

**UNIVERSITÀ DEGLI STUDI DI NAPOLI PARTHENOPE**

**DIPARTIMENTO SCIENZE E TECNOLOGIE**

**RETI DI CALCOLATORI E LABORATORIO RETI DI  
CALCOLATORI**



**PRESENTA IL PROGETTO ESAME:**

**PUB**

**SVILUPPATO DA:  
PASQUALE FERRANDINO  
MATRICOLA 0124001857  
A.A. 2024/2025**

**DOCENTE:  
EMANUEL DI NARDO**

# DESCRIZIONE DEL PROGETTO

Il progetto consiste nello sviluppo di un applicazione client/server parallelo di una gestione di un pub.

Il progetto è strutturato come segue:

C'è un solo cliente per tavolo

*Il Pub:*

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

*Il Cameriere:*

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

*Il Cliente:*

- Chiede se può entrare nel pub
- Richiede al cameriere il menu
- Effettua un ordine

# STRUTTURA DEL PROGETTO

Il progetto è sviluppato nel linguaggio di programmazione C e le socket su piattaforma Unix-Like.

Vi abbiamo diversi componenti del sistema tra cui:

**Il Pub:** che è il nostro server dove consente la connessione ai nostri camerieri. Esso gestisce i tavoli per far accomodare i clienti e prepara gli ordini.

**Il Cameriere:** gestisce le richieste del cliente e le inoltra al pub, facendo da intermediario tra cliente e pub

**Cliente:** rappresenta l'utente finale che interagisce con il sistema per prenotare un tavolo e per fare un ordine.

**Header.c:** contiene una serie di funzioni di supporto e una lista di elementi di menu.

**Header.h:** è un header file che definisce le costanti, le strutture e le dichiarazioni delle funzioni utilizzate nel progetto. All'interno vi sono librerie standard di C e Unix necessarie per il funzionamento del progetto.

# SCHEMA DI ARCHITETTURA

lo schema architetturale segue un modello client-server con più livelli, ed è strutturato in questo modo:



Dove:

- **Il cliente:** rappresenta l'utente finale che interagisce con il sistema per effettuare un'ordinazione. Invia richieste al "Cameriere" (che funge da intermediario), come ad esempio la visualizzazione del menu o l'invio di un ordine e riceve risposte dal Cameriere, come la conferma dell'ordine o informazioni riguardo al menu.
- **Il cameriere:** è una componente intermedia tra clienti e pub. Riceve richieste dai clienti e le elabora o le inoltra al Pub. Coordina le interazioni tra il cliente e il Pub, gestendo ordini, invio di informazioni e mantenendo lo stato della comunicazione.
- **Il pub:** Il Pub è il sistema centrale (o server) che gestisce le operazioni principali, come la gestione dei tavoli e degli ordini. Riceve le richieste dal Cameriere, ad esempio per confermare un ordine o aggiornare la disponibilità dei posti/tavoli. E invia al cameriere la richiesta dell'ordine pronto per essere servito al tavolo del cliente.

# DETTAGLI DI IMPLEMENTAZIONE

## PUB

Uno dei compiti principali del Pub, oltre all'accoglienza del cliente, è quella di controllare la disponibilità dei posti.

```
// Funzione per assegnare un tavolo disponibile
int assegna_posto() {
    for (int i = 0; i < MAX_POSTI; i++) {
        if (pub.posti[i] != 0) {
            int num_posto = pub.posti[i];
            pub.posti[i] = 0;
            pub.posti_liberi--;
            return num_posto;
        }
    }
    return -1; // Nessun posto disponibile
}

// Verifica posti disponibili e assegna un tavolo
pthread_mutex_lock(&pub.mutex);
num_posto = assegna_posto();
pthread_mutex_unlock(&pub.mutex);

if (num_posto != -1) {
    // Invia al cameriere il numero del tavolo assegnato
    sprintf(buffer, "Posto disponibile, tavolo libero numero %d\n", num_posto);
    send_message(client->socket_fd, buffer);
} else {
    // Invia un messaggio che indica che non ci sono posti disponibili
    send_message(client->socket_fd, "Spiacente, nessun posto disponibile.");
    close(client->socket_fd);
    free(client);
    return NULL;
}
```

Con queste funzioni, il pub controlla quale tavolo sia disponibile per far accomodare il cliente. Quando il tavolo è libero, manderà un messaggio al cameriere dove avviserà che il cliente potrà accomodarsi.

In caso di mancata disponibilità esso avviserà al cameriere che a sua volta avviserà al cliente di passare più tardi.

Ho utilizzato **pthread\_mutex\_t** per sincronizzare l'accesso concorrente ai

tavoli, dato che più thread potrebbero tentare di accedere o modificare le stesse risorse contemporaneamente.

```
// Ricezione dell'ordine dal cameriere
printf("In attesa del cameriere che ritira l'ordine...\n");
if (receive_message(client->socket_fd, buffer) < 0) {
    perror("Errore nella ricezione dell'ordine");
    close(client->socket_fd);
    free(client);
    return NULL;
}

// Stampa l'ordine ricevuto con il nome del prodotto
printf("Ordine ricevuto dal cameriere per il tavolo %d: %s\n", num_posto, buffer);

// Simulazione della preparazione dell'ordine
send_message(client->socket_fd, "Ordine confermato. Preparazione in corso...");
sleep(1);
send_message(client->socket_fd, "Ordine pronto. Consegna in corso...");

close(client->socket_fd);
```

Dopo che il cliente avrà effettuato l'ordine, il pub attenderà che il cameriere lo consegni per essere elaborato. Una volta elaborato verrà confermato l'ordine pronto per la consegna al cameriere dove quest'ultimo effettuerà la consegna al cliente

```
// Libera il tavolo dopo che il cliente ha finito
pthread_mutex_lock(&pub.mutex);
libera_posto(num_posto);
pthread_mutex_unlock(&pub.mutex);
.
```

Una volta che il cliente avrà effettuato il pagamento, lascerà automaticamente il locale e quindi il tavolo tornerà un'altra volta disponibile.

```

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Inizializza i tavoli del Pub
    inizializza_tavoli_pub();

    // Creazione del socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        error("Errore nella creazione del socket");
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PUB_PORT);

    // Binding del socket
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        error("Errore nel binding");
    }

    // Imposta il server in ascolto
    if (listen(server_fd, 3) < 0) {
        error("Errore nella listen");
    }

    printf("Pub aperto e in attesa di clienti...\n");

    // Ciclo principale per accettare connessioni dai clienti
    while (1) {
        // Accetta nuove connessioni in arrivo
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
            perror("Errore nell'accept");
            continue;
        }

        // Creazione del client e gestione del thread
        Client *client = malloc(sizeof(Client));
        if (client == NULL) {
            perror("Errore nell'allocazione della memoria per il client");
            close(new_socket);
            continue;
        }

        client->socket_fd = new_socket;
        client->address = address;

        // Creazione di un nuovo thread per gestire il client
        pthread_t tid;
        if (pthread_create(&tid, NULL, handle_client, (void *)client) != 0) {
            perror("Errore nella creazione del thread");
            close(new_socket);
            free(client);
            continue;
        }

        pthread_detach(tid); // Assicura che le risorse del thread siano liberate al termine
    }

    close(server_fd);
    return 0;
}

```

Nel main vi abbiamo tutta la creazione del Server.

All'interno del server, creiamo un socket utilizzando il protocollo **TCP/IP** (specificato da `SOCK_STREAM`). Questo è un tipo di socket orientato alla connessione, ideale per la comunicazione sicura tra server e client.

Dopo aver creato il socket, dobbiamo assegnargli un indirizzo IP e una porta. Dopodiché effettuiamo il binding del socket che associa l'indirizzo e la porta definiti al socket creato, in modo che il server possa ascoltare le richieste su

quella porta specifica. A seguire creiamo un ciclo while infinito che accetta le connessioni dei client.

Una volta accettata la connessione, il server assegna a ogni client un thread separato che gestirà la comunicazione con quel particolare cliente. Ciò consente al server di gestire più clienti contemporaneamente. Infine quando il cliente ha finito, il tavolo viene liberato e il thread viene terminato.

## CAMERIERE

Il cameriere si pone come intermediario tra il cliente e il pub, ricevendo ordini dal cliente, inoltrandoli al pub, e riportando le risposte al cliente. Il sistema sfrutta la programmazione multithread per gestire contemporaneamente più clienti.

Una delle funzioni principali del cameriere è quella di accogliere il cliente. Esso riceverà la richiesta da parte del cliente, dove quest'ultimo chiederà se ci sia un tavolo disponibile. Il cameriere inoltra la richiesta al pub (poiché è lui che gestisce i tavoli) in caso di esito positivo il cameriere farà accomodare il cliente, in caso negativo, il cliente dovrà attendere che i posti si liberano.

```
// Riceve richiesta dal cliente
receive_message(client->socket_fd, buffer);
printf("Richiesta dal cliente: %s\n", buffer);

// Invia richiesta di posto al Pub
send_message(pub_socket, "Il cliente chiede se c'è un tavolo disponibile");
receive_message(pub_socket, buffer);
printf("Risposta dal Pub: %s\n", buffer);

// Inoltra risposta al cliente
send_message(client->socket_fd, buffer);
```

Una volta che il cliente si sarà accomodato, esso effettuerà l'ordine dove il compito del cameriere sarà di inoltrarlo al pub e a sua volta attendere che l'ordine sia completato per poterlo consegnare al cliente.

```
if (strstr(buffer, "Posto disponibile") != NULL) {

    // Ricezione dell'ordine dal cliente
    receive_message(client->socket_fd, buffer);
    printf("Il cliente ha ordinato: %s\n", buffer);

    // Invia ordine al Pub
    send_message(pub_socket, buffer);
    receive_message(pub_socket, buffer);
    printf("Ordine confermato dal Pub: %s\n", buffer);

    // Invia conferma al cliente
    send_message(client->socket_fd, buffer);

    // Ricezione conferma dell'ordine pronto dal Pub
    receive_message(pub_socket, buffer);
    printf("Ordine pronto per la consegna: %s\n", buffer);
    send_message(client->socket_fd, buffer);
```



Infine, una volta che il cliente avrà consumato, simuliamo il pagamento da parte del cliente al cameriere.

```
// Simula pagamento
receive_message(client->socket_fd, buffer);
printf("Pagamento dal cliente: %s\n", buffer);
send_message(client->socket_fd, "Grazie per aver pagato, arrivederci!");
}
```

La funzione main() del cameriere avvia il server, creando un socket TCP per ascoltare le connessioni in arrivo da parte dei clienti.

Viene creato un socket tramite la chiamata socket(AF\_INET, SOCK\_STREAM, 0), che indica che il server utilizzerà il protocollo IPv4 e TCP per gestire le connessioni. Il socket viene associato all'indirizzo IP e alla porta (CAMERIERE\_PORT, definita come 8081 nel file header.h), permettendo al server di ricevere connessioni su quella porta. La funzione listen() mette il server in ascolto delle connessioni, permettendo fino a 3 connessioni in coda. Una volta che una connessione viene accettata con accept(), il cameriere crea un nuovo thread per gestire la richiesta del cliente. Ogni connessione con il cliente viene gestita da un thread separato, tramite la funzione pthread\_create(). Questo permette al cameriere di gestire più clienti simultaneamente. Il thread viene staccato per garantire che le risorse del thread vengano liberate automaticamente una volta terminata la gestione del cliente

```
int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    // Creazione del socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        error("Errore nella creazione del socket");
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(CAMERIERE_PORT);

    // Binding
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        error("Errore nel binding");
    }

    // Listen
    if (listen(server_fd, 3) < 0) {
        error("Errore nella listen");
    }

    printf("Cameriere avviato e in attesa di clienti...\n");

    while (1) {
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0) {
            error("Errore nell'accept");
        }

        Client *client = malloc(sizeof(Client));
        client->socket_fd = new_socket;
        client->address = address;

        pthread_t tid;
        pthread_create(&tid, NULL, handle_client, (void *)client);
        pthread_detach(tid);
    }

    close(server_fd);
    return 0;
}
```

# CLIENTE

Il cliente rappresenta il client del progetto. Questo client consente agli utenti di connettersi al cameriere, richiedere un tavolo, visualizzare il menu, effettuare un ordine e simulare il pagamento. La comunicazione tra il client, il cameriere e il pub avviene tramite socket TCP, garantendo una trasmissione affidabile dei dati.

Il main del cliente è il punto di ingresso del programma e gestisce l'intero flusso operativo. Il cliente si connetterà al server tramite la porta del cameriere.

```
int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        error("Errore nella creazione del socket");
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(CAMERIERE_PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        error("Indirizzo non valido o non supportato");
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        error("Connessione fallita");
    }
}
```

Una volta connesso al server. Il cliente domanderà al cameriere se ci sia un posto disponibile. Attenderà la risposta del cameriere (che lo inoltrerà al pub) dove, in caso positivo il cliente si accomoderà al tavolo indicato dal server, indicando anche il numero del tavolo, in caso negativo il cliente dovrà riprovare e attendere che si liberi un tavolo.

```
// Chiede se ci sono posti disponibili
send_message(sock, "Salve, è disponibile un tavolo?");
receive_message(sock, buffer);
printf("Risposta dal cameriere: %s\n", buffer);

// Se c'è un tavolo disponibile, estrai il numero del tavolo dalla risposta
int num_posto = -1;
if (sscanf(buffer, "Posto disponibile, tavolo numero %d", &num_posto) == 1) {
    printf("Ti è stato assegnato il tavolo numero %d.\n", num_posto);
}
```

Una volta accomodato il cliente riceverà il menu dove avrà la possibilità di scegliere cosa ordinare. Per effettuare l'ordine il cliente dovrà selezionare il numero corrispondente nel menù (da 1 a 14) e potrà selezionare più elementi.

```
if (strstr(buffer, "Posto disponibile") != NULL) {
    // Mostra il menu
    mostra_menu();

    // L'utente sceglie dal menu
    printf("Inserisci il numero degli elementi che desideri ordinare (es. 1 3 5): ");
    fgets(buffer, BUFFER_SIZE, stdin);

    // Invia l'ordine al cameriere
    send_message(sock, buffer);

    // Riceve conferma dal cameriere
    receive_message(sock, buffer);
    printf("Conferma dal cameriere: %s\n", buffer);

    // Riceve l'ordine preparato
    receive_message(sock, buffer);
    printf("Hai ricevuto il tuo ordine. Buon appetito!\n");
}
```

Infine, una volta che il cliente avrà ricevuto l'ordine e consumato simulerà il pagamento dell'ordine effettuato.

```
// Simula pagamento
send_message(sock, "Pagamento effettuato");
receive_message(sock, buffer);
printf("Hai ricevuto il conto: %s\n", buffer);
```

# HEADER.C

Nell'Header.c vi sono tutte le funzioni di supporto utilizzate dal client e dal cameriere per gestire la comunicazione tramite socket, la gestione degli errori e la visualizzazione del menu. Le funzionalità implementate sono essenziali per semplificare la comunicazione, rendere il codice più modulare e riutilizzabile, oltre a fornire una gestione degli errori efficace e centralizzata.

```
#include "header.h"

// Menu delle portate
const char *menu[] = {
    "1. Birra",
    "2. Vino",
    "3. Panino Porchetta e provola",
    "4. Patatine e carne",
    "5. Pizza Margherita",
    "6. Hot Dog",
    "7. Hamburger cheddar e patatine",
    "8. Panino Salsiccia e friarielli",
    "9. Fanta",
    "10. Gelato",
    "11. Caffè",
    "12. Limoncello",
    "13. Bustina ketchup",
    "14. Bustina maionese"
};

// Funzione per mostrare il menu
void mostra_menu() {
    printf("Menu:\n");
    for (int i = 0; i < 14; i++) {
        printf("%s\n", menu[i]);
    }
}

// Funzione per l'invio di messaggi attraverso il socket
void send_message(int socket, const char *message) {
    int length = strlen(message);
    if (send(socket, message, length, 0) != length) {
        error("Errore nell'invio del messaggio");
    }
}

// Funzione per la ricezione di messaggi attraverso il socket
int receive_message(int socket, char *buffer) {
    int valread = recv(socket, buffer, BUFFER_SIZE, 0);
    if (valread <= 0) {
        perror("Errore nella ricezione del messaggio");
        return -1; // Ritorna -1 in caso di errore
    }
    buffer[valread] = '\0'; // Terminazione stringa
    return valread; // Ritorna il numero di byte letti
}

// Funzione di gestione errori
void error(const char *msg) {
    perror(msg);
    exit(1);
}
```

# HEADER.H

Il file header.h è un file di intestazione utilizzato in un sistema client-server, che definisce costanti, strutture dati e le dichiarazioni delle funzioni utilizzate in altri file sorgente. La sua funzione principale è rendere il codice modulare e condividere definizioni e dichiarazioni tra più file senza duplicazione di codice. Tutte le componenti necessarie per la gestione della comunicazione e delle funzionalità tra il cameriere, il client e il Pub sono dichiarate in questo file. Il file include una serie di librerie standard di C per la gestione di input/output, allocazione della memoria, thread e socket di rete. Dopodichè abbiamo definito delle costanti (il limite massimo di posti a sedere, le porte di connessione del pub e del cameriere e la dimensione del buffer per la ricezione e l'invio di messaggi attraverso i socket). Abbiamo le strutture che memorizzano le informazioni del client e del pub. E la dichiarazione delle funzioni generiche.

```
#ifndef HEADER_H
#define HEADER_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <arpa/inet.h>
#include <netinet/in.h>

// Costanti
#define MAX_POSTI 5
#define PUB_PORT 8080
#define CAMERIERE_PORT 8081
#define BUFFER_SIZE 1024

// Strutture
typedef struct {
    int socket_fd;
    struct sockaddr_in address;
} Client;

typedef struct {
    int posti liberi;
    int posti[MAX_POSTI];
    pthread_mutex_t mutex;
} Pub;

// Funzioni generiche
void error(const char *msg);
void send_message(int socket, const char *message);
int receive_message(int socket, char *buffer);
void mostra_menu();

#endif // HEADER_H
```

## ESECUZIONE PROGETTO

Per eseguire bene il progetto bisogna prima di tutto effettuare una giusta compilazione. In questo caso dobbiamo compilare come segue sul terminale:

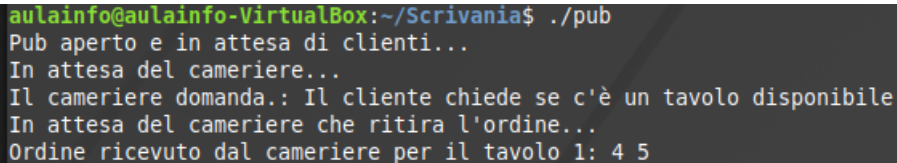
```
gcc -o pub pub.c header.c -lpthread
gcc -o cameriere cameriere.c header.c -lpthread
gcc -o cliente cliente.c header.c -lpthread
```

Una volta compilato il programma non resta che eseguirlo. Sempre sul terminale digitiamo nel seguente ordine:

```
./pub
./cameriere
./cliente
```

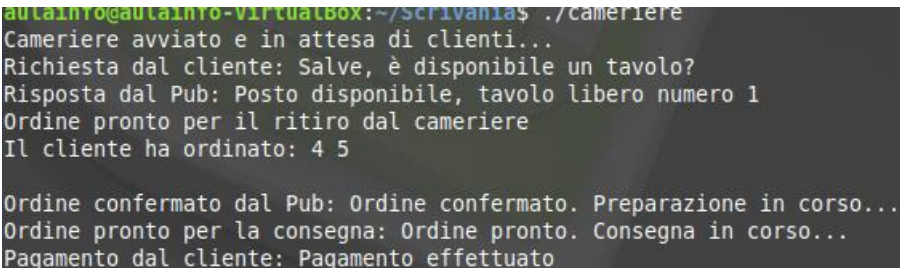
## ESEMPIO DI ESECUZIONE:

### PUB



```
aulainfo@aulainfo-VirtualBox:~/Scrivania$ ./pub
Pub aperto e in attesa di clienti...
In attesa del cameriere...
Il cameriere domanda.: Il cliente chiede se c'è un tavolo disponibile
In attesa del cameriere che ritira l'ordine...
Ordine ricevuto dal cameriere per il tavolo 1: 4 5
```

### CAMERIERE



```
aulainfo@aulainfo-VirtualBox:~/Scrivania$ ./cameriere
Cameriere avviato e in attesa di clienti...
Richiesta dal cliente: Salve, è disponibile un tavolo?
Risposta dal Pub: Posto disponibile, tavolo libero numero 1
Ordine pronto per il ritiro dal cameriere
Il cliente ha ordinato: 4 5

Ordine confermato dal Pub: Ordine confermato. Preparazione in corso...
Ordine pronto per la consegna: Ordine pronto. Consegna in corso...
Pagamento dal cliente: Pagamento effettuato
```

### CLIENTE

```
aulainfo@aulainfo-VirtualBox:~/Scrivania$ ./cliente
Risposta dal cameriere: Posto disponibile, tavolo libero numero 1
Ordine pronto per il ritiro dal cameriere
Menu:
1. Birra
2. Vino
3. Panino Porchetta e provola
4. Patatine e carne
5. Pizza Margherita
6. Hot Dog
7. Hamburger cheddar e patatine
8. Panino Salsiccia e friarielli
9. Fanta
10. Gelato
11. Caffè
12. Limoncello
13. Bustina ketchup
14. Bustina maionese
Inserisci il numero degli elementi che desideri ordinare (es. 1 3 5): 4 5
Conferma dal cameriere: Ordine confermato. Preparazione in corso...
Hai ricevuto il tuo ordine. Buon appetito!
Hai ricevuto il conto: Grazie per aver pagato, arrivederci!
aulainfo@aulainfo-VirtualBox:~/Scrivania$
```