Relazione progetto del corso Reti di Calcolatori

Nicolò Colella
Matricola: 0124/2273
Docente: Emanuel Di Nardo

Anno Accademico: 2023 - 2024

Contents

1 Descrizione del progetto

In questo paragrafo, vedremo la traccia che è stata assegnata e le note di sviluppo.

1.1 Traccia - Pub

Scrivere un'applicazione client/server parallelo in cui viene effettuata la gestione di un pub.
Gruppo 1 studente:
C'è un solo cliente per tavolo.
Pub:

- Accetta un cliente se ci sono posti disponibili;
- Prepara gli ordini.

Cameriere:

- Chiede al pub se ci sono posti disponibili per un cliente;
- Prende le ordinazioni e le invia al pub.

Cliente:

- Chiede se può entrare nel pub;
- Richiede al camerire il menù;
- Effettua un ordine.

Gruppo 2 studenti:
Simulare la possibilità di avere più clienti al tavolo, ogni cliente è un nuovo client che si siede allo stesso tavolo (utilizzare un numero di tavolo). Ogni client effettua un ordine separato.
Gruppo 3 studenti:
Il cameriere invia l'ordinazione al pub solo quando tutti i clienti ad uno stesso tavolo hanno ordinato. Il pub notifica l'avvenuta preparazione dei piatti solo quando tutti i piatti sono pronti (utilizzare un tempo random per la preparazione di più piatti).

1.2 Note di sviluppo

La prova d'esame richiede la progettazione e lo sviluppo della traccia proposta. Il progetto deve essere sviluppato secondo le seguenti linee:

- utilizzare un linguaggio di programmazione a scelta (C, Java, Python, etc...);
- utilizzare una piattaforma Unix-like;
- utilizzare le socket;
- inserire sufficienti commenti.

2 Descrizione e schema dell'architettura

In questo paragrafo, verranno descritti gli strumenti con cui è stato realizzato il progetto, soffermandoci su descrizione e schema dell'architettura.

2.1 Strumenti di sviluppo

La traccia proposta è stata sviluppata utilizzando il linguaggio di programmazione C e le socket su piattaforma Unix-like, proprio come richiesto.

2.2 Descrizione dell'architettura

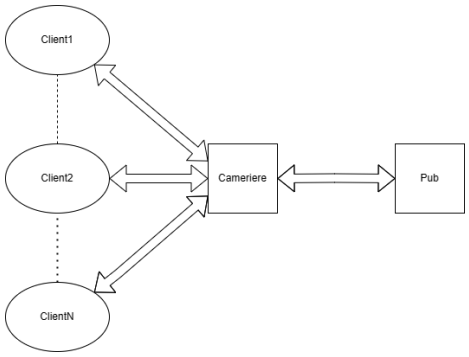
Il sistema utilizza un'architettura **client-server** in cui le tre componenti principali comunicano tra di loro utilizzando **socket TCP/IP** per la trasmissione affidabile ed efficiente dei dati.
Le tre componenti principali sono:

- **Pub**: rappresenta il server principale a cui si connettono i Camerieri. Gestisce i tavoli nel pub e la preparazione degli ordini da consegnare ai clienti.
- **Cameriere**: funge da intermediario tra il Pub e il Client.
  Infatti, agisce come client nei confronti del Pub, inoltrando le richieste di ingresso dei Client e gli ordini da questi effettuati, e come server nei confronti dei Client, facendoli accomodare al primo tavolo disponibile e servendogli il menù.
- **Client**: rappresenta il cliente vero e proprio del pub che si connette al Cameriere per effettuare il check-in, ricevere il menù ed effettuare l'ordine.

N.B. Sia **Pub** che **Cameriere** sono stati implementati come **server concorrenti** utilizzando **fork**.

2.3 Schema dell'architettura

In seguito, una figura che rappresenta lo schema di quanto appena descritto:



3 Dettagli implementativi e parti del codice sviluppato

In questo paragrafo, verranno descritte le librerie utilizzate per snellire il codice e i dettagli implementativi dei client/server con parte del codice sviluppato.

3.1 Libreria "pub_utils.h"

La libreria **pub_utils.h** fornisce un insieme di funzioni e definizioni utili per la gestione delle operazioni all'interno del pub.
Oltre a ciò, la libreria utilizza una struttura **SharedData** per gestire le informazioni sui tavoli occupati nel pub utilizzando la memoria condivisa.
In particolare, all'interno vi è un array di flag **tavoli_occupati** che servirà a distinguere i tavoli disponibili da quelli non disponibili. Di seguito una breve descrizione di ciascuna funzione:

- **is_valid_number(const char *str, int min, int max)**: verifica se una stringa rappresenta un numero intero compreso tra i valori specificati min e max. Restituisce 1 se il numero è valido, altrimenti restituisce 0. Risulterà utile quando un cliente dovrà ordinare.
- **invia_menu(int client_sock, char *message, int tavolo_assegnato)**: fa accomodare il cliente al tavolo assegnato dal pub e gli serve il menù da cui si potrà effettuare l'ordine.
- **prepara_ordine(int tavolo_assegnato)**: simula la preparazione dell'ordine per il tavolo specificato. Stampa un messaggio di notifica durante la preparazione e quando l'ordine è pronto per essere ritirato dal cameriere.
- **assegna_tavolo(SharedData *shared_data)**: cerca e assegna un tavolo disponibile restituendo il numero del tavolo assegnato. Se non ci sono tavoli disponibili, restituisce -1.
- **libera_tavolo(int tavolo_liberato, SharedData *shared_data)**: libera il tavolo specificato, impostando il relativo flag nel vettore tavoli_occupati a 0.

Le costanti definite sono:

- **SHM_KEY**: chiave per l'identificazione della memoria condivisa.
- **NUMERO_TAVOLI**: numero totale di tavoli nel pub.
- **MESSAGE_SIZE**: dimensione allocata per i messaggi tra i processi.

3.2 Libreria "socket_utils.h" e gestione dei segnali SIGNIT

La libreria **socket_utils.h** è stata scritta per incapsulare tutta una serie di funzioni e di definizioni utili per la gestione dei socket e dei segnali SIGNIT. Questi ultimi, quando si verificano, vengono gestiti inviando specifici messaggi '**terminate**' agli altri processi i quali, una volta interpretato il messaggio grazie alle funzioni handler della libreria, reagiscono di conseguenza. Di seguito una breve descrizione di ciascuna funzione:

- **create_socket()**:crea un nuovo socket e restituisce il file descriptor associato ad esso.
- **set_socket_option(int sock)**: imposta l'opzione SO_REUSEADDR sul socket specificato per consentire il riutilizzo dell'indirizzo e della porta.
- **bind_socket(int sock, const char *ip, int port)**: associa il socket all'indirizzo IP e alla porta specificati.
- **listen_socket(int sock)**: mette il socket in ascolto per le connessioni in arrivo. La lunghezza della coda di connessioni in attesa è impostata a 5.
- **accept_connection(int sock, struct sockaddr_in *client_addr)**: accetta una connessione in arrivo sul socket specificato e restituisce il file descriptor associato alla nuova connessione.
- **connect_to_address(const char *ip, int port)**: stabilisce una connessione a un indirizzo IP e porta specificati e restituisce il file descriptor associato alla connessione.
- **receive_message(int sock, char *message, size_t message_size)**: riceve un messaggio dal socket specificato e lo memorizza nella variabile message. La dimensione massima del messaggio è specificata da message_size. In caso di errore durante la ricezione, la funzione stampa un messaggio di errore, chiude il socket e termina il programma.
- **handle_terminate_generic(int sock, char *message)**: gestisce messaggi di tipo "pub_terminate", "cameriere_terminate" o "client_terminate" inviati dalle funzioni **signit_handler** definite in ciascuno dei codici principali, stampando un messaggio appropriato e chiudendo il socket specificato. Viene principalmente impiegata dai clienti e dal pub.
- **handle_terminate_for_cameriere(int pub_sock, int client_sock, char *message)**: gestisce messaggi di tipo "client_terminate" o "pub_terminate" specificamente per il cameriere. Quando riceve un messaggio di terminazione da parte del cliente ("client_terminate"), lo inoltra anche al pub e chiude il socket del client, mentre quando riceve quello del pub ("pub_terminate"), lo inoltra anche al client e chiude il socket del pub.

- **handle_terminate_for_pub(int sock, char *message, int tavolo_liberato, SharedData *shared_data)**: gestisce messaggi di tipo "cameriere_terminate" o "client_terminate" specificamente per il pub. Differisce dalle altre perché questa (eventualmente) libera anche un tavolo rimasto occupato.

Le costanti definite sono:

- **PUB_IP**: indirizzo IP del pub (localhost in questo caso).
- **PUB_PORT**: porta su cui si mette in ascolto il pub.
- **CAMERIERE_PORT**: porta su cui si mette in ascolto il cameriere.

3.3 Dettagli implementativi del Pub

3.3.1 Configurazione della memoria condivisa

Il Pub, una volta eseguito, dovrà configurare la memoria condivisa.
Per fare ciò, chiamerà innanzitutto la funzione **shmget**, la quale andrà a creare un'area di memoria condivisa e restituirà il suo identificatore in **shmid**.

```
if ((shmid = shmget(key, sizeof(SharedData), IPC_CREAT | 0666)) < 0) {
    perror("Errore durante la shmget");
    exit(EXIT_FAILURE);
}
```

Dopo aver ottenuto l'identificatore, viene utilizzata la funzione **shmat** per associare l'area di memoria condivisa all'indirizzo di memoria del processo.
Il puntatore restituito dalla funzione punta all'inizio dell'area di memoria condivisa.

```
if ((shared_data = (SharedData *)shmat(shmid, NULL, 0)) == (SharedData *)-1) {
    perror("Errore durante la shmat");
    exit(EXIT_FAILURE);
}
```

Dopo aver associato l'area di memoria condivisa al processo, è possibile utilizzare il puntatore **shared_data** per accedere e modificare i dati all'interno dell'area di memoria condivisa. Nel caso del Pub, viene inizializzato l'array **tavoli_occupati** all'interno della struttura **SharedData**, che tiene traccia dello stato (occupato o libero) di ogni tavolo nel pub.

```
for (int i = 0; i < NUMERO_TAVOLI; i++) {
    shared_data->tavoli_occupati[i] = 0; // 0 indica che il tavolo è libero
}
```

A questo punto, il pub crea il socket e si mette in ascolto per le connessioni.

3.3.2 Fork per gestire in modo concorrente i camerieri

Terminata la fase iniziale, si entra in un ciclo while infinito in cui il server del pub accetta connessioni in arrivo e ottiene il socket associato al cameriere che si è connesso; in particolare, viene eseguita una **fork** del processo corrente per creare un processo figlio.
Se la fork ha successo, il processo figlio continua l'esecuzione e si occuperà di gestire le richieste del cameriere, mentre il processo padre tornerà ad accettare altre connessioni.
Se la fork fallisce (ritorna -1), viene stampato un messaggio di errore e il socket associato al cameriere viene chiuso.

```
while (1) {
    struct sockaddr_in client_addr;
    int client_sock = accept_connection(server_sock, &client_addr);

    // Fork per gestire le comunicazioni coi camerieri
    pid_t pid = fork();
    if (pid == -1) {
        perror("Errore durante la fork");
        close(client_sock);
        continue;
    }

    if (pid == 0) {
        // Processo figlio
```

3.3.3 Accettazione di un cliente se ci sono tavoli disponibili

Quando un cameriere chiede al Pub se ci sono posti disponibili per far accomodare i clienti, quest'ultimo procede con la fase di controllo e assegnazione del tavolo chiamando la funzione **assegna_tavolo**, passandogli il puntatore all'area di memoria condivisa in cui è presente l'array che tiene traccia dei tavoli disponibili.
La funzione scorrerà tutto l'array dei tavoli occupati e restituirà come valore il primo tavolo trovato libero (==0) o -1 nel caso in cui non ci siano tavoli disponibili.
Nel caso in cui sia stato assegnato un tavolo libero, il relativo elemento nell'array **tavoli_occupati** sarà impostato ad 1.

```c
int assegna_tavolo(SharedData *shared_data) {
    int tavolo_assegnato = -1; // Inizializziamo a -1 in caso non ci siano tavoli disponibili
    for (int i = 0; i < NUMERO_TAVOLI; i++) {
        if (shared_data->tavoli_occupati[i] == 0) { // Se il tavolo è libero
            tavolo_assegnato = i + 1; // Assegniamo il tavolo disponibile più basso
            shared_data->tavoli_occupati[i] = 1; // Impostiamo il tavolo come occupato
            break;
        }
    }
    return tavolo_assegnato;
}
```

A questo punto, il Pub comunicherà al cameriere se è possibile far accomodare o meno il cliente in base al valore restituito dalla funzione **assegna_tavolo**.

```c
// Provo ad assegnare un tavolo
int tavolo_assegnato = assegna_tavolo(shared_data);

if (tavolo_assegnato == -1) { // Se non è stato assegnato un tavolo
    // Comunico al cameriere che non ci sono posti disponibili
    strcpy(message, "Mi dispiace, non ci sono tavoli disponibili.");
    send(client_sock, message, strlen(message), 0);
} else {
    // Comunico al cameriere che c'è un tavolo disponibile e quale è stato assegnato
    snprintf(message, sizeof(message), "Sì, c'è il tavolo %d disponibile.\n", tavolo_assegnato);
    printf("%s", message);
    fflush(stdout);

    // Invio la risposta al cameriere
    send(client_sock, message, strlen(message), 0);
```

3.3.4 Preparazione degli ordini

Quando il cameriere ritira un ordine da un tavolo e lo consegna al Pub, quest'ultimo deve procedere con la fase di preparazione dell'ordine.
Se ne occuperà la funzione **prepara_ordine** che, una volta chiamata, andrà a simulare la preparazione di un ordine per il tavolo che gli è stato passato come parametro (ad esempio, facendo passare 3 secondi).

```c
// Funzione per simulare la preparazione dell'ordine
void prepara_ordine(int tavolo_assegnato) {
    printf("Preparazione dell'ordine per il tavolo %d in corso...\n", tavolo_assegnato);
    sleep(3);
    printf("L'ordine per il tavolo %d è pronto per essere ritirato dal cameriere!\n", tavolo_assegnato);
}
```

Una volta terminata la preparazione dell'ordine, il Pub avvisa il cameriere che quell'ordine è pronto per essere ritirato e servito al tavolo.

```c
// Preparo l'ordine
prepara_ordine(tavolo_assegnato);

memset(message, 0, MESSAGE_SIZE);

// Dopo la preparazione dell'ordine, il pub avvisa il cameriere che è pronto
snprintf(message, sizeof(message), "Ordine per il tavolo %d pronto al servizio.", tavolo_assegnato);
send(client_sock, message, strlen(message), 0);
```

3.3.5 Liberazione dei tavoli

L'ultima parte di codice del Pub prevede la liberazione di un tavolo quando un cliente ha finito di consumare ed è uscito. Non appena arriva l'avviso da parte del cameriere, il Pub chiama la funzione **libera_tavolo** passandogli come parametri il numero del tavolo da rendere nuovamente disponibile e il puntatore all'area di memoria condivisa.

```c
// Processo di liberazione del tavolo
receive_message(client_sock, message, MESSAGE_SIZE);
handle_terminate_for_pub(client_sock, message, tavolo_assegnato, shared_data);
libera_tavolo(tavolo_assegnato, shared_data);
```

La funzione, una volta decrementato di 1 il valore del tavolo (poiché l'array parte da 0), verifica se quest'ultimo è effettivamente un tavolo valido e imposta il relativo indice dell'array a 0, per indicare la sua disponibilità.

```c
void libera_tavolo(int tavolo_liberato, SharedData *shared_data) {
    tavolo_liberato--;
    if (tavolo_liberato >= 0 && tavolo_liberato < NUMERO_TAVOLI) {
        shared_data->tavoli_occupati[tavolo_liberato] = 0; // Imposta il tavolo come libero
        printf("Il tavolo %d è stato liberato e ora è disponibile.\n", tavolo_liberato + 1);
    } else {
        printf("Tentativo di liberare un tavolo non valido.\n");
    }
}
```

Una volta liberato, viene stampato un messaggio di conferma.

3.4 Dettagli implementativi del Cameriere

3.4.1 Fork per gestire in modo concorrente i clienti

Una volta creato il socket ed essersi messo in ascolto per i client, il Cameriere entra anche lui in un ciclo while infinito in cui accetta le connessioni in arrivo e ottiene il socket associato al client che si è connesso; in particolare, anche in questo caso viene eseguita una fork del processo corrente per creare un processo figlio.
Al solito, se la fork ha successo, il processo figlio continua l'esecuzione e si occuperà di gestire le richieste dei client, mentre il processo padre tornerà ad accettare altre connessioni.

Se la fork fallisce (ritorna -1), viene stampato un messaggio di errore e il socket associato al client viene chiuso.

3.4.2 Richiesta al pub per la disponibilità di tavoli e consegna del menù

Quando un client si connette e chiede al Cameriere se c'è un tavolo disponibile per lui all'interno del pub, il Cameriere inoltra questa richiesta al pub, poiché (come abbiamo già visto) dovrà occuparsene lui. In base alla risposta ricevuta dal pub, il Cameriere deciderà correttamente se rifiutare il cliente ed invitarlo a passare più tardi oppure farlo accomodare al tavolo assegnato dal pub e servirgli il menù per poter effettuare l'ordine.

```
// Ricevo la risposta del pub
receive_message(pub_sock, message, MESSAGE_SIZE);
handle_terminate_for_cameriere(pub_sock, client_sock, message);
printf("Risposta del pub: %s", message);

int tavolo_assegnato;
if (!strncmp(message, "S", strlen("S")) == 0) {
    // Avvisa il cliente che non ci sono posti disponibili nel pub
    send(client_sock, message, strlen(message), 0);
} else {
    // Memorizza dalla stringa il numero di tavolo assegnato
    sscanf(message, "Sì, c'è il tavolo %d disponibile.", &tavolo_assegnato);
    printf("Faccio accomodare il cliente e gli servo il menù...\n");
    sleep(2);

    memset(message, 0, MESSAGE_SIZE);

    // Invia il menù al cliente specificando il tavolo
    invia_menu(client_sock, message, tavolo_assegnato);
```

In quest'ultimo caso, verrà "estratto" dalla stringa ricevuta dal pub il numero di tavolo assegnato e memorizzato in una variabile (tramite la funzione **sscanf**), dopodiché verrà chiamata la funzione **invia_menu** che si occuperà di far accomodare il cliente al tavolo e servirgli il menù.

3.4.3 Ritiro delle ordinazioni e invio al pub

Nel momento in cui un client termina il suo ordine, il Cameriere lo riceve e lo inoltra al pub in poche semplici istruzioni.

```c
// Ricevi l'ordine effettuato dal cliente
receive_message(client_sock, message, MESSAGE_SIZE);
handle_terminate_for_cameriere(pub_sock, client_sock, message);
printf("Ordine ritirato dal tavolo %d: %s.\nLo invio al pub...\n", tavolo_assegnato, message);
sleep(2);

// Inoltra l'ordine al pub
send(pub_sock, message, strlen(message), 0);
```

Analogamente, una volta che il pub ci avvisa del fatto che l'ordine è pronto, il Cameriere va a servirlo al cliente.

```c
// Ricevi l'avviso riguardo l'ordine pronto dal pub
receive_message(pub_sock, message, MESSAGE_SIZE);
handle_terminate_for_cameriere(pub_sock, client_sock, message);
printf("Avviso dal pub: %s\nVado a servirlo...\n", message);
sleep(2);

// Servo l'ordine al cliente
send(client_sock, message, strlen(message), 0);
```

3.4.4 Avviso di liberazione tavolo al pub

L'ultima parte di codice del Cameriere prevede che, una volta ricevuto dal client il messaggio relativo alla sua uscita dal pub, il Cameriere pulisca il tavolo e inoltri al pub il messaggio, in modo tale che quest'ultimo segni il tavolo che occupava quel client come nuovamente disponibile.

```c
// Ricevi il messaggio dal client riguardo la sua uscita dal pub
receive_message(client_sock, message, MESSAGE_SIZE);
handle_terminate_for_cameriere(pub_sock, client_sock, message);
printf("Il cliente al tavolo %d ha lasciato il pub, pulisco il tavolo e comunico al pub...\n", tavolo_assegnato);
sleep(2);

// Comunicalo al pub per fargli liberare il tavolo
send(pub_sock, message, strlen(message), 0);
```

3.5 Dettagli implementativi del Client

3.5.1 Richiesta di accesso al pub e ricezione del menù

Quando un Client viene eseguito, crea il socket e si connette al cameriere per chiedergli se c'è un tavolo disponibile nel pub.

```c
// Creazione del socket del client e connessione al cameriere
sock = connect_to_address(PUB_IP, CAMERIERE_PORT);
set_socket_option(sock);

printf("Sono entrato nel pub!\nChiedo se c'è un tavolo disponibile...\n");
sleep(2);

char message[MESSAGE_SIZE];

// Chiedo al cameriere se c'è un tavolo disponibile
strcpy(message, "C'è un tavolo disponibile?");
send(sock, message, strlen(message), 0);
```

Successivamente, riceve la risposta dal cameriere e se questo gli ha dato una risposta negativa, il Client sarà invitato a riprovare più tardi e verrà chiuso il socket.

```c
// Se non ci sono tavoli disponibili, chiudo la connessione e termino l'esecuzione
if (strcmp(message, "Mi dispiace, non ci sono tavoli disponibili.") == 0) {
    printf("Riprova più tardi.\n");
    close(sock);
    exit(EXIT_SUCCESS);
}
```

Invece, in caso di risposta positiva, il Client verrà fatto accomodare ad un tavolo libero e gli verrà servito il menù.

3.5.2 Effettuare l'ordine

Il Client potrà scegliere una portata alla volta dal menù digitando un numero compreso tra 1 e 12. È obbligatorio ordinare almeno una portata e una volta terminato l'ordine basterà digitare 'exit' per lasciarlo al cameriere.
Il tutto è controllato all'interno di un ciclo **do-while**.

```c
// Ciclo per acquisire l'ordine dall'utente
do {
    printf("Portata: ");
    fgets(input, sizeof(input), stdin); // Ottengo l'input del cliente
    input[strcspn(input, "\n")] = '\0'; // Rimuovo il carattere di newline dall'input

    // Controlla se il cliente vuole concludere l'ordine
    if (strcmp(input, "exit") == 0) {
        // Se il cliente tenta di uscire senza aver ordinato nulla, chiedi di ordinare almeno una portata
        if (strlen(order) == 0) {
            printf("Devi ordinare almeno una portata.\n");
            continue; // Torna all'inizio del ciclo per richiedere la portata
        }
        break;
    }

    if (!is_valid_number(input, 1, 12)) {
        printf("Inserisci un numero compreso tra 1 e 12.\n");
        continue;
    }

    strcat(order, input); // Aggiungo la portata inserita all'ordine
    strcat(order, " ");   // Aggiungo uno spazio per separare le portate
} while (1);
```

3.5.3 Uscita dal pub

Una volta ricevute le portate ordinate e aver consumato il tutto, è il momento di abbandonare il pub. Nel farlo avvisiamo anche il cameriere inviandogli un messaggio 'exit', in modo tale che potrà pulire il tavolo e informare il pub che va reso nuovamente disponibile ai Client in arrivo.

```c
// Ricevo le portate dal cameriere
receive_message(sock, message, MESSAGE_SIZE);
handle_terminate_generic(sock, message);
printf("Il cameriere ti ha appena servito ciò che hai ordinato e stai mangiando...\n");
sleep(3);

printf("Hai finito e stai uscendo dal pub.\n");
sleep(2);
// Comunico al cameriere che ho finito di consumare e lascio il pub
strcpy(message, "exit");
send(sock, message, strlen(message), 0);

close(sock);
return 0;
```

4 Manuale utente

I codici nella cartella del progetto sono organizzati come segue:

- **'src'**: cartella contenente i codici sorgenti.
- **'include'**: cartella contenente le librerie utilizzate.

4.1 Compilazione ed esecuzione del progetto

Per compilare manualmente l'intero progetto, bisogna posizionarsi nella directory principale attraverso il comando '**cd**' e digitare i seguenti comandi **gcc**:

- **gcc -o pub src/pub.c src/pub_utils.c src/socket_utils.c -Iinclude**
- **gcc -o cameriere src/cameriere.c src/pub_utils.c src/socket_utils.c -Iinclude**
- **gcc -o client src/client.c src/pub_utils.c src/socket_utils.c -Iinclude**

N.B. Se si fa copia/incolla dei comandi, per qualche strano motivo potrebbero essere omessi gli underscore '_', dunque mi raccomando di verificare questa cosa. Fatto ciò, bisogna eseguire rispettando l'ordine di esecuzione elencato di seguito (per un corretto funzionamento):

- **./pub**
- **./cameriere**
- **./client**