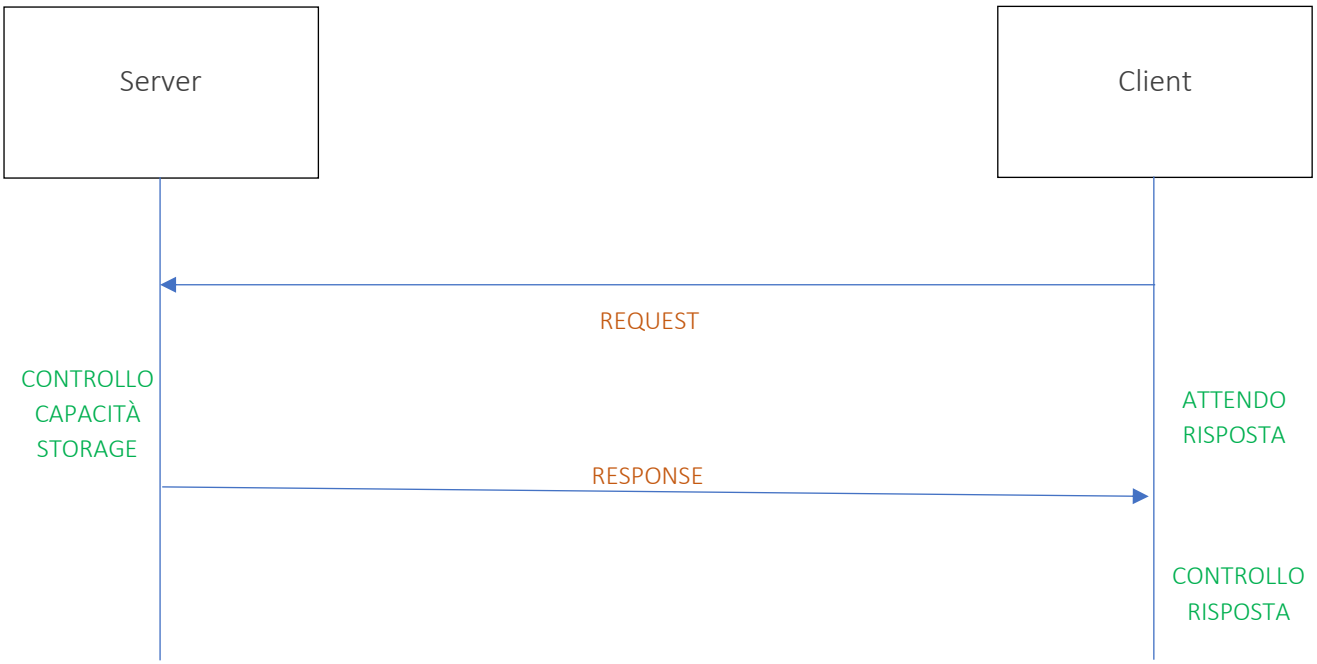
RELAZIONE PROGETTO SOL – Benedetti Gabriele | Matr. 602202

GitHub Repo: https://github.com/gbenedetti22/Progetto-SOL-2021

INTRODUZIONE

La specifica del progetto richiede di implementare un File Storage simil Drive, dove il Server fa da storage esterno e il Client invia i file da memorizzare al Server. L'unica differenza con un Drive classico è che il Server non si appoggia alla memoria di massa per memorizzare i file, bensì alla memoria centrale (RAM).

PROGETTAZIONE



REQUEST corrisponde ad una stringa, la cui sintassi è del tipo:

[*op*] : [file path] : [client PID] ? [option]

- ***op*** : operazione richiesta (Write, Read, Create ecc....)
- **file path** : il path del file da scrivere/leggere
- **client pid** : il PID univoco del Client, utilizzato dal Server per identificare quello specifico Client
- **option** : può essere:
    - 'y' per richiedere al Server di inviare i file espulsi in caso di Capacity Missess
    - 'n' per far espellere al Server i file senza inviarli al Client

RESPONSE corrisponde ad un intero che notifica al Client l'avvenuto successo o fallimento dell'operazione richiesta.

In caso di successo, RESPONSE corrisponderà alla macro S_SUCCESS. In caso di fallimento, invece, essa corrisponderà ad una macro che identifica il relativo errore. Tutte le macro sono segnate nel file "myerrno.h", situato nella cartella "*lib*" del progetto.

SCELTE EFFETTUATE

1) Se un file è aperto **non** può essere cancellato. Questo significa che in caso di CAPACITY MISSES, vengono tolti dalla coda **SOLO** i file non aperti (vedi file FIFOtest_algorithm.c per un test più esaustivo)
2) Se il Server non può più memorizzare file perché la capienza massima è stata raggiunta, e un Client tenta di creare un file (utilizzando la funzione "*openFile(file, O_CREATE))*", allora viene avviata un'operazione di pulitura da parte del Server, utilizzando sempre una politica FIFO.
   I nomi dei file espulsi vengono inviati al Client, il quale li stamperà sullo standard.
3) Se il Client termina la connessione con il Server senza chiudere alcuni file, allora il Server chiuderà per lui tutti i file non chiusi. Questo serve a ridurre la lunghezza delle liste all'interno dei file (vedi sotto).
4) Gli errori vengono stampati anche senza l'opzione "-p". Questo per dare modo all'utente di capire se le operazioni che ha richiesto sono andate a buon fine, senza avere un output pieno di messaggi.

STRUTTURE DATI, FILE E PARSER

Per la memorizzazione dei file viene usata una hash table, la cui grandezza è stabilita in base al numero di file che si vuole memorizzare. La tabella viene allocata tutta e subito per una questione di complessità: dover ricalcolare l hash e spostare tutti i file è un'operazione dispendiosa, soprattutto se si hanno tanti file. Optando per questa scelta c'è il rischio che si allochi per n file, dove n è un numero molto grande, e che poi si memorizzino solo alcuni file. Ovviamente l'utente può scegliere quanto deve essere grande la tabella, quindi se si sa che più o meno più di tot file non si vuole memorizzare, la tabella non sarà mai grande inutilmente.

Il Server utilizza inoltre un'altra hash table per tenere traccia di quanti file un determinato Client ha aperto. Questo è utile alla fine, quando il Client si disconnette, per vedere se ci sono dei file da chiudere oppure no.

hash_table storage	->	key: *filepath*	|	value: *files_s*
hash_table opened_files	->	key: *cpid*	|	value: *(*intero che identifica quanti file cpid ha aperto*)*

I file sono memorizzati con una struct *file_s*, la quale contiene le seguenti informazioni:

```
struct {
   char *path;      ->    path del file
   void *content;   ->    contenuto del file
   size_t size;     ->    grandezza in byte del file
   list *pidlist;   ->    lista dei client che hanno aperto questo file
} file_s;
```

Alla fine, il Server controlla se ci sono dei file da chiudere per ridurre la lunghezza di *pidlist*. Sapendo a prescindere, tramite la seconda hash_table, se un Client ha chiuso tutti i file oppure no, è possibile evitarsi di scorrere successivamente più e più volte elementi della lista non necessari, cancellandoli semplicemente.

Il file *config* può avere qualsiasi estensione e può contenere i seguenti campi:

N_THREAD_WORKERS	Stabilisce quanti thread Workers avviare
MAX_STORAGE_SPACE	Definisce la dimensione massima dello Storage Server
MAX_STORABLE_FILES	Definisce il numero massimo di file memorizzabili
SOCK_PATH	Stabilisce il file Socket a cui i Client devono connettersi
PRINT_LOG	0 -> il Server non stampa nulla 1 -> valore di default. Il Server stampa solo alcune operazioni 2 -> il Server stampa in output tutto quello che succede

Inoltre, gode delle seguenti proprietà:

1) Le linee vuote vengono ignorate
2) Tutto ciò che comincia con # viene considerato come commento
3) Tutti i campi possono essere omessi, in quanto hanno di base dei valori di DEFAULT
4) L'assegnamento di un campo è dato da "<campo>=<valore>". Il numero indefinito di spazi vuoti tra l'uguale e il campo/valore è consentito, in quanto vengono ignorati (quindi è possibile scrivere tipo "<campo> = <valore>")
5) Se si assegna più di una volta un campo, viene utilizzato l'ultimo valore assegnato
6) Il non rispetto di queste proprietà, può generare errore o undefined

NOTA: Il *config* parser **DEVE** essere inizializzato o con la macro DEFAULT_SETTINGS o con la funzione "settings_default(settings* s)" dove settings è una *struct* contenente tutti i parametri sopra descritti.
Un esempio di file config è memorizzato nella cartella "*config*" del progetto (il nome del file è *config.ini*)