

# 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 di 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** implementa 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 interfacce espongono le funzioni 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

## 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:

```
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 attesa di un'altra richiesta.