



**DIPARTIMENTO D'INGEGNERIA ELETTRICA E DELLE  
TECNOLOGIE DELL'INFORMAZIONE**

**CORSO DI LAUREA IN INFORMATICA**

***Progetto Lab. Di Sistemi Operativi  
“Gara di Aritmetica”  
Anno Accademico 2020/2021***

***Docente:***

**Prof. Marco Faella**

**Gruppo del Corso: I**

***Alunno:***

**Giuliano Galloppi**

**Gruppo 4**

# INDICE

## **1. Traccia progetto “Gara di Aritmetica”**

- 1.1. Descrizione sintetica
- 1.2. Descrizione dettagliata
- 1.3. Regole generali

## **2. Guida alla compilazione del server e del client**

- 2.1. Compilazione server
- 2.2. Compilazione client

## **3. Guida all’uso del server e del client**

- 3.1. Uso del server
- 3.2. Uso del client

## **4. Protocollo di comunicazione client-server**

## **5. Dettagli implementativi**

- 5.1. Collezione di domande: file *‘domande.txt’*
- 5.2. Gestione delle domande: *‘domande.h’* – *‘domande.c’*
- 5.3. Gestione della classifica: *‘classifica.h’* – *‘classifica.c’*
- 5.4. Server: *‘main.c’*
- 5.5. Client: *‘main.c’*

## 1. Traccia progetto “Gara di Aritmetica”:

### 1.1.Descrizione sintetica:

Realizzare un sistema client-server che consenta ai client di sfidarsi in una gara di aritmetica.

### 1.2.Descrizione dettagliata:

Il server propone a tutti i client connessi una sequenza di domande di aritmetica come la seguente:

Quanto fa  $13 * 3 - 4 + 68$ ?

- a. 113
- b. 103
- c. 58
- d. -10

I client hanno un minuto di tempo per rispondere a ciascuna domanda. Il server tiene traccia del tempo impiegato da ciascun client a rispondere e usa questa informazione per mostrare ai client una classifica dei giocatori. In particolare, il posto di un giocatore nella classifica dipende dal *tempo medio* impiegato da quel giocatore per rispondere alle domande.

Nuovi client si possono collegare in qualsiasi momento ed unirsi alla partita.

Il server dispone di una collezione fissa di domande e risposte, memorizzate in un file di testo (quindi, deve essere possibile modificare o aggiungere domande senza ricompilare il server).

### 1.3.Regole generali.

Il server va realizzato in linguaggio C su piattaforma UNIX/Linux. Oltre a alle system call UNIX, il server può utilizzare solo la libreria standard del C. Il server deve essere di tipo concorrente, ed in grado di gestire un numero arbitrario di client contemporaneamente. Il server effettua il log delle principali operazioni (nuove connessioni, disconnessioni, richieste da parte dei client) su standard output.

Il client va realizzato in linguaggio Java su piattaforma Android. Client e server devono comunicare tramite socket TCP o UDP. Per la realizzazione del client, in particolare per la comunicazione, è consentito esclusivamente l'utilizzo delle API standard (java.net.\*)

- **Nota:** a differenza della traccia, il gruppo appartenente al vecchio ordinamento ha realizzato sia il server che il client in linguaggio C su piattaforma UNIX/Linux, come accordato col docente.

## 2. Guida alla compilazione server e client

### 2.1. Compilazione server

Per compilare il codice sorgente del server, da terminale posizionarsi nella directory *'server'* contenenti i file *main.c* , *classifica.c* , *classifica.h* , *domande.c* , *domande.h* ed inviare il comando:

```
gcc -o s.out main.c classifica.c classifica.h domande.c domande.h -lpthread
```

### 2.2. Compilazione client

Per compilare il codice sorgente del client, da terminale posizionarsi nella directory *'client'* contenente il file *main.c* ed inviare il comando:

```
gcc -o c.out main.c
```

## 3. Uso del Server e del client

### 3.1. Uso del Server

Prima dell'esecuzione del server bisogna accertarsi che nella stessa cartella del file eseguibile *'s.out'* si trovi il file *'domande.txt'* che contiene la collezione di domande che sarà elaborata dal server per essere inviata ai client. (Consultare i dettagli implementativi per il formato di esse.)

Il file eseguibile *'s.out'* del server può essere eseguito da terminale inviando il comando:

```
./s.out [Porta TCP]
```

L'argomento *[Porta TCP]* indica la porta sulla quale il server sarà in ascolto.

E' possibile scegliere un valore compreso tra 5000 e 32768 siccome queste sono porte riservate agli utenti, ma restano consentiti valori da 0 a 65535.

L'argomento può essere omissso, in tal caso sarà impostata come porta di default la 7295.

All'esecuzione inserire 0 per avviare il server.

```
giulianogalloppi@asus-s500ca:~/server$ ./s.out 7295
GIULIANO GALLOPPI N86001508 : Progetto di LSO lato server A.A. 2020/21 - Gara di Aritmetica
Gara di Aritmetica : Server.
Numero di domande nel file domande.txt = 5
0 per avviare il server, qualsiasi altro numero per uscire.
0
Server avviato.
```

Il server non necessita di essere riavviato per resettare una partita, questo avverrà in automatico. Per interromperne l'esecuzione è necessario inviare un SIGINT con CTRL+C.

## 3.2. Uso del Client

Il file eseguibile 'c.out' del client può essere eseguito da terminale inviando il comando:

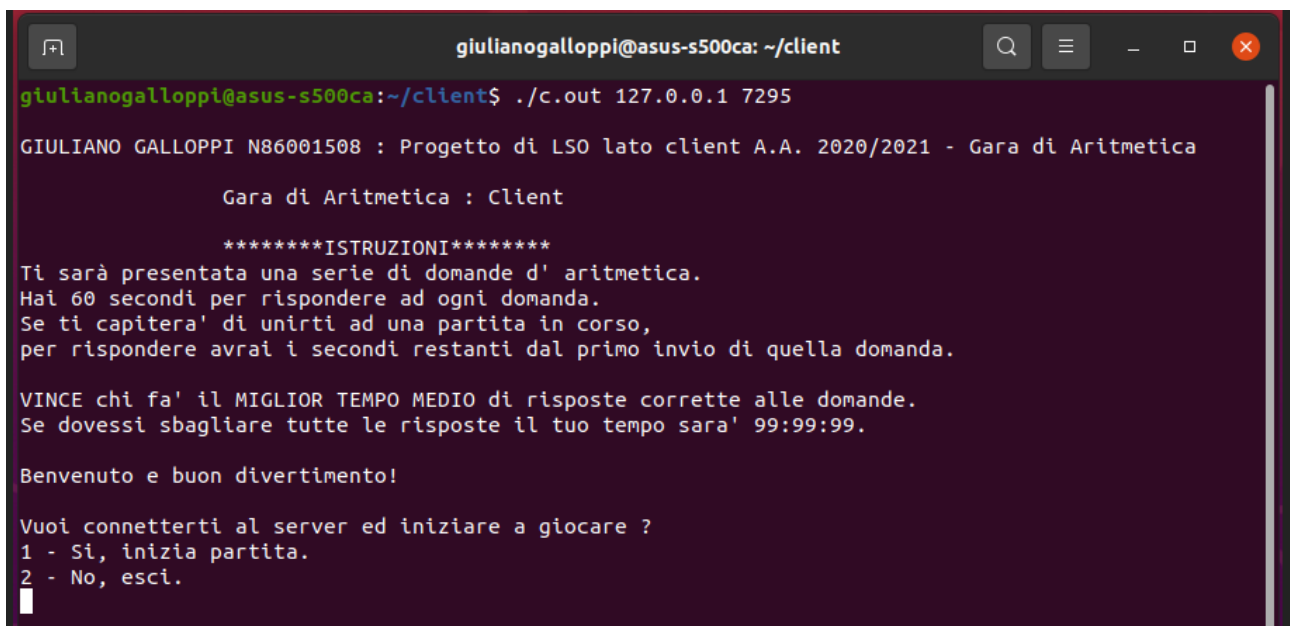
```
./c.out [Indirizzo IP] [Porta TCP]
```

L'argomento [Indirizzo IP] deve essere l'indirizzo IP dove risiede il server e [Porta TCP] deve essere il numero di porta dove il server è in ascolto.

- *Nota:* [Indirizzo IP] può assumere anche la stringa 'localhost', equivalente a 127.0.0.1, nel caso in cui questi vengano eseguiti sulla stessa macchina.

Gli argomenti possono essere omessi, in tal caso sarà impostato di default IP: 127.0.0.1 e porta 7295.

Se il server è avviato, eseguendo il client nella schermata principale saranno presentate le istruzioni per la 'Gara di Aritmetica' ed inserendo 1 ci collegheremo al server per iniziare la partita.



```
giulianogalloppi@asus-s500ca: ~/client
giulianogalloppi@asus-s500ca:~/client$ ./c.out 127.0.0.1 7295
GIULIANO GALLOPPI N86001508 : Progetto di LSO lato client A.A. 2020/2021 - Gara di Aritmetica

Gara di Aritmetica : Client

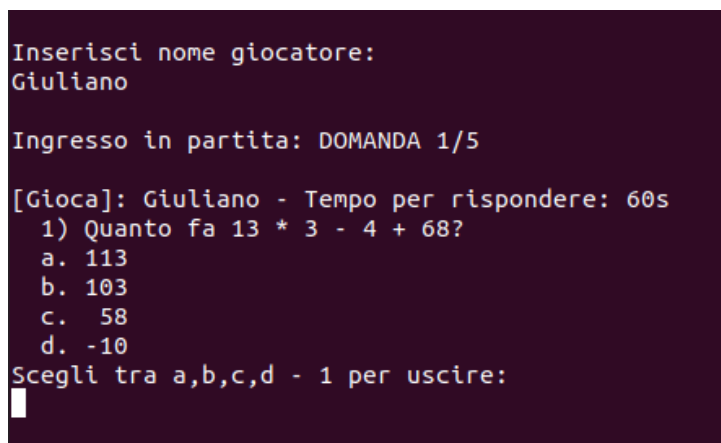
*****ISTRUZIONI*****
Ti sarà presentata una serie di domande d' aritmetica.
Hai 60 secondi per rispondere ad ogni domanda.
Se ti capiterà di unirti ad una partita in corso,
per rispondere avrai i secondi restanti dal primo invio di quella domanda.

VINCE chi fa' il MIGLIOR TEMPO MEDIO di risposte corrette alle domande.
Se dovessi sbagliare tutte le risposte il tuo tempo sarà 99:99:99.

Benvenuto e buon divertimento!

Vuoi connetterti al server ed iniziare a giocare ?
1 - Sì, inizia partita.
2 - No, esci.
█
```

Successivamente al client sarà chiesto d'inserire il nome giocatore ed inizierà a ricevere la sequenza di domande della partita a cui si sarà unito, sia questa nuova o già esistente.



```
Inserisci nome giocatore:
Giuliano

Ingresso in partita: DOMANDA 1/5

[Gioca]: Giuliano - Tempo per rispondere: 60s
1) Quanto fa 13 * 3 - 4 + 68?
a. 113
b. 103
c. 58
d. -10
Scegli tra a,b,c,d - 1 per uscire:
█
```

Un client può unirsi ad una partita in qualsiasi momento, quando accade esso riceverà il tempo restante dal primo invio di quella domanda, per rispondere.

```
Ingresso in partita: DOMANDA 2/5
Ti sei unito ad una partita gia' iniziata.

[Gioca]: Luca - Tempo per rispondere: 19s
2) Quanto fa  $189 * 1 - 22 + 1$ ?
a. 33
b. 168
c. -33
d. -10
Scegli tra a,b,c,d - 1 per uscire:
```

Si può rispondere inserendo i caratteri 'a, b, c, d' oppure abbandonare la partita inserendo 1. Se scadono i secondi disponibili per rispondere, il client sarà avvisato e la risposta conterà come sbagliata.

```
[Gioca]: Giuliano - Tempo per rispondere: 60s
1) Quanto fa  $13 * 3 - 4 + 68$ ?
a. 113
b. 103
c. 58
d. -10
Scegli tra a,b,c,d - 1 per uscire:
Timed out.

Risposta SBAGLIATA!
Tutti i giocatori hanno risposto, ecco la prossima domanda.
```

Quando un client risponde, la domanda successiva verrà visualizzata quando tutti i giocatori avranno risposto, ed a fine partita si potrà leggere la classifica per vedere il punteggio in tempo medio.

```
Domande finite! Consulta la classifica per il tuo punteggio!

CLASSIFICA:

[POSIZIONE] [CLIENT ADDRESS] [NOME GIOCATORE] [TEMPO MEDIO]
1           2.0.136.132      Luca           00:00:23
2           2.0.136.128      Giuliano        99:99:99

Giuliano il tuo IP/CLIENT ADDRESS e' : 2.0.136.128
Partita conclusa.
Arrivederci!
```

Il server, nel frattempo, stamperà il log di eventuali connessioni, abbandoni e disconnessioni dei client dalla partita.

```
giulianogalloppi@asus-s500ca:~/server$ ./s.out 7295
GIULIANO GALLOPPI N86001508 : Progetto di LSO lato server A.A. 2020/21 - Gara di Aritmetica
Gara di Aritmetica : Server.
Numero di domande nel file domande.txt = 5
0 per avviare il server, qualsiasi altro numero per uscire.
0
Server avviato.

Nuova connessione dal client d'indirizzo = 2.0.168.220 in data: 15/01/2021 alle ore: 23:30:13

Client 2.0.168.220 ha abbandonato la partita
Il client d'indirizzo : 2.0.168.220 si e' sconnesso dal server in data: 15/01/2021 alle ore: 23:30:37
```

## 4. Protocollo di comunicazione client-server

La comunicazione tra client e server avviene tramite *'socket'*, un'interfaccia che consente la comunicazione interprocesso sia localmente che tramite un collegamento in rete.


Il protocollo scelto per la comunicazione è il protocollo internet *TCP/IP*.

A livello di applicazione il server ha il compito di scrivere sulla socket la domanda ed il tempo per rispondere alla domanda, questi saranno letti dai vari client che successivamente scriveranno sulla socket la risposta, a ricezione di essa il server invierà messaggi di risposta corretta o errata letti dai client. A fine partita si occuperà di inviare la classifica dei giocatori ed indirizzo IP per riconoscersi in essa, questi letti da ogni client.

I dati sono scambiati in formato testuale, quindi sono utilizzate delle stringhe. Questo vale anche per i dati numerici i quali subiscono delle conversioni da testo a numerico e viceversa.

```
int tempoPerRispondere = currentTempoTimer;
sprintf(inviaCurrentTime, "%d", currentTempoTimer);
pthread_mutex_unlock(&semaforoCurrentTime);

send(sd, inviaCurrentTime, 20, 0);
```



```
recv(sd, tempoDelServer, 20, 0);
tempoDelServerInt = atoi(tempoDelServer);
```

Tutti i messaggi scambiati sulla socket sono a *lunghezza fissa*. La dimensione del testo della domanda, così come le risposte e messaggi generici di avviso hanno una dimensione fissa di 100, il tempo disponibile per la risposta e la risposta del client hanno una dimensione fissa di 20. La classifica ha una dimensione fissa di 4096 e l'IP visualizzato dal client 16.

## 5. Dettagli implementativi

### 5.1. Collezione di domande: file *'domande.txt'*

Il file *'domande.txt'* contiene la sequenza di domande che saranno inviate ai client. Si possono aggiungere o modificare domande senza ricompilare il server.

Il max di caratteri consentiti per il testo della domanda e delle risposte è di 100 caratteri.

Il formato corretto di una domanda è il seguente:

- Riga 1: Testo domanda
- Riga 2: risposta a
- Riga 3: risposta b
- Riga 4: risposta c
- Riga 5: risposta d
- Riga 6: risposta Corretta tra le 4 precedenti
- Riga 7: lasciata in bianco per separare le domande o per delineare la fine della sequenza.

Ad esempio, se avessimo 4 domande e volessimo aggiungerne una quinta, ci troveremo con la posizione del cursore posizionata sulla 35esima riga del file:

```
25 c. 402
26 d. 234
27 c
28
29 5) Quanto fa 10 * 10 - 50 + 100?
30 a. 100
31 b. 0
32 c. -50
33 d. 150
34 d
35 |
```

### 5.2. Gestione delle domande: *'domande.h'* – *'domande.c'*

I file *'domande.c'* e *'domande.h'* si occupano della gestione delle domande.

Queste vengono immagazzinate in una struttura dati *'Domanda'* una alla volta per ogni invio di domanda ai client.

```
struct domanda{ //Struttura che definisce una domanda
    char testo[100];
    char a[100];
    char b[100];
    char c[100];
    char d[100];
    char correctAnswer[100];
};
typedef struct domanda Domanda;
typedef Domanda* DomandaPtr;
```

Per la lettura dal file viene eseguito il metodo *void leggiDomandaDaFile(int fd, DomandaPtr\* dom);* che sfrutta il metodo *void leggiRigaDaFile(int fd, char buffer[]);* che per leggere le righe di ogni domanda utilizza la system call **'read'**.



### 5.3. Gestione della classifica: ‘classifica.h’ – ‘classifica.c’

I file ‘classifica.c’ e ‘classifica.h’ si occupano della gestione della classifica.

Essa è stata strutturata come una Linked List di giocatori (quindi client) con relative informazioni.

```
#define INFINITY_EXAMPLE 86400

struct giocatore{ //Struttura giocatore(client)
    char nomeGiocatore[100];
    char clientAddress[50]; //Sara' univoco quindi usato come 'id' per ogni giocatore
    int tempoTotaleRisposte;//misurato in secondi
    int totRisposte;//misurato in secondi
    int tempoMedio;//misurato in secondi
    int postoClass;
    struct giocatore* nextGiocatore;
};

typedef struct giocatore Giocatore;
typedef Giocatore* ClassificaListPtr; //Classifica e' vista come una lista di giocatori
```

Per ottenere la media del tempo di un client si sono utilizzate funzioni di conversione, che dai secondi totali cumulati dalle risposte date correttamente si ottengono ore, minuti e secondi effettivi. Queste vengono sfruttate dalla funzione *char\* convertTimeToString(int totalSeconds)* che a sua volta è utilizzata dalla funzione *void getSortedClassificaToString(ClassificaListPtr inizioLista, char classificaStr[])*, quest’ultima ha il compito a fine partita di trasformare la classifica in stringa così da poterla inviare ai client sulla socket.

### 5.4. Server: ‘main.c’

Nel main del server si trova la gestione del tempo della partita, dal quale è possibile ricavare il tempo impiegato da un client a rispondere, è affidata alla funzione *void \*threadTimer(void \*arg)* eseguita da un thread che simula un timer. La gestione dei client se ne occupa la funzione *void\* threadServerClient(void \*arg)* a sua volta avviata da thread appositi.

Questi thread sono stati sincronizzati grazie ai ‘**semafori**’ (mutex) ed alle ‘**condition variable**’ le quali hanno permesso di evitare le ‘**race condition**’ sui dati condivisi.

#### • void \*threadTimer(void \*arg)

Utilizza:

```
pthread_t tidTimer; //tid del thread che gestisce il timer
pthread_mutex_t semaforoTimer = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condizioneTimer = PTHREAD_COND_INITIALIZER;
int flagTimer = 0; //FALSO

pthread_mutex_t semaforoCurrentTime = PTHREAD_MUTEX_INITIALIZER; //Mutex del tempo corrente
pthread_cond_t condizioneAggiornamentoTempo = PTHREAD_COND_INITIALIZER; //Condizione sul tempo corrente
int tempoAggiornato = 1; //Condizione di aggiornamento del tempo corrente per ogni invio di domanda
int currentTempoTimer = 60; //Var globale del tempo corrente usata dai thread che gestiscono il client
```

E’ la funzione di avvio da dare in ingresso al thread che funge da timer. Questo sarà avviato dal primo client che si collega ed inizia a giocare, fino a quel momento il thread attenderà sulla condition variable ‘**condizioneTimer**’ associata al mutex ‘**semaforoTimer**’. Avviato il timer contestualmente all’invio della prima domanda, o ad ogni nuova domanda, avverrà un timestamp col quale si effettuerà il controllo dei secondi passati.

Il mutex ‘**semaforoCurrentTime**’, la condition variable ‘**condizioneAggiornamentoTempo**’ e la risorsa condivisa ‘tempoAggiornato’ fanno in modo di sincronizzare i client col timer affinché ad ogni invio di domanda sia sicuro che questi ricevano 60 sec a disposizione per rispondere, (tranne quando questi si aggiungono ad partita con una domanda già avviata) e proteggendo la variabile ‘currentTempoTimer’ che permette di ricavare il tempo di risposta di un client.

- **void\* threadServerClient(void \*arg)**

È la funzione d' avvio data in ingresso ai thread che si occupano della gestione di un client quando si connette al server e della sincronizzazione con gli altri già connessi ed il thread timer. Quindi effettua l'invio della domanda, ricezione della risposta, aggiornamento ed invio classifica del client, la quale essendo condivisa tra tutti i client i suoi accessi sono protetti dal mutex

**'semaforoClassifica'**. Le funzioni di invio, e quindi scrittura sulla socket, vengono effettuate tramite la system call **'send'** e quelle di ricezione, quindi lettura dalla socket, con la system call **'recv'**. La sincronizzazione con gli altri thread che gestiscono i client in partita è garantita dai mutex **'semaforoStannoRispondendo'** , **'semaforoProssimaDomanda'** e la condition variable **'condizioneProssimaDomanda'** e la variabile globale **'stannoRispondendo'**.

I client si sincronizzano sulla domanda, ovvero, dall'ingresso in partita quando un client risponde questo aspetterà che tutti i client abbiano risposto se c'è qualcuno che sta ancora rispondendo, pertanto il thread in tale caso va in uno stato di wait sulla **'condizioneProssimaDomanda'**.

L'ultimo client rimasto in partita che risponderà, (quindi stava ancora rispondendo alla domanda), sia che inserisca una risposta o esca dalla partita o perché abbia terminato il tempo a disposizione, farà in modo che venga estratta la domanda successiva, il timer sarà bloccato per poi resettarsi e risveglierà tutti i client in attesa sulla condition variable in modo che tutti possano ripartire insieme e ricevere contemporaneamente la domanda successiva.

La sincronizzazione col timer, come già detto, avviene sulla variabile **'tempoAggiornato'** che blocca qualsiasi thread se prima il timer non resetta il tempo di risposta alla domanda (*variabile currentTempoTimer*) garantendo anche qui che non ci siano **'race condition'**.

Quando le domande saranno terminate sarà poi inviata la stringa al client contenente la classifica e la stringa contenente il suo indirizzo IP.

Il metodo *void checkAbbandonoClient(int sd, char rispostaClient[], char clientAddressString[], int count, int fineDom)* controlla se la risposta data dal client equivale a voler uscire dalla partita, nelle occasioni in cui capita che è l'ultimo client ad abbandonare la partita, esso agisce allo stesso modo in cui avrebbe agito rispondendo normalmente, permettendo il proseguimento della partita correttamente.

## **5.5. Client: 'main.c'**

Il client riceve la domanda, classifica, messaggi riguardo la correttezza o meno delle risposte date ed invia la risposta alla domanda sulla socket, esso collabora col server riguardo la scadenza del tempo a disposizione per rispondere ad una domanda, implementato grazie alla system call **'select'**.

Il metodo *void letturaConSceltaSelect(int sd, int sec, fd\_set\* readfds, char rispostaC[])* utilizza la system call select per monitorare lo standard input (STDIN\_FILENO) , se questa ritorna e lo standard input è pronto (verificato con la funzione FD\_ISSET) allora vuol dire che è avvenuto un evento sullo standard input, quindi il metodo memorizzerà la risposta inserita dal client, (sempre che questa sia un carattere valido per la risposta) altrimenti se il valore di **'sec'** che corrisponde al timeout dato alla select scadrà, allora la system call ritornerà e sarà visualizzato un messaggio di timeout mentre nella risposta verrà memorizzato uno **'0'**, che in questo caso indica l'avvenimento del timeout ed una volta inviata la risposta al server essa sarà valutata certamente sbagliata.

Sono state gestite anche le chiusure anomale del client generate dai segnali inviati da CTRL+X , CTRL+Z ed il segnale SIGPIPE utilizzando la system call *signal* che richiama **uscitaHandler** chiudendo il client, di conseguenza si è preferito scegliere anche l'arresto del server e degli altri client in cascata siccome in tale caso la partita risulterebbe poi falsata.