

Opzione download file: 4

Durante la fase di autenticazione viene chiesto all'utente di inserire il nome del file e viene invocata la funzione **int get(char* filename, char* username, int sock)** che si occupa di tutto il processo. per maggiori dettagli vedere specifiche della funzione.

Specifiche metadata server

L'inizializzazione del metadata server consiste nel creare un semaforo per gestire la scrittura di alcuni file, eseguire un processo figlio int **corruptionchecker()** per controllare le incongruenze con i metadata server e infine rimanere in ascolto per le future connessioni da parte dei client e i metadata server.

Nel caso il semaforo esiste già viene cancellato e ricreato.

Il metadata server al momento accetta al massimo 20 connessioni ma si può facilmente modificare per far accettare richieste illimitate.

Per ogni nuovo client che si connette viene dedicato un processo figlio che esegue la funzione **void HandleTCPClient(int clntSocket, char* connectionip, sem_t* drdata_semaphore)**, per maggiori dettagli vedere la specifica della funzione.

Per eseguire il metadata serve bisogna dare come input da linea di comando la porta dove si esegue ovvero **./server <PORT>**.

Le credenziali degli utenti vengono salvati in un file chiamato **credentials** la struttura del file è nel seguente formato "**<username> <password>**", nella prima colonna sono salvati i nomi utenti e nella seconda colonna le rispettive password.

I dati dei repository vengono salvati in un file chiamato **drlist.conf**, nel seguente formato "**<IP> <PORT> <AvailableStorage>**", il campo available storage non è utilizzato perché si suppone per semplicità che le data repository abbiano sempre spazio a disposizione, nel caso si voglia tenere conto dello spazio dei repository il file è già predisposto per le successive modifiche.

Inoltre quando un utente fa un upload per la prima volta viene creata una cartella con il nome dell'utente e all'interno viene creato un file speciale che ha per nome il nome dell'utente contenente la lista dei file uploadati quindi disponibili nei repository, viene creato anche un file che per nome il nome del file uploadato che contiene la lista dei repository in cui si trovano i blocchi e le rispettive quantità dei blocchi per repository.

Durante il download di file jpg si crea una cartella con caratteri speciali non riconosciuto dal sistema operativo.

Specifiche data repository

Il data repository per essere eseguito bisogna specificare da riga di comando la porta dove esso deve girare, l'indirizzo ip e la porta del metadata server.

La prima operazione eseguita dalla data repository è quella di notificare il metadata server della sua presenza eseguendo la **funzione int notifymds(char* mdsip, char* mdsport, char* drport)**, e quindi rimane in ascolto per le connessioni da parte del metadata server e client.

La data repository al momento accetta massimo al 30 connessioni ma si può facilmente modificare per far accettare richieste illimitate.

Per maggiori informazioni vedere la specifica della funzione **void HandleDRTCPClient(int clntSocket, int totalstorage, int usedstorage)**.

La lista degli utenti che hanno depositato i blocchi sono salvati nel file definito dalla variabile **USER_CONFIG_FILE**.

La repository è organizzata a cartelle, ogni utente ha una cartella e all'interno una per ogni file dove risiedono i blocchi dei file.

Per ogni richiesta di upload e download che viene notificato dal metadata server la repository crea un file dentro la cartella specificata da **AUTH_FOLDER** con il nome della chiave che sarà eliminato una volta che riceverà la stessa chiave da parte del client.

Specifiche funzioni client

BlockList

`struct blocks* blocklist(struct blocks* blst, int blockindex, char* key, char* chunk, char* md5)`

La funzione si occupa di creare una lista linkata contenente i blocchi necessari per l'invio ad una data repository.

La funzione prende in input una lista **blst** del tipo **struct blocks***, l'indice del singolo blocco **blockindex**, la chiave del blocco assegnata dal metadata server **key**, la porzione del file letto in bytes e l'hash md5 della porzione del blocco **md5**.

In output ritorna la lista **blst** a cui è stato aggiunto l'elemento.

La funzione crea un blocco fissato di byte lungo **BLOCK_SIZE** a cui vengono concatenati i parametri in ingresso come bytes.

Una volta creato il blocco di bytes si crea una variabile del tipo **struct blocks** che ha un array di **char*** a cui verrà assegnato il blocco creato.

Adesso che il blocco è all'interno della variabile struct si scorre la lista fino a raggiungere l'ultimo elemento per poter concatenarlo in coda.

CheckMD5

`int checkmd5(char* block)`

Funzione che prende in ingresso un blocco ([#, key, hash, chunk]) e controlla se il blocco è corrotto o meno.

Il blocco passato in ingresso viene scompattato così da avere ogni elemento del blocco in una variabile apposita, poi si calcola l'hash del blocco con la funzione **char* md5(char* chunk)** e la si confronta con l'hash contenente nel blocco.

Se gli hash sono uguali, ciò significa che il blocco non è corrotto, viene ritornato l'indice del blocco ovvero # altrimenti viene ritornato il -1.

CreateSocket

`int createsocket(char* ip, char* port)`

La funzione ha come parametro IP e PORT e crea una socket, il codice è lo stesso visto a lezione.

In caso di successo ritorna il valore della socket altrimenti stampa un messaggio di errore e ritorna -1.

DieWithError

`void DieWithError(char *errorMessage)`

Funzione che prende una stringa in ingresso da stampare come messaggio di errore e chiude il programma.

MD5

`char* md5(char* chunk)`

Funzione che prende come parametro una stringa e ritorna l'hash della stringa come valore di ritorno.

PrintBlock

`int printblock(char* block)`

Funzione che prende come parametro un blocco.

Il blocco viene scompattato per ottenere tutti i valori separati ed assegnati a variabili di tipo compatibile per poter essere stampati.

La funzione stampa caratteri randomici in caso il blocco contenga porzioni di file non testuali.

PrintList

```
int printlist(struct blocks* head)
```

La funzione prende in ingresso una lista di blocchi.

La lista viene ciclata e per ogni blocco viene stampato il contenuto scompattando tutti i componenti dei singoli blocchi e mettendoli in apposite variabili compatibili dato che le liste contengono bytes.

La funzione potrebbe essere migliorata facendo uso della funzione **int printblock(char* block)** invece di scompattare e stampare che è ciò che fa printblock.

La funzione stampa caratteri randomici in caso il blocco contenga porzioni di file non testuali.

Put

```
int put(char* drip, char* drport, struct blocks* blst, char* clntinf)
```

Funzione che prende in ingresso IP e PORTA di un data repository, lista dei blocchi da inviare e un comando da inviare alla data repository per avvisarlo di un imminente trasferimento di blocchi.

Il primo passo è quella di creare una socket per connettersi al metadata serve in questione attraverso la funzione **int createsocket(char* drip, char* drport)**.

Una volta connessi con la repository la funzione avvisa la data repository di un imminente trasferimento inviando il parametro d'ingresso **clntinf** che è un comando nel seguente formato "**clntinf <username> <filename> <key>**" e viene inviato attraverso la funzione **int writetosocket(int socket, char* string, int size)**.

Adesso tramite un ciclo vengono inviati ogni blocco alla repository con la funzione **int writetosocket(int socket, char* string, int size)** e subito dopo si aspetta per un Ack da parte del server, viene controllato se l'Ack ricevuto è uguale alla stringa "ERROR", in caso affermativo si riprocede ad inviare di nuovo il blocco.

Una volta inviati tutti i blocchi viene inviato un Ack da parte del client per indicare la fine della trasmissione e chiude la socket.

In caso di un Ack uguale alla stringa "ERROR" la funzione reinvia il pacchetto all'infinito finché non viene ricevuto un Ack positivo e ciò può mandare in stallo il client.

ReadFromSocket

```
char* readfromsocket(int socket, int size)
```

La funzione prende in ingresso una socket e una dimensione.

Viene letto dalla socket bytes della dimensione indicata e ritorna un buffer contenente il contenuto letto dal buffer.

In caso di errore si chiude il programma.

WriteBlock

```
int writeblock(char* chunk, char* path)
```

La funzione prende in ingresso una porzione del file e un path dove scrivere i byte della porzione.

La funzione apre il file in modalita lettura, se il file non esiste viene creato e le porzioni dei file vengono scritti in modalita append.

I file vengono creati con i permessi di lettura, scrittura ed esecuzione (00700) da parte del proprietario del file.

La funzione non ritorna nessun valore di ritorno e non fa controlli sulla syscall open.

WriteToSocket

```
int writetosocket(int socket, char* string, int size)
```

Funzione che prende in ingresso una socket, una stringa di bytes e una dimensione.

La funzione invia alla socket dedicata la stringa di byte della dimensione specificata.

Nel caso la dimensione sia diversa da quella della stringa si potrebbe inviare meno bytes o sfiorare la stringa ed inviare piu bytes del dovuto.

In caso di errore si chiude il client.

Specifiche funzioni metadata server

AcceptTCPConnection

```
struct connection AcceptTCPConnection(int servSock)
```

Funzione per accettare una connessione esterna da parte di un client o data repository.

Il codice è quello usato a lezione.

CheckAvailableUserName

```
int checkavailableusername(char* username)
```

La funzione prende in ingresso un nome utente e controlla nel file **credentials** se esiste un utente con lo stesso nome, il file **credentials** viene aperto in modalita lettura.

Se esiste già un utente con il nome indicato viene ritornato il valore 1 nel caso il nome utente è disponibile viene ritornato il valore 0.

Nel caso ci siano problemi con l'apertura del file viene mostrato un messaggio di errore ma non vengono prese altre azioni quindi il codice può andare in errore.

CheckCredentials

```
int checkcredential(char* username, char* passwd)
```

La funzione prende in ingresso username e password da autenticare.

Viene aperto il file **credentials** in modalita lettura, viene controllato per primo se esiste un nome utente uguale al username indicato nel parametro della funzione se esiste allora viene controllato se anche la password combacia.

In caso di successo viene ritornato il valore 0, autenticato con successo, in caso contrario viene ritornato il valore 1, autenticazione fallita.

Nel caso ci siano problemi con l'apertura del file viene mostrato un messaggio di errore ma non vengono prese altre azioni quindi il codice può andare in errore.

CheckDoubleDR

```
int checkdoubledr(char* connectionip, char* portsize)
```

La funzione prende in input IP (della repository) e una stringa contenente la porta e lo spazio su disco, controlla se la repository è già presente.

La funzione si ricava la porta e lo spazio su disco dal parametro **portsize**, apre il file drlist.conf in modalità lettura e legge riga per riga. Da ogni riga viene estratto l'indirizzo IP e la porta della repository e controllata con quella di input se combaciano allora la repository era stata salvata in precedenza e ritorna il valore 1, altrimenti se la repository è nuova viene ritornato il valore 0.

Nel caso la ci siano problemi con l'apertura del file non viene fatto alcun controllo e ciò può mandare in errore il programma nel caso il file non esista.

CheckNewFile

```
int checknewfile(char* username, char* filename)
```

La funzione prende in input il nome utente e il nome del file (da fare upload) e controlla se è già stato fatto l'upload o meno.

La funzione crea il path partendo dal nome utente e concatena ad esso il carattere "/" e il nome del file dato che ogni utente ha una cartella dove all'interno ci sono i file che hanno per nome i nomi dei file uploadati che contengono la lista dei data repository in cui si trovano i blocchi.

Una volta creato il path si tenta di aprire il file in modalità lettura e si controlla se l'apertura del file è andata a buon fine o meno.

Se l'apertura del file è andata a buon fine vuol dire che il file esiste ed è stato fatto l'upload del file e viene ritornato 1 altrimenti il file non esiste ed è un nuovo file da uploadare quindi viene ritornato 0.

CreateNewFolder

```
int createfolder(char* foldername)
```

La funzione prende come parametro una stringa (nome della cartella) e crea una cartella con i permessi di lettura, scrittura ed esecuzione per l'utente (0700).

CreateSocket

```
int createsocket(char* ip, char* port)
```

La funzione ha come parametro IP e PORT e crea una socket, il codice è lo stesso visto a lezione.

In caso di successo ritorna il valore della socket altrimenti stampa un messaggio di errore e ritorna -1.

CreateTCPServerSocket

```
int CreateTCPServerSocket(unsigned short port)
```

Funzione che si occupa di creare la socket, il codice usato è quello visto a lezione.

DieWithError

```
void DieWithError(char *errorMessage)
```

Funzione che prende una stringa in ingresso da stampare come messaggio di errore e chiude il programma.

DRStatus

```
int drstatus(char* drip, char* drport)
```

Funzione che si occupa di connettersi ad una repository e creare una socket.

Se la creazione della socket va a buon fine e riesce a connettersi vuol dire che la repository è UP e ritorna 0 altrimenti la repository è DOWN e ritorna 1.

Il codice è lo stesso di **int createsocket(char* ip, char* port)** quindi può essere migliorato chiamando direttamente createsocket.

Funzione ridondante.

GenerateKey

char* generatekey(int keysize)

Funzione che prende come parametro la dimensione della chiave e genera una stringa randomica alfanumerica di data lunghezza.

Per la funzione randomica viene usato come seed il tempo attuale in microsecondi in modo da essere il più randomici possibile.

La chiave non contiene caratteri alfabetici minuscoli ma solo maiuscoli e numeri.

Ritorna la chiave.

GetFileKey

char* getfilekey(char* username, char* filename)

La funzione prende come parametro il nome dell'utente e il nome del file e ottiene la chiave del file.

La funzione crea il path del file da aprire concatenando "username"+"/"+"username" che è il file speciale che tiene traccia di tutti i file dell'utente specificato e le loro chiavi.

Si tenta di aprire il file in modalità lettura e si legge riga per riga, appena si trova la corrispondenza con il nome del file del parametro ritorna la chiave in caso contrario se il file non è presente nella lista si ritorna NULL.

Viene fatto il controllo durante l'apertura del file nel caso il file speciale delle corrispondenze non esista si ritorna comunque NULL.

HandleTCPClient

void HandleTCPClient(int clientSocket, char* connectionip, sem_t* drdata_semaphore)

È la funzione principale del metadata serve in quanto si occupa di gestire le richieste da parte dei client e data repository.

La funzione è strutturata come una cascata di IF ed ELSE dove ogni if esegue un comando ricevuto da parte del client o data repository.

Al momento il metadata serve esegue i seguenti comandi: auth, info, get, put, register, drup, delete, exit.

La funzione è sempre in ascolto, ricevuto un messaggio estrae il comando ed esegue le operazioni del comando.

Comando "auth"

Formato comando: auth <username> <password>

Questa sezione si occupa di autenticare il client.

Estrae dal messaggio ricevuto il nome utente e la password, controlla l'esistenza delle credenziali tramite il metodo **int checkcredential(char* username, char* passwd)**.

Se le credenziali sono corrette viene creata una stringa di risposta con il messaggio "Successfully authenticated.\n" altrimenti la stringa è "Authentication failed.\n" e in fine viene inviato al client tramite la funzione **int writetosocket(int socket, char* string, int size)**.

Salva il nome utente nel caso serva per i comandi successivi.

La funzione ritorna ad essere in ascolto per il prossimo comando da ricevere.

Comando "info"

Formato comando: info <user/filename>

Questo comando viene ricevuto da parte di un data repository durante il controllo dell'integrità dei blocchi ovvero controlla se la repository ha la quantità esatta dei blocchi che dovrebbe avere. Il file che si controlla è del tipo username/filename contenente gli indirizzi IP, porte e numero dei blocchi dei vari data repository in cui si trovano i blocchi del file.

Si estrae il path dal messaggio ricevuto e si procede ad aprire il file in modalità lettura. Si legge il file riga per riga, trovata la corrispondenza IP e Porta si estrae la quantità di blocchi e la si invia alla repository tramite la funzione **writetosocket(int socket, char* string, int size)**.

Nel caso non si trova nessuna corrispondenza viene inviato il -1 in formato stringa.

Esempio file username/filename:

192.168.1.3 5555 1

192.168.1.4 1234 3

192.168.1.5 6677 2

Durante l'apertura del file non viene fatto nessun controllo, se il file è inesistente il programma va in errore.

Command "get"

Formato comando: get <filename>

Questo comando viene ricevuto dal client per effettuare il download di un file.

Viene estratto il nome del file, viene controllato che il file esista con la funzione **int checknewfile(char* username, char* filename)**, nel caso il file non esista viene mandato un messaggio al client e si aspetta per un nuovo comando, in caso il file esista si estrae la chiave con la funzione **char* getfilekey(char* username, char* filename)**, si compone il messaggio da inviare al client "KEY <chiave>", si invia il messaggio con la funzione **writetosocket(int socket, char* string, int size)**, e aspetta un Ack da parte del client.

Ricevuto l'Ack da parte del client si procede all'apertura del file "username/filename", si legge riga per riga e ogni riga, per ogni riga si estrae IP e Porta per notificare la repository di una connessione in arrivo, per fare ciò si usa la funzione **int notifydr(char* drip, char* drport, char* username, char* key)**.

Una volta notificata la repository viene inviato la riga letta del file al client con la funzione **writetosocket(int socket, char* string, int size)** così che può contattare le repository e sa quanti blocchi deve inviare per ogni repository.

Per ogni riga inviata si aspetta un Ack di risposta che indica la ricezione del messaggio e una volta finito di inviare si invia un Ack finale che segnala la fine della trasmissione.

Quando viene invocato la funzione **char* getfilekey(char* username, char* filename)** non viene fatto nessun controllo sul valore di ritorno.

Durante l'invio della chiave si controlla l'Ack in caso di errore viene mostrato un messaggio ma non viene fermato l'esecuzione del programma.

Durante l'apertura del file "username/filename" non viene controllato se l'apertura del file in modalità lettura sia andata a buon fine.

Comando "put"

Formato comando: put <filename> <filesize>

Questo comando viene ricevuto dal client per effettuare l'upload di un file.

Viene estratto il nome del file e la dimensione del file dal messaggio.

Viene effettuato un controllo sul file name, nel caso sia vuoto si ritorna in ascolto di un nuovo comando altrimenti si prosegue.

Nel caso in cui si prosegue viene controllato se il file esiste già o meno con la funzione **int checknewfile(char* username, char* filename)**, se il file esiste già mostra un messaggio e invia il messaggio del file già esistente al client tramite la funzione **writetosocket(int socket, char* string, int size)**.

Si calcola il numero di blocchi con la seguente formula **totblocks = [filesize/CHUNK_SIZE]**, si apre il file contenente la lista delle repository in modalità lettura e per ogni repository nel file si contano le repository attualmente UP e disponibili con la funzione **int drstatus(char* drip, char* drport)**.

Si calcola il minimo numero di blocchi che ogni repository deve avere, se il numero totale dei repository disponibili è minore del numero totale dei blocchi allora si calcola il numero minimo di blocchi che ogni repository deve avere **minblock = totblocks/totrepo** e il resto **leftblocks = totblocks % totrepo** altrimenti **minblock = totblocks**.

Adesso bisogna distribuire i blocchi rimasti (se ce ne sono) leftblocks tra tutti i dr in ordine partendo dal primo.

Si crea un array di valori lungo **totblocks** e a ogni cella assegno **totblocks+1, leftblocks** volte.

Si genera una chiave e la si invia al client con la funzione **writetosocket(int socket, char* string, int size)**.

Si crea un file di associazioni nomefile e chiave se non esiste già, si apre il file in modalità append e create e si scrive dentro i nuovi valori.

Si apre il file contenente la lista dei repository in modalità lettura, il file "username/filename" in modalità append e create e viene scritto IP, Porta e numero di blocchi in quest'ultimo file e invia al client le stesse informazioni.

Alla fine invia un Ack per avvisare il client della fine della trasmissione.

Durante l'apertura del file contenente la lista dei data repository non viene controllato se l'apertura del file ha avuto successo o meno.

Durante l'apertura in modalità append e create del file dell'utente (username/filename) non viene controllato se l'apertura del file ha avuto successo o meno.

Durante l'apertura del file contenente la lista dei data repository per la seconda volta non viene controllato se l'apertura del file ha avuto successo o meno.

Command "register"

Formato comando: register <username> <password>

Il comando viene inviato dal client per effettuare la registrazione.

Viene estratto dal messaggio il nome utente e la password, viene invocata la funzione **int registration(char* username, char* passwd)**, passando come parametro le credenziali ottenute.

Viene controllato il valore di ritorno della funzione, se è 0 allora si invia un messaggio di successo altrimenti si invia un messaggio di errore e in entrambi i casi viene usata la funzione **writetosocket(int socket, char* string, int size)** per inviare.

Viene creata una cartella con il nome dell'utente e non viene tenuto conto se la registrazione ha avuto successo o meno.

Command "drup"

Formato comando: drup <drport> <storage>

Comando inviato dai data repository per avvisare il metadata server della loro presenza.

Viene estratta la porta e la dimensione della repository e invocata la funzione **int checkdoubledr(char* connectionip, char* portsize)** passando i parametri appropriati.

Nel caso la repository sia già presente si mostra solo un messaggio e il programma ritorna in ascolto per il successivo comando.

Qualora la repository non fosse presente e fosse nuova si apre il file contenente la lista delle repository in modalità append e si aggiungono i dati relativi alla repository.

Se l'apertura del file genera un errore il programma si chiude.

L'operazione di apertura del file, scrittura e chiusura viene effettuata con un semaforo per evitare race conditions.

In fine si invia un Ack di conferma alla repository per indicare la corretta registrazione.

Command delete

Formato comando: delete <username> <filename>

Il comando viene mandato dal client per eliminare un file presente nei repository.

Vengono estratti il nome utente e il nome del file da eliminare.

Viene creato il path del file che contiene la lista dei repository dove risiedono i blocchi (username/filename). Viene aperto il file in modalità lettura e controllato se l'apertura ha avuto successo o meno.

Se l'apertura del file non ha avuto successo si mostra un messaggio e il programma ritorna in ascolto del prossimo comando altrimenti si prosegue.

Una volta aperto il file con successo si legge riga per riga e per ogni riga si ricava IP e Porta dei repository, viene effettuata la connessione alla repository e creato il comando per eliminare i blocchi, formato è la seguente "delete <username> <filename>".

Si invia il comando alla repository, viene eliminato il file nel metadata server (username/filename) e viene eliminato il file dalla lista dei file del user (username/username).

Durante l'apertura del file speciale del user (username/filename) non viene controllato se l'apertura è andata a buon fine o meno.

Command "exit"

Formato comando: exit

Il comando può essere inviato da chiunque, chiude la socket e il programma.

NotifyDR

int notifydr(char* drip, char* drport, char* username, char* key)

Notifica la repository di una connessione da parte di un client e consegna una chiave che verrà usata dal client per autenticarsi.

La funzione prende come parametro ip e porta della repository e il nome utente ed una chiave che verrà usata come chiave di autenticazione da parte del client.

La funzione crea una socket e si connette alla repository, viene creato il comando nel seguente formato "newclnt <username> <key>" e viene inviato il comando.

Si aspetta di ricevere un Ack che è la conferma della ricezione da parte della repository.

ReadFromSocket

char* readfromsocket(int socket, int size)

La funzione prende in ingresso una socket e una dimensione.

Viene letto dalla socket bytes della dimensione indicata e ritorna un buffer contenente il contenuto letto dal buffer.

In caso di errore si chiude il programma.

Registration

int registration(char* username, char* passwd)

La funzione prende come parametro il nome utente e la password e controlla se il nome utente è disponibile con la funzione **int checkavailableusername(char* username)** e viene controllato il valore di ritorno.

Se il nome utente non è disponibile ritorna 1 altrimenti si apre il file contenente le credenziali di tutti gli utenti e si appende il nome utente e la password nel file.

Non viene controllato l'esito dell'apertura del file in modalità append.

Non viene usato un semaforo per scrivere le nuove credenziali dentro il file comune a più client.

RemoveFile

int removefile(char* path)

La funzione prende come parametro il path di un file da eliminare e lo elimina.

In caso di successo ritorna 0 altrimenti 1.

WriteToFile

int writetofile(char* info, char* path)

Funzione che prende come parametro la stringa da scrivere e il path del file dove scrivere.

Si apre il file in modalita appen e create e e viene scritto la stringa.

Nel caso l'apertura del file sia andata in errore viene mostrato un messaggio e viene ritornato il valore 1 altrimenti 0.

WriteToSocket

```
int writetosocket(int socket, char* string, int size)
```

Funzione che prende in ingresso una socket, una stringa di bytes e una dimensione.

La funzione invia alla socket dedicata la stringa di byte della dimensione specificata.

Nel caso la dimensione sia diversa da quella della stringa si potrebbe inviare meno bytes o sfiorare la stringa ed inviare piu bytes del dovuto.

In caso di errore si chiude il client.

Specifiche funzioni data repository

AcceptTCPConnection

```
struct connection AcceptTCPConnection(int servSock)
```

Funzione per accettare una connessione esterna da parte di un client o data repository.

Il codice è quello usato a lezione.

CreateSocket

```
int createsocket(char* ip, char* port)
```

La funzione ha come parametro IP e PORT e crea una socket, il codice è lo stesso visto a lezione.

In caso di successo ritorna il valore della socket altrimenti stampa un messaggio di errore e ritorna -1.

CheckNewFile

```
int checknewfile(char* username, char* filename)
```

La funzione prende in ingresso il nome utente e il nome del file (da fare upload) e controlla se è già stato fatto l'upload o meno.

La funzione crea il path partendo dal nome utente e concatena ad esso il carattere "/" e il nome del file dato che ogni utente ha una cartella dove all'interno ci sono i file che hanno per nome i nomi dei file uploadati che contengono la lista dei data repository in cui si trovano i blocchi.

Una volta creato il path si tenta di aprire il file in modalita lettura e si controlla se l'apertura del file è andata a buon fine o meno.

Se l'apertura del file è andata a buon fine vuol dire che il file esiste ed è stato fatto l'upload del file e viene ritornato 1 altrimenti il file non esiste ed è un nuovo file da uploadare quindi viene ritornato 0.

CheckNewUser

```
int checknewuser(char* username)
```

Funzione che controlla se l'utente connesso è nuovo quindi è la sua prima connessione è già connesso altre volte.

Si apre il file definito nella variabile USER_CONFIG_FILE in modalità lettura e viene confrontato se il nome utente specificato compare nel file.

In caso di utente esistente viene ritornato 1 altrimenti 0.

Nel caso in cui l'apertura del file USER_CONFIG_FILE generi un errore viene mostrato un messaggio di errore e viene ritornato 0.

CountFiles

```
int countfiles(char* path)
```

Funzione che prende come parametro il path di una cartella e conta il numero di file al suo interno, escludendo la cartella corrente e la cartella superiore (. e ..).

Viene ritornato il numero di file presenti.

Non viene controllato l'esito dell'apertura della cartella.

CreateDRTCPServerSocket

```
int CreateDRTCPServerSocket(unsigned short port)
```

Funzione che si occupa di creare la socket, il codice usato è quello visto a lezione.

CreateFolder

```
int createfolder(char* foldername)
```

La funzione prende come parametro una stringa (nome della cartella) e crea una cartella con i permessi di lettura, scrittura ed esecuzione per l'utente (0700).

DieWithError

```
void DieWithError(char *errorMessage)
```

Funzione che prende una stringa in ingresso da stampare come messaggio di errore e chiude il programma.

Get

```
int get(char* path, int socket, char* key)
```

La funzione si occupa di inviare i blocchi al client quando esegue un download.

Funzione che riceve come parametro il path della cartella dove prendere i blocchi, una socket a cui inviare i blocchi e la chiave di autenticazione.

Viene controllato la chiave di autenticazione attraverso la funzione `int removeauthfile(char* key)` e viene controllato l'esito, in caso di fallimento si chiude la connessione e il processo si chiude.

Si apre la cartella e si scansionano i file, e per ogni file (tranne . e ..) la si apre, si legge il blocco e la si invia alla socket, per ogni blocco si aspetta un Ack di risposta alla repository.

L'invio dei blocchi è sequenziale dato che dopo l'invio di ogni blocco aspetta un Ack come risposta.

L'apertura della cartella non è controllata in caso non esista.

HandleDRTCPClient

```
void HandleDRTCPClient(int clntSocket, int totalstorage, int usedstorage)
```

E' la funzione principale della repository che si occupa di richieste da parte del client e del metadata server.

la funzione è strutturata come una cascata di IF ed ELSE dove ogni if esegue un comando ricevuto da parte del client o metadata server.

Al momento la data repository esegue i seguenti comandi: `newclnt`, `clntinf`, `download`, `info`, `delete`.

Una volta che il lcinet o il metadata server è connesso il processo figlio rimane sempre in attesa di ricevere un comando.

Comando "newclnt"

Formato comando: newclnt <username> <key>

Questo comando viene mandato dal metadata server per avvisare dell'arrivo di un connessione

Si ricava il nome utente e la chiave dal messaggio, viene mandato un Ack per avvisare della corretta ricezione del messaggio. Viene creato una cartella con il nome specificato nel file di configurazione AUTH_FOLDER tramite la funzione **int createfolder(char* foldername)**.

Si crea il path di un file che avrà per nome la chiave specificata nel messaggio e sarà creata all'interno della cartella AUTH_FOLDER.

Per creare il file viene usato il metodo **int writetofile(char* path, char* string)** che si occupa anche di creare il file solo che come parametro di stringa viene inviato una stringa vuota dato che non si vuole scrivere alcun contenuto all'interno del file, basta il nome del file che è la chiave.

Viene controllato se l'utente specificato nel messaggio è un nuovo utente o meno e nel caso fosse un utente nuovo viene aggiunto il nome dell'utente al file contenente la lista degli utenti tramite la funzione **int writeblock(char* block, char* path)**.

Comando "clntinf"

Formato comando: clntinf <username> <filename> <key>

Il comando viene inviato dal client per effettuare il download dei blocchi.

Vengono estratti il nome utente, nome del file e la chiave dal messaggio e si tenta di rimuovere il file di autenticazione che ha per nome la chiave del messaggio tramite la funzione **int removeauthfile(char* key)**.

Si controlla l'esito della funzione e se non è stato possibile rimuovere, manca presenza file, allora la repository chiude la connessione con il client.

In caso di rimozione del file con successo che indica l'autenticazione avvenuta con successo da parte del client il server rimane in ascolto per ricevere i blocchi uno alla volta usando la funzione **char* readfromsocket(int socket, int size)**, per ogni blocco ricevuto viene controllato l'hash MD5 della porzione del file all'interno del blocco tramite la funzione **int checkmd5(char* block)**, se il controllo dell'hash va a buon fine manda un Ack al client con la funzione **int writetosocket(int socket, char* string, int size)**, per richiedere il blocco successivo.

Inviato l'Ack si scrive il nome del file all'interno del file speciale che contiene la lista dei file dell'utente (username/username.conf). Ciò viene fatto una volta per file.

Si crea il path per salvare il blocco all'interno della cartella del user (username/filename/#block).

Se il nome del file contiene il carattere "." Viene ingorato tutto quello che viene dopo

- mario.jpg -> mario
- img.png -> img
- a.out -> a
- test -> test

Ogni blocco ha per nome del file l'indice del blocco così sono sempre ordinati.

Comando "download"

Formato comando: download <username> <filename> <key>

Il comando viene inviato dal client per effettuare il download dei blocchi presenti nella repository.

Viene creato il path della cartella dove risiedono i blocchi (username/filename/) e viene invocato il metodo `int get(char* path, int socket, char* key)` che si occupa di interagire con il client.

Comando "info"

Formato comando: info <pathfolder>

Il comando viene inviato dal metadata server per controllare se la quantità di blocchi che il metadata server ha è uguale a quella della repository quindi fa un controllo di integrità del file.

Viene estratto il `ilpath` dal messaggio e invocata la funzione `int countfiles(char* path)` per contare il numero di file presenti nella cartella e la risposta viene inviata al metadata server tramite la funzione **`int writetosocket(int socket, char* string, int size)`**.

Comando "delete"

Formato comando: delete <username> <filename>

Il comando viene inviato dal metadata server per cancellare i blocchi di un file nella data repository.

Si estraggono il nome utente e il nome del file da cancellare dal messaggio e si crea il path della cartella dove risiedono i blocchi (username/filename/).

Si modifica il file speciale dell'utente (username/username.conf) per eliminare il file dalla lista di tutti i suoi file presenti nel server.

Si crea un comando shell per eliminare la cartella (`rm -r folderpath`).

L'esito dell'apertura del file speciale non viene controllata se è andata a buon fine o meno.

L'esito della funzione `system` non viene controllata se la cartella è stata eliminata con successo o meno.

MD5

`char* md5(char* chunk)`

Funzione che prende come parametro una stringa e ritorna l'hash della stringa come valore di ritorno.

PrintBlock

`int printblock(char* block)`

Funzione che prende come parametro un blocco.

Il blocco viene scompartato per ottenere tutti i valori separati ed assegnati a variabili di tipo compatibile per poter essere stampati.

La funzione stampa caratteri randomici in caso il blocco contenga porzioni di file non testuali.

ReadFromSocket

`char* readfromsocket(int socket, int size)`

La funzione prende in ingresso una socket e una dimensione.

Viene letto dalla socket bytes della dimensione indicata e ritorna un buffer contenente il contenuto letto dal buffer.

In caso di errore si chiude il programma.

RemoveAuthFolder

```
int removeauthfile(char* key)
```

La funzione si occupa di ruovere un file all'interno della cartella specificata nel file di nocnfigurazioni AUTH_FOLDER.

Prende come parametro la chiave, si crea il path ed elimina il file.

In caso di successo ritorna 0 altrimenti 1.

RemoveFile

```
int removefile(char* path)
```

La funzione prende come parametrro il path di un file da eliminare e lo elimina.

In caso di successo ritorna 0 altrimenti 1.

WriteBlock

```
int writeblock(char* block, char* path)
```

La funzione prende in ingresso un blocco e un path dove scrivere i byte della porzione.

La funzione apre il file in modalita lettura, se il file non esiste viene creato e il blocco viene sritto all'interno.

I file vengono creati con i permessi di lettura, scrittura ed esecuzione (0700) da parte del proprietario del file.

WriteToFile

```
int writetofile(char* info, char* path)
```

Funzione che prende come parametro la stringa da scrivere e il path de lfile dove scrivere.

Si apre il file in modalita appen e create e e viene scritto la stringa.

Nel caso l'apertura del file sia andata in errore viene mostrato un messaggio e viene ritornato il valore 1 altrimenti 0.

WriteToSocket

```
int writetosocket(int socket, char* string, int size)
```

Funzione che prende in ingressouna socket, una stringa si bytes e una dimensione.

La funzione invia alla socket dedicata la stringa di byte della dimensione specificata.

Nel caso la dimensione sia diversa da quella della streinga si potrebbe inviare meno bytes o sfiorare la stringa ed inviare piu bytes del dovuto.

In caso di errore si chiude il client.

Note

Molti controlli sono fatti a monte dell'esistenza di file e cartelle (non tutte), quindi durante l'apertura di un file o cartella l'esistenza di esse è certa.

Molti free dei buffer non sono stati eseguiti cosa che potrebbe causare problemi alla normale esecuzione del programma.