Reti 1

Relazione del Progetto di Laboratorio - Anno 2019/2020

Alessandro Borsoi 20014853

Compilazione e avvio

La compilazione avviene tramite `make`. Per compilare è possibile usare il comando `make all` mentre `make clean` elimina tutti gli artefatti di compilazione. L'applicazione server e l'applicazione client risiedono nella directory `apps` per cui, dopo aver compilato, è sufficiente eseguire dalla radice del progetto i due programmi tramite `./apps/server <numero_porta>` e `./apps/client <indirizzo_server> <numero_porta>` come indicato nelle specifiche.

Sono stati creati anche degli unit test, presenti nella directory `test`, che è possibile avviare con `./test/test_<nome_componente>`. Il comando `make all` compila in automatico anche loro.

Il progetto è stato sviluppato quasi interamente su un MacBook Pro con `gcc` e i flag di compilazione `-Wall -Wextra -std=c11 -pedantic` ma la correttezza e assenza di warning è stata verificata anche su una macchina (Arch) Linux. Inoltre è stata fatta una ulteriore verifica di correttezza usando il compilatore `clang`. In aggiunta è stato usato sulle due macchine il programma `valgrind` con i flag `--tool=memcheck --leak-check=full` su tutti gli eseguibili del progetto (client, server e test) per verificare l'assenza di memory leaks o in generale problemi legati alla gestione della memoria.

Struttura del progetto

Il progetto si compone di diverse directory, lasciando nella radice solo il Makefile generale. Come già accennato, `apps` contiene le due applicazioni client e server vere e proprie. Queste dipendono da librerie il cui header pubblico è contenuto nella cartella `include` (importata in fase di compilazione). All'interno di `include` si distinguono i componenti utilizzati dal client e dal server con un unico header comune `protocol.h` che definisce la dimensione dei messaggi del protocollo e gli stati possibili. Il codice sorgente di questi header è contenuto in `src`, compilato come libreria in `bin` e linkato staticamente. È stata fatta questa scelta per esporre una interfaccia facilmente testabile dei componenti di utilità per i due programmi principali. Questi componenti sono:

- `store` usato dal server, astrae un generico repository per i numeri in ingresso ed espone il calcolo di media e varianza;
- `protocol` implemeta di fatto le logiche di ricevimento e risposta lato server, con lo store come dipendenza;
- `splitter` usato dal client, implementa la logica di partizionamento in più messaggi di una sequenza numerica (anche molto lunga) inserita dall'utente tramite file di testo.

Tutte e tre le interfaccie espongono le funzione pubbliche usando `upo_` come prefisso per evitare eventuali collisioni di nomi.

Nella directory `test` ci sono gli unit test dei tre componenti appena menzionati. Infine in `data` ci sono dei file di testo usati sia negli unit test che nei test manuali delle due applicazioni.

Client

Per il client si è scelta una logica di inserimento dati da file di testo per facilitare l'inserimento di molti dati contemporaneamente. Il file di testo deve essere composto da una sequenza di numeri interi positivi

separata da spazi. Il controllo utilizza la funzione `isspace` di `ctype.h` per cui sono validi tutti i caratteri di separazione validati dalla funzione (`' ', '\n', '\t', '\v', '\f', '\r'`). All'avvio del programma client viene richiesto il path del file di testo (assoluto o relativo alla `pwd` dell'utente). Nella cartella `data` ci sono alcuni file che possono essere usati come input di prova. Il client continuerà a chiedere un file valido (sia che non sia stato trovato o che contenga caratteri non validi) fino a quando non si decide eventualmente di uscire premendo `q`. All'inserimento di un file valido, il client si occuperà di mandare la sequenza di numeri al server, opportunamente segmentata nel caso ce ne sia bisogno. La gestione dei messaggi è quella richiesta da specifica.

A livello di codice, dopo una fase di setup della connessione, viene passato l'hadler della socket alla funzione `void program(int socket)` che si occuperà di gestire l'intero processo di input e invio dei dati e dei possibili errori. All'inizio ci si aspetta un messaggio di `OK_START` dal server come previsto dal protocollo. La funzione `upo_protocol_response_t parse(char *input)` si occupa di determinare il dipo di messaggio che arriva dal server. Una volta validato (e stampato a video), si entra in un loop (definito nella `bool client_input(upo_protocol_splitter_t *splitter)`) che si occupa di gestire l'input da parte dell'utente. Il client utilizza la libreria `splitter` per gestire la validazione, il parsing e la segmentazione dei numeri da inviare al server nel rispetto delle regole di protocollo definite. In `./test/test_splitter` sono definiti gli unit test per questo componente usando i file presenti in `data`. Una volta ottenuto un file valido, si entra in un loop di invio dati che continua fin tanto che la risposta da parte del server è `OK_DATA`. la funzione `size_t upo_protocol_splitter_next(upo_protocol_splitter_t splitter, char *output, size_t output_size)` si occupa di scrivere in output il messaggio da mandare al server (finendo con `"0\n"`), e di ritornare il numero di dati inviato come controllo rispetto alla risposta del server. Di nuovo viene letta la risposta, parsato il tipo ed eseguiti i controlli del caso nel rispetto delle specifiche. In caso di corretto invio o di errore, si esce dal programma distruggendo le strutture dati precedentemente create.

Server

Come per il client, la logica di risposta implementata nel server risiede interamente nella funzione `void program(int socket)` che riceve in input la socket ed entra in loop per soddisfare la richiesta fino a quando un messaggio di terminazione non è stato prodotto. La funzione è interamente listata qui:

```c
void program(int socket)
{
    upo_store_t store = upo_store_create();
    char input[UPO_PROTOCOL_MAX + 1];
    char output[UPO_PROTOCOL_MAX];

    write(socket, WELCOME, sizeof(WELCOME));
    while (1)
    {
        memset(input, '\0', UPO_PROTOCOL_MAX + 1);
        memset(output, '\0', UPO_PROTOCOL_MAX);
        read(socket, input, UPO_PROTOCOL_MAX);
        upo_protocol_response_t response = upo_protocol(store, input, output);
        write(socket, output, UPO_PROTOCOL_MAX);
        if (response != OK_DATA)
            break;
    }
    close(socket);
    upo_store_destroy(&store);
}
```

Dopo la definizione delle strutture necessarie (lo store per il calcolo, e gli array per l'input dal client e l'output del server), viene stampato il messaggio di benvenuto. Si entra poi nel loop che continua solo nel caso di `OK_DATA`. In `upo_protocol` è gestita la logica del protocollo dove, dato in ingresso lo store e il messaggio del

client, viene scritto il messaggio in uscita e ritornato il tipo di risposta per permettere il controllo dello stato. Uscendo dal ciclo vengono pulite le strutture non più necessarie e ci si rimette in attessa di un'altra richiesta.