Relazione Reti di Calcolatori e Laboratorio di Reti di Calcolatori

Antonio Iannone 0124002543

Francesco Pio Gravino 0124002703

August 23, 2024

Università degli Studi di Napoli Parthenope

GitHub Repository

Anno 2023/2024

Contents

1 Descrizione del progetto

Questa sezione del documento si propone di esplorare in dettaglio il progetto. Inizieremo analizzando l'architettura complessiva del sistema, delineando le responsabilità delle principali entità coinvolte: il Pub, il Cameriere e il Cliente. Successivamente, esamineremo gli schemi di comunicazione e le interazioni tra queste componenti.
Questa sezione mira a fornire una panoramica chiara e dettagliata del design e dell'implementazione dell'applicazione, evidenziando le decisioni architetturali e i flussi di lavoro che supportano il corretto funzionamento del sistema.

1.1 Traccia del progetto:

Scrivere un'applicazione client/server parallelo in cui viene effettuata la gestione di un pub

1.1.1 Pub

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

1.1.2 Cameriere

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

1.1.3 Cliente

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

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.

1.2 Note di Sviluppo

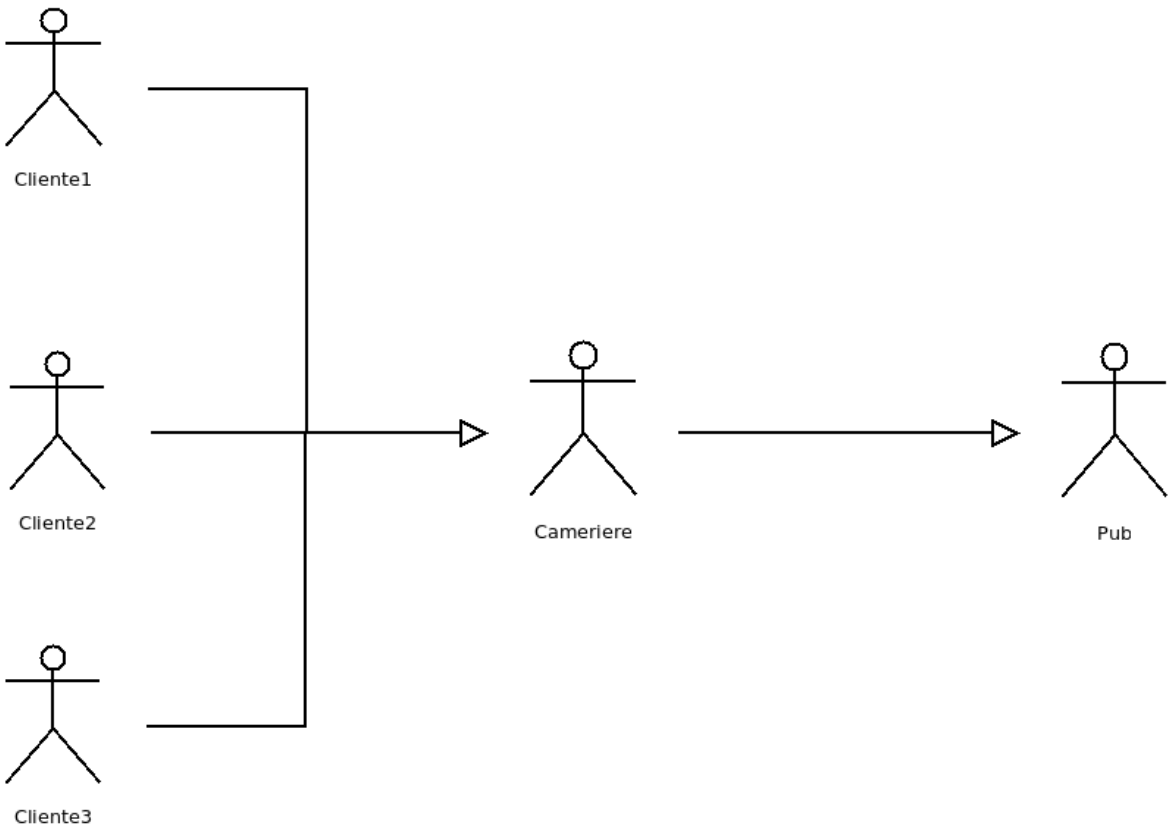
Il progetto è stato realizzato in C++ per due motivi fondamentali. In primo luogo, per mantenere una continuità con il linguaggio utilizzato durante il corso, evitando così di introdurre complessità aggiuntive dovute all'apprendimento di un nuovo linguaggio. In secondo luogo, il C++ consente di programmare in maniera efficace utilizzando la programmazione ad oggetti, che a nostro avviso è estremamente adatta per la realizzazione di questo progetto. La programmazione ad oggetti permette di modellare in modo naturale le entità coinvolte (Pub, Cameriere, Cliente) come classi, favorendo una chiara separazione delle responsabilità e una gestione più intuitiva dello stato e dei comportamenti delle diverse componenti del sistema.

L'intero progetto è stato sviluppato su una piattaforma Unix-like, sfruttando le socket per la comunicazione tra client e server.

2 Descrizione e schema dell'Architettura

In questo paragrafo verrà illustrata l'architettura complessiva del progetto, evidenziando le componenti principali e le loro interazioni. Verrà presentato uno schema dettagliato che fornirà una visione d'insieme del sistema, seguita da una descrizione approfondita di ciascun elemento, con particolare attenzione agli strumenti e alle tecnologie impiegate per la sua realizzazione.

2.1 Schema ed Entità coinvolte

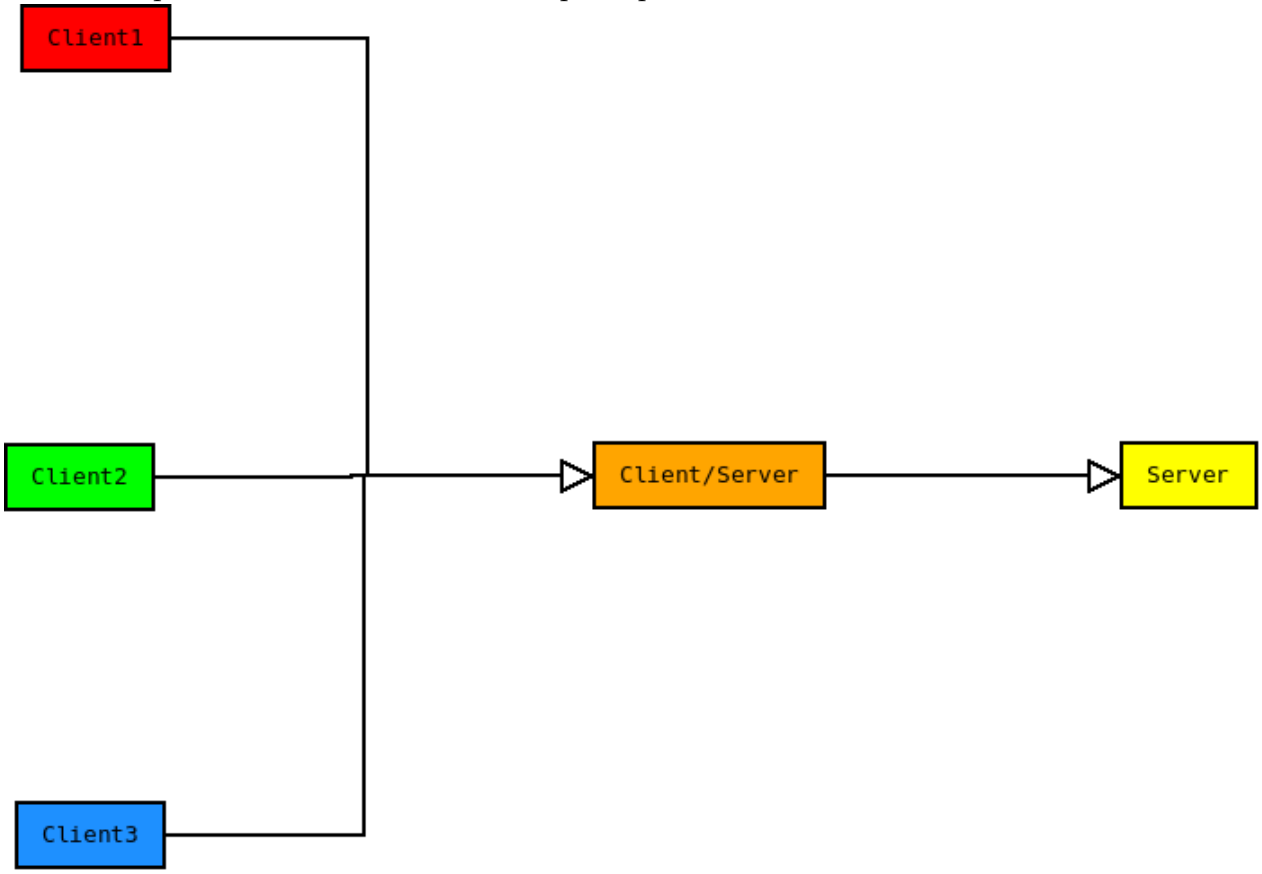


Il programma presentato si basa su un'architettura client-server. In questa configurazione, il client(o più client) invia una richiesta al server, il quale elabora la richiesta e risponde. In particolare, la comunicazione avviene tramite il protocollo TCP/IP utilizzando le socket. Il server è implementato come un **server concorrente**, il che significa che può gestire più richieste di connessione contemporaneamente. Questo approccio è essenziale quando ci si aspetta che il server debba servire più client allo stesso tempo,come nel nostro caso, garantendo un

servizio continuo e senza ritardi significativi per ciascun client. In un server non concorrente, ogni richiesta verrebbe gestita una alla volta, il che potrebbe causare ritardi o addirittura rifiuto delle connessioni se il server è occupato con una richiesta precedente. Nella nostra implementazione Il server concorrente descritto è implementato utilizzando la chiamata di sistema **fork()**, che andremo ad analizzare nel dettaglio nel capitolo successivo.

3 Implementazione del codice

In questo capitolo, esamineremo in dettaglio il codice sviluppato in `C++` per l'implementazione del sistema di gestione del *pub*. Analizzeremo passo per passo ogni entità coinvolta nel progetto, descrivendo il funzionamento del `client`, del `cameriere` e del `pub` stesso. Il nostro obiettivo sarà di spiegare e motivare tutte le scelte di programmazione fatte, fornendo una chiara comprensione delle tecniche e dei principi utilizzati nella costruzione.



È fondamentale tenere sempre a mente questo semplice schema per avere chiaro il tipo di architettura implementata.

3.1 Socket.h

Il primo file che andremo ad analizzare è il file *Socket.h*, ovvero un header file *C++*, che contiene la definizione della classe "Socket" che verrà utilizzata per gestire la comunicazione di rete nella restante parte del progetto.

Iniziamo definendo le librerie utilizzate in questo file. È importante notare che le librerie descritte in questo paragrafo non saranno riesaminate nei capitoli successivi, se riutilizzate, al fine di evitare ripetizioni.

```
#include <iostream> //input/output standard in C++
#include <cstring>  //manipolazione di stringhe in C
#include <arpa/inet.h> //funzioni per le conversioni di indirizzi IP
#include <sys/socket.h>  //definizioni per i socket
#include <unistd.h>  //Fornisce accesso a chiamate di sistema come close
    per chiudere un file descriptor
#include <fcntl.h>  //Fornisce accesso a chiamate di sistema per
    manipolare file descriptor
```

Passiamo ora all'analisi dettagliata della classe Socket, una componente fondamentale del nostro progetto. In C++, una classe è una struttura che permette di raggruppare variabili (chiamate attributi) e funzioni (chiamate metodi) sotto un'unica entità, definendo così un nuovo tipo di dato. Le classi sono uno degli elementi chiave della programmazione orientata agli oggetti, consentendo di modellare concetti complessi attraverso l'incapsulamento dei dati e delle funzionalità. Nel nostro caso, la classe Socket è progettata per gestire le operazioni di rete, come la creazione di connessioni, l'invio e la ricezione di dati. Nei paragrafi successivi, esploreremo passo dopo passo i dettagli della sua implementazione, spiegando e motivando ogni scelta di progettazione.

```
class Socket {
    private:
        int m_sock; // descrizione del socket
        sockaddr_in m_addr; // indirizzo del socket (IP e porta)
    public:
```

```
6         Socket();
7         ~Socket();
8
9         bool create(); // crea il socket
10        bool bind(int port); // associa il socket ad una porta
11        bool listen() const; // mette il socket in ascolto
12        bool accept(Socket& newSocket) const; // accetta una connessione
              in entrata
13        bool connect(const string& host, int port); // si connette ad un
              server
14        bool close(); // chiude il socket
15
16        bool send(const string& message) const; // invia un messaggio
17        int receive(string& message) const; // riceve un messaggio
18
19        void setNonBlocking(const bool); // setta il socket in modalita'
              non bloccante
20 };
```

- `int m_sock;`: Questo è il file descriptor del socket, che rappresenta un identificatore univoco per il socket aperto nel sistema operativo.
- `sockaddr_in m_addr;`: Struttura che memorizza l'indirizzo IP e la porta del socket.

3.1.1 Socket()

```
1 Socket::Socket() : m_sock(-1) { // inizializza il socket a -1
2     memset(&m_addr, 0, sizeof(m_addr)); // inizializza l'indirizzo a 0
3 }
```

Questa è la definizione del costruttore della classe Socket, che, viene automaticamente chiamato quando viene creato un oggetto della classe Socket. Abbiamo che `m_sock` viene inizializzato con il valore `-1` che, generalmente, indica che il socket non è valido o non è stato ancora

creato correttamente.

`memset(&m_addr, 0, sizeof(m_addr));`

- `memset` è una funzione standard della libreria C che viene utilizzata per impostare tutti i byte di un'area di memoria a un valore specifico, in questo caso `0`.
- `&m_addr` è l'indirizzo della struttura `sockaddr_in` che viene utilizzata per memorizzare l'indirizzo IP e la porta.
- `sizeof(m_addr)` determina la dimensione in byte della struttura `m_addr`.
- Impostando tutta la memoria di `m_addr` a 0, si inizializza l'intera struttura a valori nulli. Questo è utile per evitare valori spazzatura in memoria che potrebbero causare un comportamento imprevisto quando si utilizza questa struttura in seguito.

3.1.2 ~Socket()

Il distruttore è una funzione speciale in C++ che viene chiamata automaticamente quando un oggetto della classe viene distrutto, per esempio quando esce fuori dal suo scope o viene esplicitamente eliminato.

```
Socket::~Socket() { // distruttore per chiudere il socket
    close();
}
```

Quando il distruttore viene invocato, chiama il metodo `close()` per assicurarsi che il socket venga chiuso correttamente, evitando che risorse di sistema rimangano occupate inutilmente.

3.1.3 create()

Il metodo `create()` ha il compito di creare un socket e configurarlo per l'uso.

```
bool Socket::create() { // crea il socket
    m_sock = socket(AF_INET, SOCK_STREAM, 0); // crea un socket TCP
    if (m_sock == -1) { // se la creazione del socket fallisce
        cerr << "Errore nella creazione del socket: " << strerror(errno)
           << endl;
```

```
5           return false; // ritorna false
6       }
7
8       int on = 1; // abilita l'opzione SO_REUSEADDR
9       // imposta l'opzione SO_REUSEADDR per il socket m_sock
10      if (setsockopt(m_sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
            {
11          cerr << "Errore nell'impostazione delle opzioni del socket: " <<
                strerror(errno) << endl;
12          return false;
13      }
14
15      return true;
16  }
```

`m_sock = socket(AF_INET, SOCK_STREAM, 0);`

- Questa chiamata alla funzione `socket()` crea un nuovo socket.
- `AF_INET` indica che il socket utilizzerà il protocollo IPv4.
- `SOCK_STREAM` specifica che si tratta di un socket di tipo TCP, che fornisce un servizio di comunicazione orientato alla connessione.
- Il terzo parametro (`0`) seleziona il protocollo predefinito per il tipo di socket, che è TCP in questo caso.
- La funzione `socket()` restituisce un descrittore di socket, che viene memorizzato nella variabile `m_sock`. Se la creazione del socket fallisce, `m_sock` sarà impostato a `-1`.

`if (setsockopt(m_sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)`

- La funzione `setsockopt()` viene utilizzata per configurare le opzioni del socket.
- `SOL_SOCKET` indica che l'opzione è a livello di socket.

- `SO_REUSEADDR` è l'opzione che permette di riutilizzare gli indirizzi locali (utile per riavviare il server senza dover attendere che il sistema operativo rilasci il porto).
- `&on` è l'indirizzo della variabile che contiene il valore dell'opzione.
- `sizeof(on)` è la dimensione della variabile `on`.
- Se `setsockopt()` fallisce, viene visualizzato un messaggio di errore e il metodo ritorna `false`.

Dettaglio	**Descrizione**
Input	Nessuno
Output	`bool` : Ritorna true se la creazione e configurazione del socket sono avvenute con successo, altrimenti ritorna false in caso di errore.

Table 1: Dettagli del metodo `create()`.

3.1.4 bind()

Il metodo `bind()` ha il compito di associare un socket a una specifica porta sulla macchina locale.

```
1 bool Socket::bind(int port) { // associa il socket ad una porta
2     if (m_sock == -1) {
3         cerr << "Socket non valido. Creare il socket prima di fare il bind
              ." << endl;
4         return false;
5     }
6
7     m_addr.sin_family = AF_INET; // inizializzazione della famiglia dell'
          indirizzo a IPv4
8     m_addr.sin_addr.s_addr = htonl(INADDR_ANY); //Imposta l'indirizzo IP a
           INADDR_ANY per accettare connessioni da qualsiasi interfaccia di
          rete.
9     m_addr.sin_port = htons(port); // inizializzazione della porta del
          server
```

```
10
11      if (::bind(m_sock, (struct sockaddr*)&m_addr, sizeof(m_addr)) < 0) {
            // associa il socket ad un indirizzo e porta
12          cerr << "Errore nel bind del socket: " << strerror(errno) << endl;
13          return false;
14      }
15
16      return true;
17  }
```

Dettaglio	Descrizione
Input	
`int port`	Porta alla quale associare il socket (passato come intero).
Output	
`bool`	Ritorna `true` se l'associazione del socket alla porta è avvenuta con successo. Ritorna `false` se si è verificato un errore durante l'operazione.

Table 2: Dettagli del metodo `bind()`.

3.1.5 listen()

Il metodo `listen()` prepara un socket ad accettare connessioni in entrata. Questa funzione è essenziale per i socket server che devono mettersi in ascolto delle richieste di connessione da parte dei client.

```
1  bool Socket::listen() const { // mette il socket in ascolto
2      if (m_sock == -1) {
3          cerr << "Socket non valido. Creare e fare il bind del socket prima
               di metterlo in ascolto." << endl;
4          return false;
5      }
6
7      if (::listen(m_sock, 1024) < 0) { // inizio dell'ascolto del socket
```

```
8       //il secondo parametro 1024 rappresenta il numero massimo di
            connessioni in coda che possono essere gestite.
9           cerr << "Errore nell'ascolto del socket: " << strerror(errno) <<
                endl;
10          return false;
11      }
12
13      return true;
14  }
```

Dettaglio	Descrizione
Input	
`Nessuno`	Il metodo `listen()` non richiede parametri di input.
Output	
`bool`	Ritorna `true` se il socket è stato messo in ascolto con successo. Ritorna `false` se si è verificato un errore durante l'operazione.

Table 3: Dettagli del metodo `listen()`.

3.1.6 accept()

Il metodo `accept()` accetta una connessione in entrata su un socket in ascolto e crea un nuovo socket per gestire questa connessione. Questo metodo è essenziale per i server che devono ricevere e gestire le richieste di connessione dai client.

```
1   bool Socket::accept(Socket& newSocket) const { // accetta una connessione
        in entrata
2       if (m_sock == -1) {
3           cerr << "Socket non valido. Creare, fare il bind e mettere in
                ascolto il socket prima di accettare connessioni." << endl;
4           return false;
5       }
6
7       socklen_t addr_length = sizeof(m_addr); //Determina la dimensione
            della struttura che conterra' l'indirizzo del client.
```

```
8       newSocket.m_sock = ::accept(m_sock, (sockaddr*)&m_addr, &addr_length);
9       // accetta la connessione in entrata e restituisce un nuovo socket per
            gestirla
10
11      if (newSocket.m_sock <= 0) { // se la connessione non e' stata
            accettata
12          cerr << "Errore nell'accettare la connessione: " << strerror(errno
                ) << endl;
13          return false;
14      } else {
15          return true;
16      }
17  }
```

Dettaglio	Descrizione
Input	
`Socket& newSocket`	Oggetto di tipo `Socket` passato per riferimento. Al termine della chiamata, questo oggetto conterrà il nuovo socket per la connessione accettata.
Output	
`bool`	Ritorna `true` se la connessione è stata accettata con successo e un nuovo socket è stato creato per gestirla. Ritorna `false` se si è verificato un errore durante l'accettazione della connessione.

Table 4: Dettagli del metodo `accept()`.

3.1.7 connect()

Il metodo `connect()` stabilisce una connessione a un server utilizzando un socket. Questa funzione è essenziale per i client che devono collegarsi a un server per comunicare.

```
bool Socket::connect(const string& host, int port) { // si connette ad un server
    if (m_sock == -1) {
        cerr << "Socket non valido. Creare il socket prima di connettersi." << endl;
        return false;
    }

    m_addr.sin_family = AF_INET; // inizializzazione della famiglia dell'indirizzo a IPv4
    m_addr.sin_port = htons(port); // inizializzazione della porta del server

    if (inet_pton(AF_INET, host.c_str(), &m_addr.sin_addr) <= 0) { // converte l'indirizzo IP da stringa a binario e lo memorizza in m_addr.sin_addr
        cerr << "Errore nella conversione dell'indirizzo IP: " << strerror(errno) << endl;
        return false;
    }

    if (::connect(m_sock, (sockaddr*)&m_addr, sizeof(m_addr)) < 0) { // connessione al server
        cerr << "Errore nella connessione al server: " << strerror(errno) << endl;
        return false;
    }

    return true;
}
```

Dettaglio	Descrizione
Input	
`const string& host`	Indirizzo IP del server a cui ci si vuole connettere (passato come stringa).
`int port`	Porta del server a cui ci si vuole connettere (passato come intero).
Output	
`bool`	Ritorna `true` se la connessione è stata stabilita con successo. Ritorna `false` se si è verificato un errore durante la connessione.

Table 5: Dettagli del metodo `connect()`.

3.1.8 send()

Il metodo `send()` è utilizzato per inviare un messaggio attraverso un socket.

```
1  bool Socket::send(const string& message) const { // invia un messaggio
2      if (m_sock == -1) {
3          cerr << "Socket non valido. Creare il socket prima di inviare
               messaggi." << endl;
4          return false;
5      }
6
7      if (::send(m_sock, message.c_str(), message.size(), MSG_NOSIGNAL) < 0)
           { // invia il messaggio attraverso il socket
8          cerr << "Errore nell'invio del messaggio: " << strerror(errno) <<
               endl;
9          return false;
10     } else {
11         return true;
12     }
13 }
```

..send(m_sock,message.c_str (),message.size() ,MSG_NOSIGNAL)

- `m_sock`: Il file descriptor del socket attraverso cui il messaggio è inviato. Deve essere un socket valido e aperto.

- `message.c_str()`: Fornisce un puntatore al buffer di dati da inviare, ottenuto dalla stringa `message`. La funzione `c_str()` restituisce un puntatore a una stringa C, che è compatibile con `send()`.
- `message.size()`: Indica la lunghezza del messaggio in byte. La funzione `size()` della stringa restituisce la quantità di dati da inviare.
- `MSG_NOSIGNAL`: Un flag che previene la generazione di un segnale `SIGPIPE` se il socket è chiuso durante l'invio.

Dettaglio	**Descrizione**
Input	`const string& message`: Il messaggio da inviare attraverso il socket, passato come stringa.
Output	`bool`: Ritorna `true` se il messaggio è stato inviato con successo. Ritorna `false` se si è verificato un errore durante l'invio.

Table 6: Dettagli del metodo `send()`.

3.1.9 receive()

La funzione `receive()` ha il compito di ricevere un messaggio attraverso un socket e di memorizzarlo in una stringa.

```
1  int Socket::receive(string& message) const { // riceve un messaggio
2      if (m_sock == -1) {
3          cerr << "Socket non valido. Creare il socket prima di ricevere
               messaggi." << endl;
4          return -1;
5      }
6
7      char buffer[1024]; // buffer per memorizzare il messaggio
8      message.clear(); // pulisce il messaggio
9      memset(buffer, 0, sizeof(buffer)); // inizializza il buffer a 0
10
11     int status = ::recv(m_sock, buffer, sizeof(buffer), 0); // riceve il
           messaggio attraverso il socket
```

```
12      if (status < 0) { // se la ricezione del messaggio fallisce
13          cerr << "Errore nella ricezione del messaggio: " << strerror(errno
                ) << endl;
14          return -1;
15      } else if (status == 0) { // se la connessione e' stata chiusa
16          cerr << "Connessione chiusa dal peer." << endl;
17          return 0;
18      } else { // se la ricezione del messaggio ha successo
19          message.assign(buffer, status); // assegna il messaggio del buffer
20          return status; // ritorna la lunghezza del messaggio
21      }
22  }
```

La funzione `recv()` tenta di ricevere i dati dal socket e li memorizza nel buffer. Restituisce il numero di byte ricevuti o un valore negativo in caso di errore. In particolare, la stringa message viene riempita con i dati contenuti nel `buffer`, limitandosi ai primi `status` byte.

Dettaglio	**Descrizione**
Input	
`string& message`	Stringa in cui viene memorizzato il messaggio ricevuto dal socket.
Output	
`int`	Restituisce il numero di byte ricevuti, `0` se la connessione è chiusa, `-1` in caso di errore.

Table 7: Dettagli del metodo `receive()`.

3.1.10 close()

Il metodo `close()` si occupa di chiudere il socket associato all'oggetto `Socket` e gestisce eventuali errori che possono verificarsi durante questa operazione.

```
1   bool Socket::close() { //chiude la socket
2       //Tenta di chiudere il socket usando la funzione di sistema
3       if (::close(m_sock) < 0) { //se la chiusura del socket fallisce
4           cerr << "Errore nella chiusura del socket: " << strerror(errno) <<
                endl;
```

```
        return false;
    }

    m_sock = -1; // Ripristina la descrizione del socket

    // Controllare se ci sono stati errori specifici durante la chiusura
    if (errno == EINTR) {
        cerr << "Chiusura del socket interrotta da un segnale." << endl;
    } else if (errno == EBADF) {
        cerr << "Descrizione del socket non valida." << endl;
    } else if (errno == EIO) {
        cerr << "Errore I/O durante la chiusura del socket." << endl;
    }

    return true;
}
```

Dettaglio	**Descrizione**
Input	
Nessuno	Il metodo non richiede parametri di input.
Output	
`bool`	Ritorna `true` se il socket è stato chiuso con successo. Ritorna `false` se si è verificato un errore durante la chiusura del socket.

Table 8: Dettagli del metodo `close()`.

3.1.11 setNonBlocking()

Il metodo `setNonBlocking()` permette di configurare il socket in modalità non bloccante o bloccante, a seconda del valore del parametro `b`. In modalità non bloccante, le operazioni di I/O non costringono il programma a fermarsi in attesa dell'esito.

```
void Socket::setNonBlocking(const bool b) { // setta il socket in modalita
    ' non bloccante
    int opts = fcntl(m_sock, F_GETFL); // recupera le opzioni del socket
```

```
3
4       if (opts < 0) { //se non riesco a leggere le impostazioni del socket
5           cerr << "Errore nel recupero delle opzioni del socket: " <<
                strerror(errno) << endl;
6           return;
7       }
8
9       if (b) { // se b e' vero
10          opts |= O_NONBLOCK; // setta il socket in modalita non bloccante
11      }
12
13      else {
14          opts &= ~O_NONBLOCK; // setta il socket in modalita' bloccante
15      }
16
17      if (fcntl(m_sock, F_SETFL, opts) < 0) { // Applica le nuove
            impostazioni al socket
18          cerr << "Errore nell'impostazione delle opzioni del socket: " <<
                strerror(errno) << endl;
19      }
20  }
```

`fcntl(m_sock, F_SETFL, opts)` è una chiamata di sistema che permette di modificare le opzioni del file descriptor `m_sock`. In questo contesto, viene utilizzata per impostare le modalità operative del socket.

- `m_sock`: Il file descriptor del socket su cui applicare le modifiche.
- `F_SETFL`: Il comando che indica a `fcntl()` di impostare le opzioni del file descriptor, sovrascrivendo le opzioni precedenti.
- `opts`: Le nuove opzioni da applicare al socket, ad esempio `O_NONBLOCK` per la modalità non bloccante.

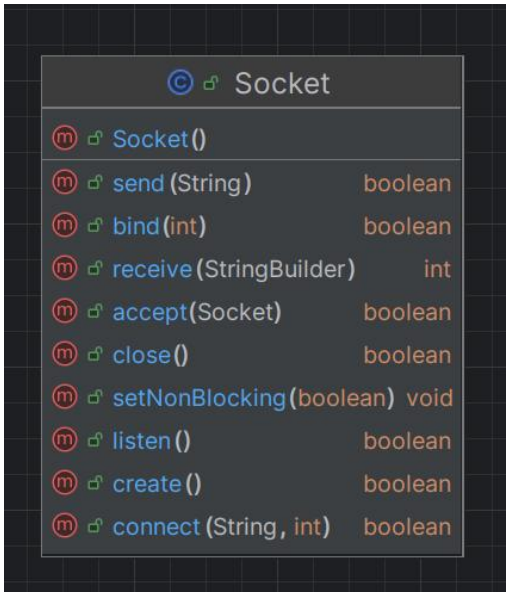
Se la chiamata ha successo, il socket opererà con le nuove opzioni specificate; altrimenti, verrà restituito un errore.

Dettaglio	Descrizione
Input	
`const bool b`	`true` per impostare il socket in modalità non bloccante, `false` per modalità bloccante.
Output	
`void`	Il metodo non restituisce un valore, ma modifica lo stato del socket.

Table 9: Dettagli del metodo `setNonBlocking()`.

3.1.12 Conclusioni Finali

La classe `Socket` rappresenta una soluzione robusta e versatile per la gestione della comunicazione attraverso i socket, sia lato client che server. Come descritto in questa relazione, la classe include una serie di metodi fondamentali per creare, configurare, gestire e chiudere i socket, nonché per inviare e ricevere messaggi. Questa dualità di utilizzo dimostra la flessibilità della classe e la sua applicabilità a una vasta gamma di scenari di rete. L'implementazione dei metodi `create()`, `bind()`, `listen()` e `accept()` è particolarmente rilevante per i server, che devono gestire le connessioni in modo efficiente. D'altra parte, i metodi `connect()`, `send()` e `receive()` sono essenziali per i client, che richiedono una comunicazione fluida e affidabile con i server.

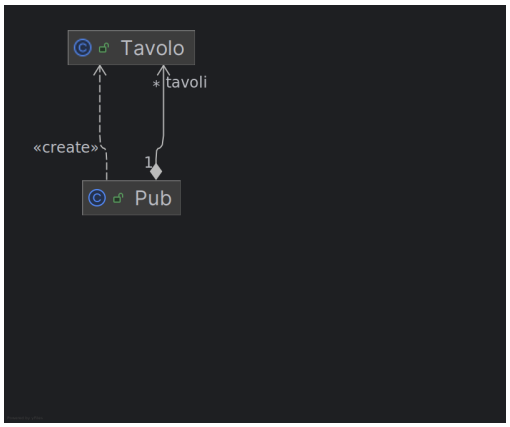


3.2 Server : Pub

In questo capitolo, ci concentreremo sull'analisi della prima delle tre componenti fondamentali del nostro progetto: il *Pub*, che in questo contesto funge da server. Il *Pub* è una delle tre entità centrali del sistema, incaricata di gestire la comunicazione e fornire tutte le risposte necessarie al *Cameriere*, che rappresenta il client in questo specifico caso.

Il server *Pub* è strutturato in due file principali: `Pub.h` e `Tavolo.h`. Questi file delineano rispettivamente le due entità fondamentali del pub, ciascuna dotata di funzioni specifiche che verranno analizzate nel dettaglio nelle sezioni successive. La classe definita in `Pub.h` rappresenta il cuore del server, gestendo le operazioni globali del pub, mentre la classe in `Tavolo.h` modella le interazioni e lo stato dei singoli tavoli all'interno del sistema.

Oltre a questi file, il server include un `main.cpp`, nel quale viene gestita la connessione con il *Cameriere* e avviata la comunicazione. È importante sottolineare che il *Pub* non comunica mai direttamente con il cliente finale.



3.2.1 Pub.h

La classe `Pub` rappresenta il cuore del sistema server, modellando l'entità centrale del Pub all'interno del nostro progetto. Questa classe è responsabile della gestione complessiva del locale, inclusa l'organizzazione dei tavoli e la gestione dei clienti.

`Pub` è progettata per gestire dinamicamente i tavoli e i clienti attraverso una serie di metodi che facilitano l'aggiunta di nuovi tavoli, la verifica della disponibilità di posti, la gestione degli ordini e la liberazione dei posti occupati. La classe utilizza una struttura

interna per memorizzare e gestire le informazioni relative ai tavoli, il numero massimo di clienti e tavoli disponibili, e l'identificatore della memoria condivisa.


```
/*
 Classe Pub per rappresentare il Pub
*/

#ifndef PUB_H
#define PUB_H
```

```
7
8  #include <iostream>
9  #include "Tavolo.h"
10 using namespace std;
11
12 class Pub {
13     private:
14
15         int maxClienti; // numero massimo di cliente che possono entrare
               nel locale
16         int maxTavoli; // numero massimo di tavoli disponibili
17         int ntavoli; // numero di tavolo presenti nel Pub
18         int shmID; // identificatore memoria condivisa
19         Tavolo* tavoli; // lista dei tavoli nel locale
20
21     public:
22
23         // Creo un Pub con il numero massimo di clienti e di tavoli
24         Pub(int maxClienti, int maxTavoli);
25         ~Pub(); // Distruttore per il dettach della memoria condivisa
26         bool aggiungiTavolo(int maxSedie); // metodo per aggiungere un
               tavolo nel locale
27         bool postoDisponibile(int numeroTavolo); // metodo per verificare
               se ci sono posti nel tavolo scelto
28         bool aggiungiCliente(int numeroTavolo); // metodo per aggiungere
               il cliente al tavolo scelto
29         void preparaOrdine(int tavolo); // metodo per preparare l'ordine
               del cliente
30         int tavoloVuoto(); //verifica se il tavolo e' vuoto
31         void liberaPosto(int numeroTavolo); // metodo per liberare un
               posto al tavolo
32         int postiDisponibili(); // verifica se ci sono posti liberi nel
               Pub
33 };
34
35 #endif // PUB_H
```

Pub()-Costruttore Il costruttore della classe `Pub` è responsabile dell'inizializzazione dell'oggetto `Pub`, in particolare della creazione e gestione della memoria condivisa per un array di oggetti `Tavolo`.

```
/* Costruttore per inizializzare l'array di tavoli in memoria condivisa e
   impostare il numero massimo di clienti, il numero massimo di tavoli
   e inizizializzare ntavoli */
Pub::Pub(int maxClienti, int maxTavoli) : maxClienti(maxClienti),
   maxTavoli(maxTavoli), ntavoli(1) {

    // Creazione/accesso alla memoria condivisa per l'array tavoli
    if ((this->shmID = shmget(IPC_PRIVATE, sizeof(Tavolo), IPC_CREAT |
       0666)) < 0) {
        cerr << "Errore durante la shmget per Tavolo" << endl;
        exit(1);
    }

    // Attacco della memoria condivisa
    this->tavoli = (Tavolo *) shmat(shmID, NULL, 0);
    if (tavoli == (Tavolo*)-1) {
        cerr << "Errore durante la shmat per tavoli" << endl;
        exit(1);
    }
}
```

- `maxClienti(maxClienti)`: Il costruttore prende un parametro `maxClienti` e lo assegna al membro della classe `maxClienti`. Questo parametro rappresenta il numero massimo di clienti che possono entrare nel pub.
- `maxTavoli(maxTavoli)`: Analogamente, il parametro `maxTavoli` viene assegnato al

membro della classe `maxTavoli`. Rappresenta il numero massimo di tavoli che possono essere presenti nel pub.

- `ntavoli(1)`: Il membro della classe `ntavoli` viene inizializzato a 1, indicando che il pub parte con un solo tavolo disponibile.

Il costruttore prosegue con la creazione e l'attacco della memoria condivisa, che viene utilizzata per gestire l'array di tavoli nel pub.

È ovviamente necessario andare a comprendere meglio e motivare la scelta dell'utilizzo della **memoria condivisa** in questo contesto. La **memoria condivisa** viene inizializzata per consentire la condivisione di dati tra diversi **processi** o **thread** che partecipano alla gestione del pub. Nel progetto in questione, essendo un **client/server parallelo**, questa tecnica è essenziale perché permette di avere un accesso comune a risorse critiche, come l'**array di tavoli**, da parte di più entità.

Inizializzare la **memoria condivisa** consente a questi processi di **comunicare** e **coordinarsi** efficacemente, assicurando che le informazioni riguardanti lo stato dei tavoli (come la disponibilità e gli ordini) siano sempre aggiornate e accessibili da tutti i partecipanti. Questa condivisione di memoria è particolarmente utile in un contesto in cui diversi processi devono lavorare insieme in modo **sincrono** e **coordinato**, evitando problemi di inconsistenza e garantendo un'esperienza fluida e corretta per la gestione del pub.

Il tutto viene gestito tramite:

- `shmget`: Questa funzione di sistema viene utilizzata per creare o ottenere l'accesso a un segmento di memoria condivisa. I parametri utilizzati sono:
  - `IPC_PRIVATE`: Indica che la memoria condivisa è privata e non viene condivisa con altri processi.
  - `sizeof(Tavolo)`: Specifica la dimensione del segmento di memoria che sarà sufficiente per contenere un oggetto `Tavolo`.
  - `IPC_CREAT | 0666`: I flag `IPC_CREAT` e `0666` specificano che il segmento deve essere creato se non esiste già, con permessi di lettura e scrittura per tutti gli utenti.

La funzione in questione restituisce **shmID** che sarebbe L'identificatore della memoria condivisa ottenuto da *shmget*. Non resta dunque che effettuare l'`attacco` alla memoria condivisa tramite `shmat` che prende come parametri:

- `shmID`: L'identificatore della memoria condivisa ottenuto da `shmget`.
- `NULL`: Indica che il sistema può scegliere l'indirizzo dove mappare la memoria condivisa.
- `0`: Flag che indica che non ci sono opzioni speciali (come il solo accesso in lettura).

Distruttore `Pub::~Pub()` Il distruttore `Pub::~Pub()` è utilizzato per liberare le risorse allocate durante la vita dell'oggetto `Pub`, ed è fondamentale per gestire correttamente la **memoria condivisa** utilizzata dalla classe `Pub`.

```
Pub::~Pub(){
    // Dettach della memoria condivisa
    shmdt(tavoli);
    shmctl(shmID, IPC_RMID, NULL);
}
```

- **Dettach della memoria condivisa:** La funzione `shmdt(tavoli)` viene chiamata per scollegare il segmento di memoria condivisa dall'indirizzo di spazio del processo corrente. Questo è essenziale per assicurarsi che la memoria non rimanga inutilmente collegata dopo la terminazione dell'oggetto.
- **Rimozione del segmento di memoria condivisa:** La funzione `shmctl(shmID, IPC_RMID, NULL)` viene utilizzata per marcare il segmento di memoria condivisa per la rimozione. Questo significa che il segmento sarà eliminato quando non ci saranno più processi che lo utilizzano, liberando così la memoria.

aggiungiTavolo(...) La funzione `Pub::aggiungiTavolo(int maxSedie)` è progettata per aggiungere un nuovo tavolo all'interno del pub.

```
bool Pub::aggiungiTavolo(int maxSedie) {
    if (ntavoli <= maxTavoli) { //Se non ho raggiunto il massimo di tavoli
        disponibili
        new (&tavoli[ntavoli]) Tavolo(maxSedie); //aggiungo un nuovo
           tavolo

        //incremento il numero di tavoli
        ntavoli++;

        return true; // Restituisce true se il tavolo e' stato aggiunto
    }

    return false; // Restituisce false se il tavolo non e' stato aggiunto
}
```

Dettaglio	**Descrizione**
Input	
`int maxSedie`	Numero massimo di sedie per il nuovo tavolo.
Output	
`bool`	Ritorna `true` se il tavolo è stato aggiunto con successo. Ritorna `false` se il numero massimo di tavoli è stato raggiunto e il tavolo non è stato aggiunto.

Table 10: Dettagli della funzione `aggiungiTavolo`.

aggiungiCliente(...) La funzione `Pub::aggiungiCliente(int numeroTavolo)` è responsabile dell'aggiunta di un cliente al tavolo specificato all'interno del pub.

```
bool Pub::aggiungiCliente(int numeroTavolo) {
    if (tavoli[numeroTavolo].addCliente()) { //In base al numero di tavolo
        aggiungo il cliente
        return true; // restituisce vero se e' andato a buon fine la
           procedura
```

```
    }
    return false; // altrimenti restituisce falso per un errore
}
```

Dettaglio	Descrizione
Input	
`int numeroTavolo`	Indica il numero del tavolo al quale aggiungere un cliente.
Output	
`bool`	Ritorna `true` se il cliente è stato aggiunto con successo al tavolo. Ritorna `false` se non è stato possibile aggiungere il cliente, ad esempio se il tavolo è pieno.

Table 11: Dettagli della funzione `aggiungiCliente`.

tavoloVuoto() La funzione `Pub::tavoloVuoto()` è utilizzata per determinare se ci sono tavoli vuoti all'interno del pub e restituire il numero del primo tavolo vuoto trovato. (Valido per la richiesta del cliente di accomodarsi ad un nuovo tavolo).

```
int Pub::tavoloVuoto(){
    for(int i = 1; i <= ntavoli; i++){ // Scorro tutti i tavoli
        if(tavoli[i].tavoloVuoto()){ //Se il tavolo e' vuoto
            return i; // restituisce il numero di tavolo
        }
    }

    return -1; // altrimenti restituisce un numero negativo per segnalare
       che non ci sono tavoli vuoti
}
```

Dettaglio	Descrizione
Input	
Nessuno	La funzione non richiede parametri di input.
Output	
`int`	Ritorna l'indice del primo tavolo vuoto trovato. Se non ci sono tavoli vuoti, ritorna `-1`.

Table 12: Dettagli della funzione `tavoloVuoto`.

liberaPosto(...) La funzione `Pub::liberaPosto()` è utilizzata per liberare un posto specifico in un tavolo del pub.

```
void Pub::liberaPosto(int numeroTavolo) {
        tavoli[numeroTavolo].liberaPosto(); //libera il posto al numero
           del tavolo
}
```

Dettaglio	Descrizione
Input	
`int numeroTavolo`	L'indice del tavolo dal quale liberare un posto.
Output	
Nessuno	La funzione non restituisce alcun valore. Modifica lo stato del tavolo specificato per riflettere che un posto è stato liberato.

Table 13: Dettagli della funzione `liberaPosto`.

preparaOrdine(...) La funzione `Pub::preparaOrdine()` simula il processo di preparazione di un ordine per un tavolo specifico.

```
void Pub::preparaOrdine(int ntavolo) {
    cout << "Sto preparando l'ordine del tavolo " << ntavolo << endl; //
       stampa per simulare l'ordine di preparazione
    sleep (5); //finta attesa
    cout << "Ordine pronto per essere consegnato al tavolo " << ntavolo <<
        " dal cameriere" << endl; //stampa per simulare l'ordine pronto
}
```

Dettaglio	Descrizione
Input	
`int ntavolo`	L'indice del tavolo per il quale l'ordine viene preparato.
Output	
Nessuno	La funzione non restituisce alcun valore. Simula il processo di preparazione dell'ordine con stampe a video e una pausa temporale.

Table 14: Dettagli della funzione `preparaOrdine`.

postiDisponibili() La funzione `Pub::postiDisponibili()` calcola e restituisce il numero totale di posti disponibili nei tavoli del pub.

```
int Pub::postiDisponibili() {
    int posti = 0; //variabile per contare il numero di posti disponibili
    for (int i = 1; i <= ntavoli; i++) { // Scorre tutti i tavoli
        //calcola il numero di posti liberi in base al numero massimo di
           sedie disponibili
        posti += (tavoli[i].getMaxSedieTavolo() - tavoli[i].
           getNumeroClienti());
    }
    return posti; // ritorna il numero di posti disponibili
}
```

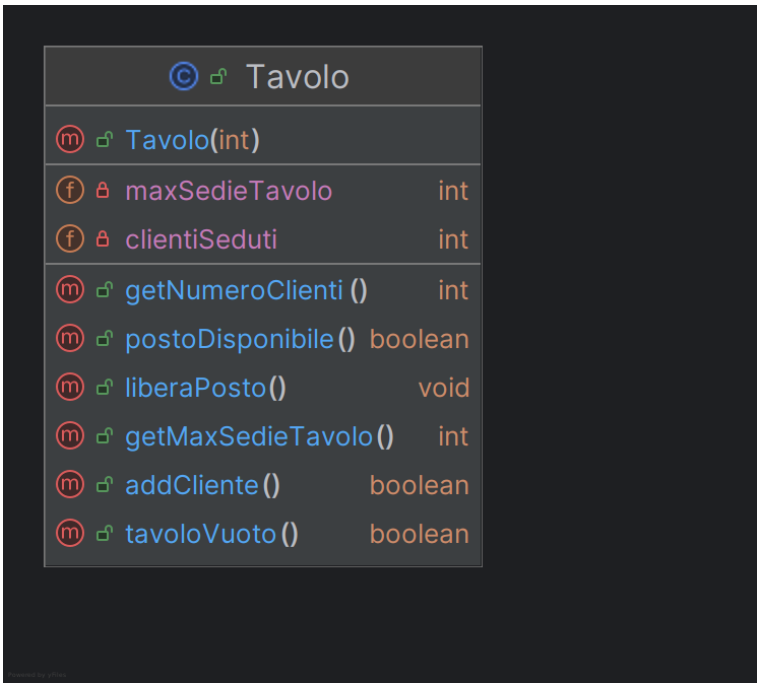
Per ogni tavolo, la funzione calcola il numero di posti liberi sottraendo il numero di clienti presenti dal numero massimo di sedie disponibili. Questo valore viene poi aggiunto alla variabile `posti`.

Dettaglio	Descrizione
Input	
Nessuno	La funzione non richiede parametri di input.
Output	
`int`	Ritorna un valore intero che rappresenta il numero totale di posti disponibili in tutti i tavoli del pub.

Table 15: Dettagli della funzione `postiDisponibili`.

3.2.2 Tavolo.h

Nel contesto del progetto, la classe `Tavolo` è fondamentale per la gestione e la rappresentazione dei tavoli all'interno del pub. Questa classe definisce le caratteristiche e le operazioni che possono essere effettuate sui tavoli, come il numero massimo di sedie disponibili e il numero di clienti attualmente seduti, essenziali per il corretto funzionamento del sistema di gestione del locale.




```
/*
 Classe Tavolo per rappresentare i tavoli all'interno del Pub
*/

#ifndef TAVOLO_H
```

```
#define TAVOLO_H

class Tavolo{
    private:

        int maxSedieTavolo; // numero massimo di sedie disponibili per
           tavolo
        int clientiSeduti; // numero di clienti seduti al tavolo

    public:

        //Crea un nuovo tavolo con un numero di sedie disponibili
        Tavolo(int maxSedie);
        bool addCliente(); // aggiunge un cliente al tavolo
        bool postoDisponibile(); // verifica se il tavolo non e' pieno
        void liberaPosto(); //libera il posto al tavolo
        int getNumeroTavolo(); // restituisce il numero del tavolo
        int getNumeroClienti(); // restituisce il numero di clienti seduti
            al tavolo
        int getMaxSedieTavolo(); // restituisce massimo di posti
           disponibile al tavolo
        bool tavoloVuoto(); // verifica se il tavolo e' vuoto
};
```

Costruttore : Tavolo(...) Il costruttore della classe `Tavolo` ha il compito di inizializzare i membri della classe per rappresentare correttamente lo stato iniziale di un tavolo nel pub.

```
// Costruttore per inizializzare clientiSeduti e impostare il numero
   massimo di sedie
Tavolo::Tavolo(int maxSedie) : maxSedieTavolo(maxSedie), clientiSeduti(0)
   {}
```

addCliente() La funzione `addCliente()` della classe `Tavolo` si occupa di aggiungere un cliente al tavolo, se vi sono posti disponibili.

```
bool Tavolo::addCliente(){
    if(postoDisponibile()){ // Se ci sono posti disponibili
        clientiSeduti++; // aggiungo il cliente incrementando il contatore
        return true; // restituisce vero se ci sono posti disponibili
    }

    return false; // restituisce falso se non ci sono posti disponibili
}
```

È la funzione utilizzata in 11*(aggiungiCliente)* della classe Pub.

Dettaglio	**Descrizione**
Input	Nessuno
Output	`bool`
Descrizione Output	Ritorna `true` se il cliente è stato aggiunto con successo, altrimenti `false` se il tavolo è pieno.

Table 16: Dettagli della funzione `addCliente()`.

postoDisponibile() La funzione `postoDisponibile()` verifica se ci sono posti disponibili al tavolo, confrontando il numero di clienti seduti con il numero massimo di sedie disponibili. Se il numero di `clientiSeduti` è inferiore a `maxSedieTavolo`, la funzione restituisce `true`, indicando che c'è ancora spazio per altri clienti. Altrimenti, restituisce `false`.

```
bool Tavolo::postoDisponibile(){
    return clientiSeduti < maxSedieTavolo; //restituisce true se il tavolo
        non e' pieno altrimenti restituisce false
}
```

Dettaglio	Descrizione
Input	Nessuno
Output	`bool`
Descrizione Output	Restituisce `true` se il tavolo non è pieno, altrimenti `false`.

Table 17: Dettagli della funzione `postoDisponibile()`.

liberaPosto() La funzione `liberaPosto()` viene utilizzata per liberare un posto al tavolo, decrementando il contatore `clientiSeduti`. Prima di effettuare questa operazione, la funzione verifica che il tavolo non sia già vuoto, controllando se `clientiSeduti` è maggiore di 0. Se il tavolo non è vuoto, decrementa il contatore di uno.

```
void Tavolo::liberaPosto(){
    if(clientiSeduti > 0){ // Se il tavolo non e' vuoto
        clientiSeduti--; // rimuove il cliente decrementando il contatore
    }
}
```

È la funzione utilizzata in 13*(liberaPosto)* della classe Pub.

Dettaglio	Descrizione
Input	Nessuno
Output	Nessuno
Descrizione	Decrementa il numero di clienti seduti al tavolo, se `clientiSeduti` è maggiore di 0.

Table 18: Dettagli della funzione `liberaPosto()`.

getNumeroClienti() La funzione `getNumeroClienti()` non è altro che una funzione **getter**, viene utilizzata per ottenere il numero di clienti attualmente seduti al tavolo. Questa funzione ritorna semplicemente il valore del membro `clientiSeduti` della classe, che tiene traccia del numero di clienti presenti.

```
int Tavolo::getNumeroClienti(){
    return this->clientiSeduti; // restituisce il numero di clienti seduti
        al tavolo
}
```

È la funzione utilizzata in 15*(postiDisponibili)* della classe Pub.

tavoloVuoto() Questa funzione è un metodo della classe `Tavolo` che verifica se un tavolo è vuoto, ovvero se non ci sono clienti seduti. La funzione ritorna `true` se il numero di `clientiSeduti` è uguale a zero, indicando che il tavolo è vuoto; altrimenti, ritorna `false`.

```
bool Tavolo::tavoloVuoto(){
    return this->clientiSeduti == 0; // restituisce true se il tavolo e'
       vuoto, altrimenti false
}
```

È la funzione utilizzata in 12*(tavoloVuoto)* della classe Pub.

getMaxSedieTavolo() Questa funzione è un metodo *getter* della classe `Tavolo` che restituisce il numero massimo di sedie disponibili al tavolo. Il valore restituito è memorizzato nel membro privato `maxSedieTavolo`.

```
int Tavolo::getMaxSedieTavolo(){  //restituisce massimo di posti
   disponibile al tavolo
    return this->maxSedieTavolo;
}
```

3.2.3 main.cpp

Il file `main.cpp` rappresenta il punto di ingresso principale per l'applicazione del server del pub, gestendo l'inizializzazione e la gestione del server stesso. Questo codice è responsabile di diverse operazioni chiave, tra cui la configurazione della **memoria condivisa**, la gestione delle **socket** e l'elaborazione delle richieste dei clienti. In particolare, il codice si occupa di configurare il server per accettare connessioni da clienti (camerieri) e gestire le richieste attraverso **processi figli**, ognuno dedicato a un client specifico.

Dettaglio	Descrizione
Creazione e Attacco Memoria Condivisa	Utilizza `shmget()` e `shmat()` per creare e accedere a un segmento di memoria condivisa per l'oggetto `Pub`.
Inizializzazione del Pub	Crea un'istanza dell'oggetto `Pub` e inizializza con un numero predefinito di tavoli e posti.
Creazione e Configurazione della Socket	Configura la socket del server per accettare connessioni sulla porta 25564 e la mette in ascolto.
Gestione Connessioni Client	Accetta connessioni dai client, crea processi figli per gestire ciascuna connessione, e gestisce le operazioni sui tavoli del pub.
Signal Handling	Implementa un gestore di segnali per gestire l'interruzione del programma e liberare risorse.

Table 19: Dettagli delle operazioni nel file `main.cpp`.

Librerie utilizzate

```
1
2 #include <iostream>    // Gestisce l'input e l'output standard (cin, cout,
      cerr).
3 #include <cstdlib>     // Include funzioni di utilita' come exit() per
    terminare il programma e altre funzioni generiche.
4 #include <string>      // Fornisce la classe string per la gestione delle
    stringhe.
5 #include <signal.h>    // Permette di gestire i segnali del sistema
    operativo, come SIGINT.
6 #include <sys/ipc.h>   // Definisce le costanti e le strutture per la
    gestione della memoria condivisa IPC.
7 #include <sys/shm.h>   // Fornisce le funzioni per creare, attaccare e
    controllare la memoria condivisa.
8 #include <sys/types.h> // Include definizioni di tipo generico, come pid_t
    .
9 #include <unistd.h>    // Fornisce funzioni per le operazioni di sistema,
    come fork() e sleep().
10 #include "Pub.h"       // Include la definizione della classe Pub, che
    gestisce i tavoli e le operazioni del pub.
```

```
#include "../Socket/Socket.h" // Include la definizione della classe
    Socket, utilizzata per la comunicazione di rete.
```

Gestione dei segnali La gestione dei segnali è un aspetto cruciale nella gestione dei processi e delle risorse in un'applicazione server. Nel file main.cpp, sono state implementate funzioni specifiche per gestire i segnali e le disconnessioni in modo che il server possa operare in modo robusto e sicuro.

```
// Signal handler per l'interruzione del programma
void signalHandler(int signum) {
    if (serverSocketPtr != nullptr) { // se ho il contenuto della socket
         clientSocketPtr->send("termine_pub");

        // chiudo i socket
        clientSocketPtr->close();
        serverSocketPtr->close();
        cout << "Server socket closed successfully." << endl; // Stampa di
            successo
    }

    // Dettach della memoria condivisa
    shmdt(pub);
    exit(signum); // esco dal programma in base al codice passato
}
```

La funzione `signalHandler` è un gestore di segnali (*signal handler*) utilizzato per gestire eventi di interruzione del programma, come l'interruzione dell'esecuzione (ad esempio, quando l'utente invia un segnale di terminazione come `SIGINT` tramite Ctrl+C).

Se il puntatore alla socket (`serverSocketPtr`) è valido, viene inviato un messaggio per segnalare la terminazione del server e poi la socket viene chiusa. Questo passaggio garantisce che tutte le comunicazioni in corso siano completate e la connessione sia chiusa in modo sicuro. La memoria condivisa è dissociata utilizzando `shmdt`. Questo è essenziale per liberare la

memoria e impedire perdite o corruzioni di dati. Infine, il programma termina con il codice del segnale ricevuto. Questo permette di gestire l'uscita in modo controllato e fornire un codice di uscita appropriato.

Gestione delle disconnessioni La funzione `termine_pub` è progettata per gestire situazioni in cui un cameriere o un cliente si disconnette improvvisamente durante l'esecuzione del programma. Questo è particolarmente importante in un ambiente multi-cliente per garantire che le risorse siano gestite e liberate correttamente in caso di disconnessione non prevista.

```
1  void termine_pub(Socket *client, string *message, int ntavolo, Pub *pub) {
2
3      // Se il cameriere termina improvvisamente di funzionare
4      if (message->compare("termine_cameriere") == 0) {
5          cout << "Il cameriere si e' disconnesso improvvisamente!" << endl;
               // Stampa di errore
6          pub->liberaPosto(ntavolo); // libero il posto al numero di tavolo
7          cout << "Si e' liberato il posto al tavolo n: "  << ntavolo <<
               endl; // Stampa di avviso
8          client->close(); // chiusura della socket client
9          exit(0); // Uscita dal programma
10     }
11
12     // Se il cliente termina improvvisamente di funzionare
13     else if (message->compare("termine_cliente") == 0) {
14         cout << "Un cliente si e' disconnesso improvvisamente!" << endl;
               // Stampa di errore
15         pub->liberaPosto(ntavolo); // libero il posto al numero di tavolo
16         cout << "Si e' liberato il posto al tavolo n: "  << ntavolo <<
               endl; // Stampa di avviso
17         exit(0); // Uscita dal programma
18     }
19 }
```

Se il messaggio ricevuto indica che il **cameriere** si è disconnesso, la funzione stampa un

messaggio di errore, libera il posto al tavolo utilizzato dal cameriere e chiude la socket associata al client. Infine, termina il programma. Se il messaggio indica che un **cliente** si è disconnesso, la funzione esegue un'azione simile a quella per la disconnessione del cameriere: stampa un messaggio di errore, libera il posto al tavolo e termina il programma.

Gestione della Memoria Condivisa e della Socket Questa parte iniziale riguarda la configurazione della memoria condivisa e della socket per un'applicazione server.

```
//VARIABILI GLOBALI
// Socket globale per il signal handling
Socket* serverSocketPtr = nullptr;
Socket* clientSocketPtr = nullptr;
Pub* pub = nullptr; // Puntatore alla classe Pub per la memoria condivisa

int main() {
    int shmid; // Identificatore della memoria condivisa
    Socket serverSocket; // dichiarazione della socket server


    // Creazione/accesso alla memoria condivisa
    if ((shmid = shmget(IPC_PRIVATE, sizeof(Pub), IPC_CREAT | 0666)) < 0)
       {
        cerr << "Errore durante la shmget per Pub" << endl;
        exit(1); // In caso di errore esco dal programma
    }

    // Attacco della memoria condivisa
    if ((pub = (Pub *)shmat(shmid, NULL, 0)) == (Pub *)-1) {
        cerr << "Errore durante la shmat per Pub" << endl;
        exit(1);
    }
```

Inizializzazione e configurazione del Pub Ora ci occuperemo dell'inizializzazione e configurazione dell'oggetto Pub e della gestione dei tavoli all'interno del pub.

```
// Inizializzazione del pub con 5 tavoli e un massimo di 20 posti
    new (pub) Pub(20, 5);

    // aggiunta dei tavoli
    if(!pub->aggiungiTavolo(4)){
        cerr << "Errore durante l'aggiunta dei tavoli";
    }

    if(!pub->aggiungiTavolo(6)){
        cerr << "Errore durante l'aggiunta dei tavoli";
    }

    if(!pub->aggiungiTavolo(2)){
        cerr << "Errore durante l'aggiunta dei tavoli";
    }

    if(!pub->aggiungiTavolo(5)){
        cerr << "Errore durante l'aggiunta dei tavoli";
    }

    if(!pub->aggiungiTavolo(2)){
        cerr << "Errore durante l'aggiunta dei tavoli";
    }
```

Registrazione del Signal Handler e Configurazione della Socket Una volta registrato il **Signal Handler** si arriva alla parte fondamentale del codice : `La Creazione e Configurazione della Socket`.

```
// Registrazione del signal handler
    signal(SIGINT, signalHandler);
    serverSocketPtr = &serverSocket; // passo l'indirizzo di memoria della
        variabile serverSocket

    // Creazione della socket server
```

```
    if (!serverSocket.create()) {
        cerr << "Errore nella creazione del socket!" << endl;
        exit(1);
    }

    // Costruisce la socket con la porta 25564
    if (!serverSocket.bind(25564)) {
        cerr << "Errore nell'associazione del socket alla porta!" << endl;
        exit(1);
    }

    // Imposta la socket in ascolto
    if (!serverSocket.listen()) {
        cerr << "Errore nel mettere il socket in ascolto!" << endl;
        exit(1);
    }

    cout << "Il pub e' aperto in attesa di clienti" << endl; // Stampa di
       avviso
```

Di conseguenza il pub si mette di ascolto ed attende nuove connessioni in entrata.

Gestione delle Connessioni Il ciclo `while (true)` è utilizzato per gestire continuamente le richieste di connessione da parte dei client, in questo caso rappresentati dai camerieri che si connettono al server del Pub.

```
  while (true) { // ciclo per accettare piu richieste client

        Socket clientSocket; // socket per acquisire le informazioni del
           client (cameriere)

        if (!serverSocket.accept(clientSocket)) { // accetta la
           connessione del client
            cerr << "Errore nell'accettare la connessione!" << endl;
            continue; // continua con il ciclo
```

```
8         }
9
10        clientSocketPtr = &clientSocket;
11
12        cout << "Il cameriere sta servendo un Cliente" << endl; // stampa
              un messaggio di successo della connessione con il client
13
14        // Crea un processo figlio per gestire la connessione client
15        pid_t pid = fork();
16
17        if(pid < 0) { // Errore creazione processo figlio
18            cerr << "Errore nella fork";
19            continue; // continua con il ciclo
20        }
```

Il ciclo permette al server di rimanere in esecuzione per un tempo indefinito, in attesa di connessioni da parte dei client. Ogni volta che un client tenta di connettersi, viene creata una nuova `Socket` chiamata `clientSocket` per gestire la comunicazione con quel client. La funzione `accept()` è utilizzata per accettare una nuova connessione in ingresso. Per ogni connessione client accettata, viene creato un nuovo processo figlio utilizzando la funzione `fork()`. Il processo figlio è responsabile della gestione della comunicazione con il client. Se la `fork()` fallisce, viene stampato un messaggio di errore e il server continua con il ciclo, pronto ad accettare altre connessioni.

Processo Figlio Una volta che un processo figlio è stato creato, viene utilizzato per gestire la comunicazione con il client che si è appena connesso al server.

```
1 if (pid == 0) { // Processo figlio
2
3             string message; // memorizza i messaggi che arrivano dal
                  client
4             int ntavolo; // memorizza il numero di tavolo
5
```

```
6             if (clientSocket.receive(message) <= 0) { // se il messaggio
                  non viene ricevuto correttamente
7                 exit(1); // In caso di errore termino il processo figlio
8             }
9
10            //verifica se ci sono posti disponibili per far accomodare il
                  cliente
11            if(message.compare("Ci sono posti liberi?") == 0){
12                if(pub->postiDisponibili() > 0){ // verifica disponibilita
                      posti
13                    if (!clientSocket.send("Si")) { // Invia si al
                          cameriere se ci sono posti
14                        exit(1); // Termina il processo figlio
15                    }
16
17                    message.clear();
18
19                    if (clientSocket.receive(message) <= 0) { // riceve
                          dal cameriere la scelta del tavolo
20                        exit(1); // Termina il processo figlio
21                    }
22
23                    // Se il cliente decide di accomodarsi ad un nuovo
                          tavolo, si controlla se ci sono tavoli vuoti
                          disponibili
24                    if (message.compare("nuovo") == 0) {
25                        ntavolo = pub->tavoloVuoto(); // memorizza il
                              numero di tavolo vuoto
26                        if (ntavolo > 0) { // Verifica se ci sono tavoli
                              vuoti disponibili
27                            if (pub->aggiungiCliente(ntavolo)) { //
                                  Aggiunge il cliente al nuovo tavolo
28                                cout << "Cliente aggiunto al tavolo: " <<
                                      ntavolo << endl;
29                                clientSocket.send(to_string(ntavolo)); //
                                      Invia il numero di tavolo al cameriere
```

```
30                          } else {
31                              cerr << "Errore: non e' stato possibile
                                    aggiungere il cliente al tavolo: " <<
                                    ntavolo << endl;
32                              exit(1); // Termina il processo figlio
33                          }
34                      } else {
35                          clientSocket.send("Non ci sono tavoli vuoti");
                                // Avvisa che non ci sono tavoli vuoti
                                disponibili
36                          exit(1); // Termina il processo figlio
37                      }
38                  }
39
40                  // Se il cliente decide di accomodarsi in un tavolo
                        gia' occupato ma con posti disponibili
41                  else{
42                      try {
43                          ntavolo = stoi(message); // conversione della
                                stinga per recuperare il numero di tavolo
                                richiesto
44                          if(pub->aggiungiCliente(ntavolo)){ // Aggiunge
                                il cliente al tavolo se ci sono posti
45                              cout << "Cliente aggiunto al tavolo: " <<
                                    ntavolo << endl;
46                              clientSocket.send(message); // invia un
                                    messaggio di avviso al cameriere di
                                    posti disponibili
47                          }
48                          else{ //Se non ci sono posti
49                              clientSocket.send("Tavolo inesistente o
                                    posti non disponibili"); // avvisa il
                                    camerire
50                          }
51                      // Eccezioni in caso di errore della conversione
                            della stringa in intero
```

```
52                          } catch (const invalid_argument& e) {
53                              clientSocket.send("Messaggio non valido: non e
                                    ' un numero");
54                              exit(1); // Termina il processo figlio
55                          }
56                      }
57
58                      message.clear();
59
60                      if (clientSocket.receive(message) <= 0) { // Ricevo l'
                            ordine dal cameriere
61                          exit(1); // Termina il processo figlio
62                      }
63
64                      termine_pub(&clientSocket, &message, ntavolo, pub); //
                            Se il cliente o il cameriere si disconnettono
                            senza preavviso
65
66                      if(message.substr(0, 14).compare("Prepara ordine") ==
                            0){ // Se il cameriere ha consegnato l'ordine
67                          pub->preparaOrdine(stoi(message.substr(15, 16)));
                                // Prepara l'ordine per il tavolo indicato dal
                                cameriere
68                          // Il pub avvisa il camerire che pu  servire l'
                                ordine
69                          clientSocket.send("Ordine pronto al tavono n: " +
                                message.substr(15, 16));
70                      }
71
72                      message.clear();
73                  }
74                  // Avvisa il cameriere se nel Pub non ci sono posti
75                  else{
76                      if (!clientSocket.send("No")) {
77                          exit(1); // Errore nell'invio
78                      }
```

```
79                  }
80              }
81
82              message.clear();
83
84              // Il Cameriere avvisa che il cliente ha lasciato il locale
85              clientSocket.receive(message);
86
87              cout << "Cameriere: " << message << endl; // Stampa il
                    messaggio del cameriere
88
89
90              if(message.substr(0,29).compare("Cliente ha liberato il tavolo
                    ") == 0){ // Se il cliente se ne e' andato
91                  pub->liberaPosto(stoi(message.substr(33))); // libera il
                        posto al numero di tavolo
92                  cout << "Si e' liberato il posto al tavolo n: "  <<
                        message.substr(33) << endl; // stampa un avviso
93              }
94              serverSocket.close(); // chiude la socket
95              exit(0); // Termina il processo figlio
96          }
97      }
98
99      // Dettach della memoria condivisa
100     shmdt(pub);
101     shmctl(shmid, IPC_RMID, NULL);
102
103     return 0;
104 }
```

Quindi, tramite l'utilizzo dei metodi precedentemente creati Il Pub è in grado di :

- **Ricevere Messaggi:** Il processo figlio attende messaggi dal client e li gestisce in base al contenuto. Se la ricezione fallisce, il processo figlio termina.

- **Gestire i Posti Disponibili:** Se il messaggio ricevuto è una richiesta di disponibilità di posti (`"Ci sono posti liberi?"`), il processo figlio verifica la disponibilità e risponde al client di conseguenza. Se ci sono posti disponibili, il cliente può scegliere un tavolo nuovo o uno già occupato.

- **Gestire l'Ordine:** Una volta che il cliente è seduto, il processo figlio attende l'ordine dal cameriere e lo inoltra al Pub per la preparazione.

- **Liberare i Posti:** Se il cliente lascia il tavolo, il processo figlio aggiorna lo stato del Pub liberando il posto occupato.

Il processo figlio termina la sua esecuzione chiudendo la socket e rilasciando le risorse quando non è più necessario mantenere la comunicazione con il client. Questo approccio garantisce che ogni connessione client venga gestita in maniera isolata, evitando interferenze tra le richieste. Inoltre, nel codice sono previsti una serie di controlli e comportamenti specifici, tra cui:

- **Errori di Ricezione:** In caso di problemi nel ricevere i messaggi dal client, il processo figlio termina per prevenire stati inconsistenze.

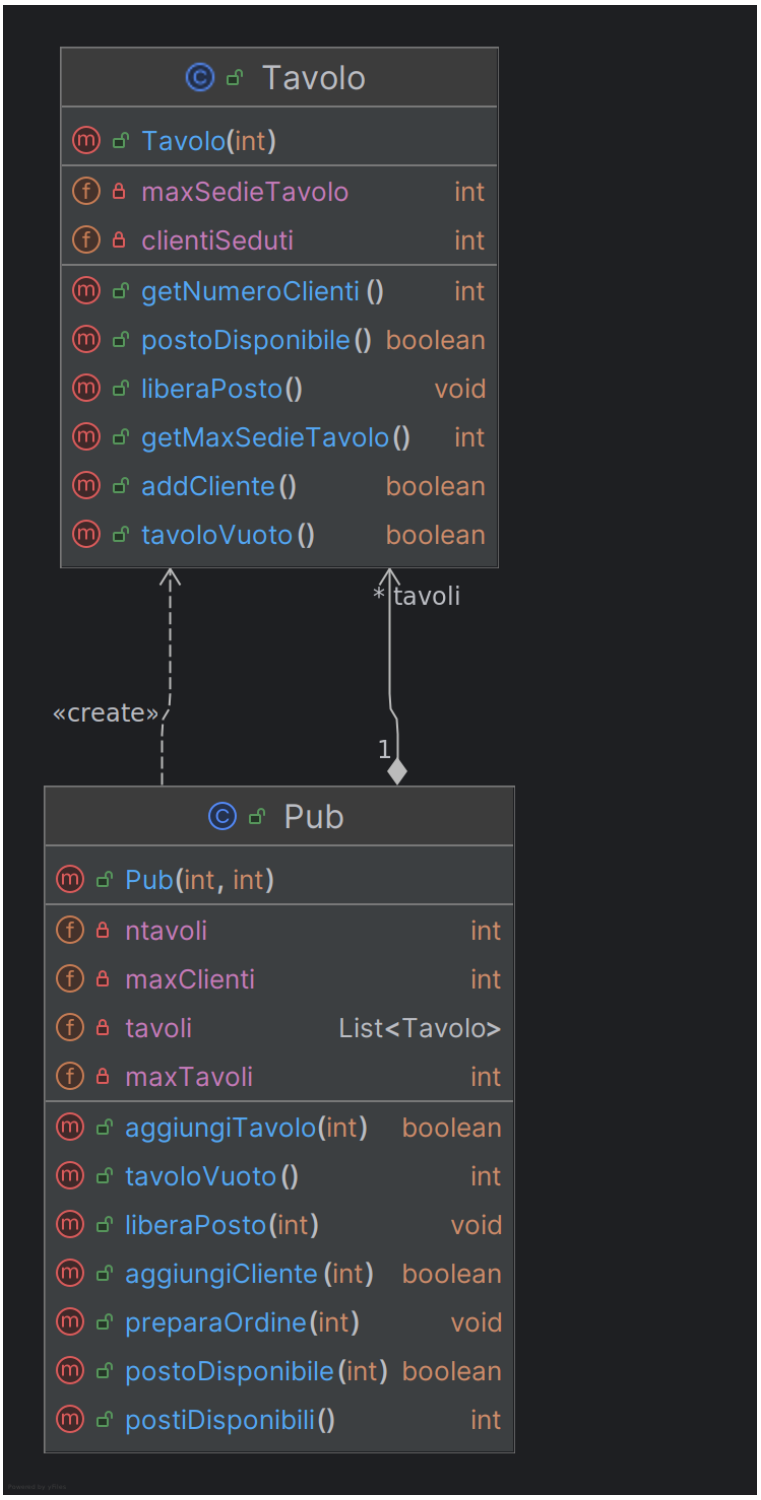
- **Conversione di Stringhe:** La conversione della stringa che rappresenta il numero di tavolo in un intero è gestita con un `try-catch` per evitare crash in caso di input non validi.

- **Gestione degli Ordini:** Se viene ricevuto un messaggio di ordine, il processo si assicura che l'ordine venga preparato correttamente e notifica il cameriere.

3.2.4 Schema Server

Il **server** realizzato rappresenta un sistema robusto per la gestione delle connessioni client in un contesto di simulazione di un Pub. Le classi **Tavolo** e **Pub** formano la struttura fondamentale per la gestione dei tavoli e dei posti a sedere, consentendo un'allocazione dinamica e ottimale delle risorse. La classe Tavolo gestisce i posti disponibili e occupati per ogni tavolo, mentre la classe Pub coordina l'intero locale, gestendo le richieste di nuovi clienti e assicurando che ogni ordine venga processato correttamente.

Il **main** funge da punto centrale per la gestione delle connessioni, utilizzando socket per interagire con i camerieri che rappresentano i client. Attraverso un ciclo di ascolto continuo, il server accetta nuove connessioni e crea processi figli per gestirle in modo isolato. Questo approccio garantisce che ogni cliente venga servito correttamente senza interferenze tra le diverse richieste. La gestione dei segnali assicura che, in caso di interruzioni, le risorse vengano rilasciate in modo ordinato, evitando perdite di dati o inconsistenze.


Tavolo
Tavolo(int)
maxSedieTavolo int
clientiSeduti int
getNumeroClienti () int
postoDisponibile () boolean
liberaPosto() void
getMaxSedieTavolo() int
addCliente() boolean
tavoloVuoto () boolean
*
tavoli
«create»
1
Pub
Pub(int, int)
ntavoli int
maxClienti int
tavoli List<Tavolo>
maxTavoli int
aggiungiTavolo(int) boolean
tavoloVuoto() int
liberaPosto(int) void
aggiungiCliente (int) boolean
preparaOrdine(int) void
postoDisponibile(int) boolean
postiDisponibili() int

3.3 Cameriere - main.cpp

Il codice del cameriere rappresenta una parte essenziale della comunicazione all'interno del sistema di simulazione client-server del Pub. Il cameriere agisce come un **intermediario** tra i clienti e il Pub, gestendo la comunicazione `bidirezionale` e assicurando che le richieste dei clienti siano soddisfatte correttamente. Il cameriere si occupa di *accogliere i clienti*, *verificare la disponibilità dei tavoli* presso il Pub, *inviare i menù e trasmettere gli ordini*. Inoltre, il cameriere gestisce anche le situazioni di emergenza come disconnessioni improvvise, grazie a meccanismi di gestione dei segnali.

3.3.1 Librerie e Socket Globali

Le librerie di questo file sono le stesse già analizzate nei file precedeti cosi come l'utilizzo dei puntatori per gestire il signal handling.

```
// Socket globale per il signal handling
Socket* clientSocketPtr = nullptr;
Socket* serverSocketPtr = nullptr;
```

3.3.2 Gestione delle Interruzioni e Funzioni Ausiliarie del Cameriere

```
// Signal handler per l'interruzione del programma
void signalHandler(int);

// Funzione per inviare il menu al cameriere
void send_menu(Socket*);

// Funzione del cameriere in caso di SIGNIT
void termine_cameriere(Socket *, Socket *, string *);
```

Descrizione delle Funzioni:

- `signalHandler(int signum)`: Questa funzione viene invocata quando il programma riceve un segnale di interruzione, come `SIGINT`. La funzione si occupa di chiudere in modo sicuro le socket aperte prima di terminare il programma.
  - Se `clientSocketPtr` non è nullo, invia un messaggio al Pub per notificare la terminazione del cameriere e chiude la socket del client.
  - Se `serverSocketPtr` non è nullo, chiude la socket del server.
  - Infine, il programma termina invocando la funzione `exit()` con il segnale ricevuto come codice di uscita.

```
void signalHandler(int signum) {
    //chiusura della socket client
    if (clientSocketPtr != nullptr) {
        clientSocketPtr->send("termine_cameriere"); // invia l'avviso
            al pub
        clientSocketPtr->close(); // chiude la socket
        std::cout << "Client socket closed successfully." << std::endl
            ; // stampa di avviso
    }
    //chiusura della socket server
    if (serverSocketPtr != nullptr) {
        serverSocketPtr->close(); // chiude la socket
        std::cout << "Server socket closed successfully." << std::endl
            ; // stampa di avviso
    }
    exit(signum); //uscita dal programma
}
```

- `send_menu(Socket* client)`: Questa funzione invia il menu del Pub al cliente connesso. Il menu è rappresentato come una stringa multilinea, contenente una lista di opzioni "originali". Se l'invio del messaggio fallisce, la funzione chiude il processo figlio con un errore.

```
void send_menu(Socket* client) {
    // menu del Pub
    string message = R"(Menu Pub
                        1) Fourier small menu
                        2) ADSL large menu
                        3) Fibra medium menu
                        4) ALOHA single hamburger
                        5) Ethernet vegan menu small
                        6) Berkeley Socket menu
                        7) UDP BBQ menu
                        8) DNS Nuggets x6
                        9) DNS Nuggets x12
                        10) LAN Wrap menu
                        11) MultiplexingCola
                        12) Gran P2P Bacon menu
                        13) ISO Water
                        14) RFID Fanta
                        )";

    if (!client->send(message)) { // invia il menu al cliente
        cerr << "Errore nell'invio del menu al cliente" << endl;
        exit(1); // Termina il processo figlio
    }
}
```

- `termine_cameriere(Socket* remoteSocket, Socket* client, string* message)`: Questa funzione gestisce la terminazione del cameriere quando riceve un segnale `SIGINT`. Verifica se il client o il Pub si sono disconnessi improvvisamente e chiude la rispettiva connessione in modo sicuro, inviando i messaggi di terminazione necessari e terminando il processo.

```
void termine_cameriere(Socket *remoteSocket, Socket *client, string *
    message) {

```

```
3       // Se il cliente termina improvvisamente di funzionare
4       if (message->compare("termine_cliente") == 0) {
5           cout << "Un cliente si e' disconnesso improvvisamente!" <<
                endl; // Stampa di errore
6           remoteSocket->send(*message); // invia l'avviso al Pub
7           client->close(); // chiude la socket
8           exit(0); // Termina il programma
9       } else if (message->compare("termine_pub") == 0) {
10          cout << "Il pub si e' disconnesso improvvisamente! Si prega di
                riavviarlo." << endl; // Stampa di errore
11          client->send(*message); // invia l'avviso al cliente
12          remoteSocket->close(); // chiude la socket
13          exit(0);
14      }
15  }
```

3.3.3 Funzionalità del Cameriere

La funzione `main()` rappresenta il nucleo dell'applicazione del cameriere. In questa parte del codice, vengono gestite l'inizializzazione e la configurazione della socket server, permettendo al cameriere di ascoltare le richieste dei clienti e di comunicare con il Pub. Le operazioni di `creazione`, `bind` e `listen` della socket sono già state analizzate in dettaglio, quindi non è necessario ripeterne la spiegazione qui.

In questa sezione, ci concentreremo esclusivamente sulle parti più rilevanti del codice del cameriere, evitando di dilungarci inutilmente. Il codice completo, con commenti dettagliati, è comunque disponibile su **GitHub**.

- **Accoglienza del Cliente:**
  - Quando un cliente si connette, il cameriere verifica se ci sono posti liberi chiedendolo al Pub.
  - Se ci sono posti disponibili, il cameriere chiede al cliente se desidera accomodarsi a un nuovo tavolo o a uno già esistente.

- **Scelta del Tavolo:**
  - **Nuovo Tavolo:** Se il cliente sceglie un nuovo tavolo, il cameriere chiede al Pub quale sia il primo tavolo vuoto disponibile.
  - **Tavolo Esistente:** Se il cliente desidera sedersi a un tavolo già esistente, deve fornire il numero del tavolo. Il cameriere invia questa informazione al Pub, che verifica la disponibilità del tavolo.
- **Accomodamento del Cliente:**
  - Dopo aver ricevuto una risposta dal Pub, il cameriere fa accomodare il cliente al tavolo specificato.
- **Richiesta del Menu:**
  - Il cliente può richiedere il menu, che viene fornito dal cameriere tramite una funzione apposita già descritta.
- **Presa dell'Ordinazione:**
  - Il cameriere raccoglie l'ordinazione del cliente e la invia al Pub per la preparazione.
- **Consegna dell'Ordine:**
  - Una volta che il Pub ha terminato la preparazione, comunica al cameriere che l'ordine è pronto per la consegna.
  - Il cameriere consegna quindi l'ordine al tavolo e al cliente specificato.
- **Notifica di Uscita del Cliente:**
  - Quando il cliente lascia il tavolo e abbandona il Pub, il cameriere notifica il Pub, che aggiorna il numero di clienti seduti.

Tutte queste operazioni vengono gestite attraverso una serie di scambi di messaggi tra il cliente, il cameriere e il Pub. È importante notare che il Pub non comunica mai direttamente con il cliente finale; tutta la comunicazione avviene tramite il cameriere. Il

flusso di lavoro è gestito tramite una serie di controlli condizionali `if-else` e controlli delle eccezioni per assicurare che ogni operazione avvenga correttamente e in modo sicuro.

3.4 Cliente - main.cpp

Il codice del client rappresenta la "visione opposta" rispetto a quella del cameriere, poiché è il client stesso a inviare le richieste al cameriere. Per questo motivo, risulta ridondante ripetere le operazioni già descritte in dettaglio per il cameriere.

Una differenza significativa da notare riguarda l'introduzione della funzione `numero_valido()`, che consente di verificare se il numero della portata inserito dal cliente è valido, ovvero compreso nel range del menu. Questo controllo viene effettuato direttamente nel codice del client e gestito all'interno della funzione `main()`, come segue:

```
1 if (!numero_valido(stoi(scelta), 1, 14)) {
2 cout << "Inserisci un numero compreso tra 1 e 14." << endl;
3 continue; // Torna all'inizio della richiesta della portata
4 }
```

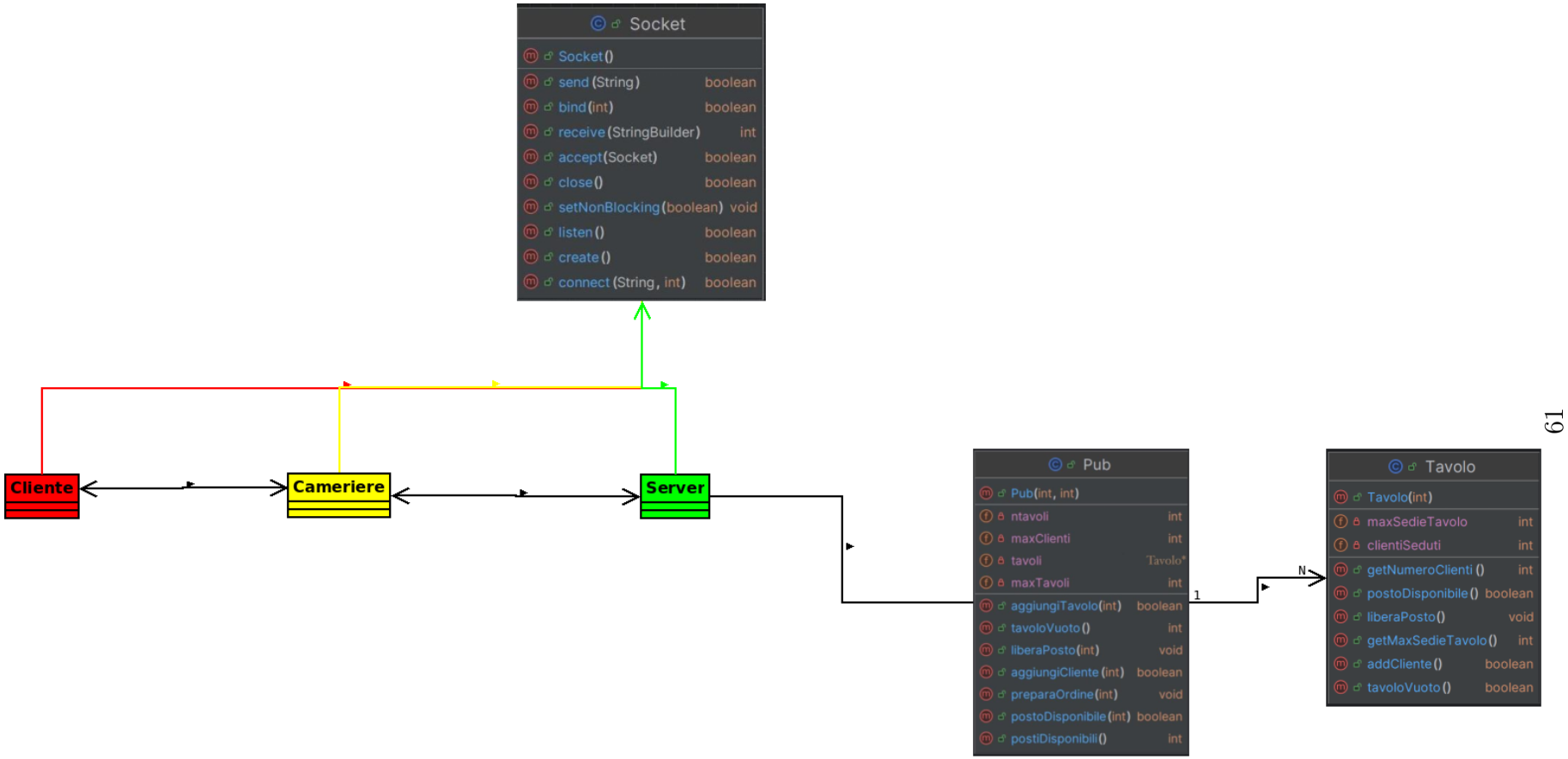
In questo modo, si evita di interrompere la ricezione dell'ordine, permettendo al cliente di inserire un numero valido senza bloccare il flusso dell'applicazione.

4 Diagrammi

In questa sezione vengono presentati i *diagrammi* principali utilizzati per modellare e comprendere l'architettura del progetto. I diagrammi forniscono una rappresentazione visiva delle strutture e delle interazioni tra i componenti chiave del sistema, facilitando la comprensione delle relazioni e delle responsabilità all'interno del codice.

4.1 Diagramma delle Classi

Il diagramma delle classi offre una panoramica delle classi principali utilizzate nel progetto, evidenziando i loro attributi, metodi e le relazioni tra di esse. In particolare, il diagramma illustra la struttura delle classi `Socket`, `Pub` e `Tavolo`, mentre per quanto riguarda Il cameriere ed il cliente, non essendo delle classi, ci concediamo una *"licenza poetica"* per poterli rappresentare sotto forma di classi.

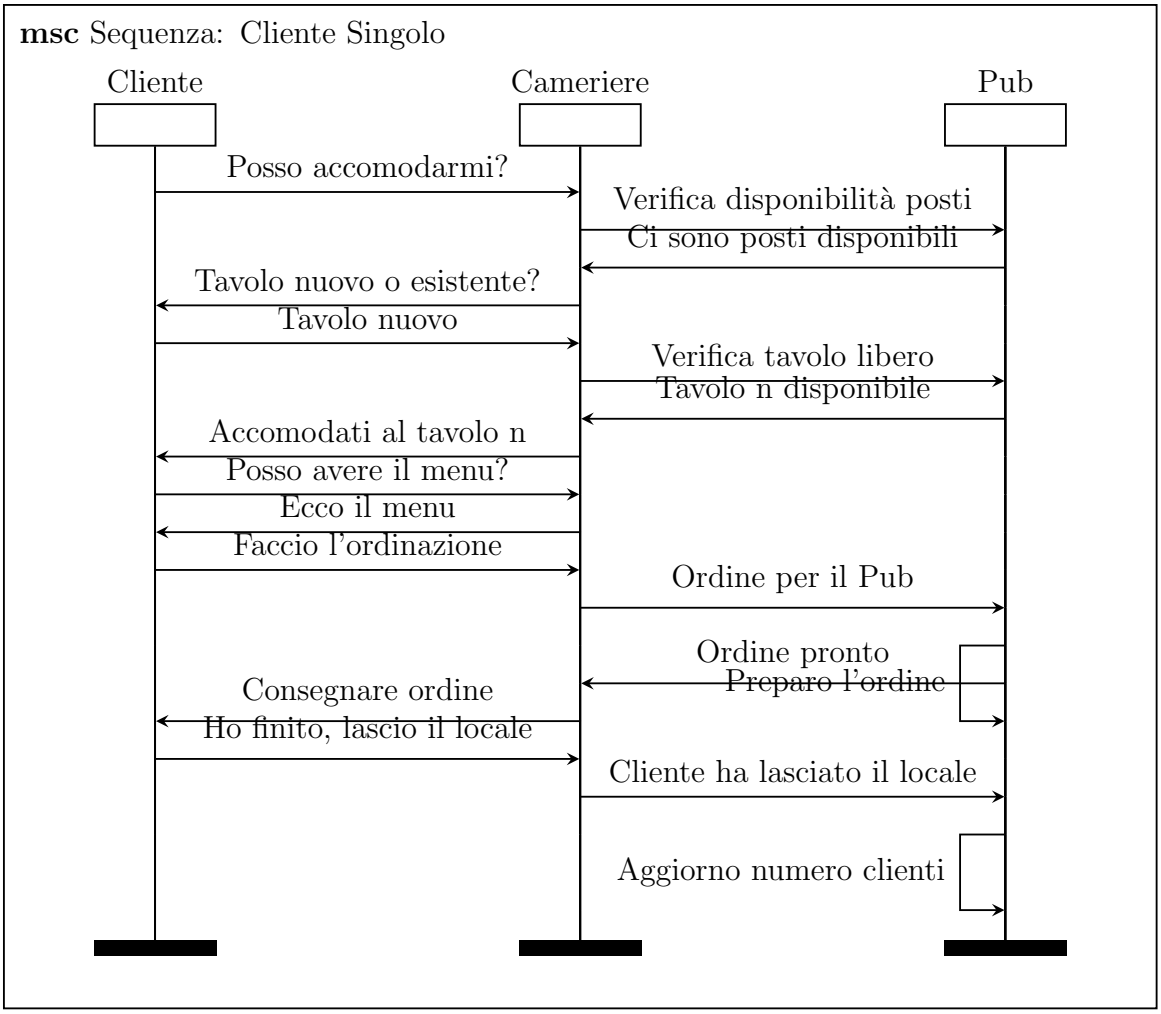

Socket
Socket()
send(String) boolean
bind(int) boolean
receive(StringBuilder) int
accept(Socket) boolean
close() boolean
setNonBlocking(boolean) void
listen() boolean
create() boolean
connect(String, int) boolean
Cliente
Cameriere
Server
Pub
Pub(int, int)
ntavoli int
maxClienti int
tavoli Tavolo*
maxTavoli int
aggiungiTavolo(int) boolean
tavoloVuoto() int
liberaPosto(int) void
aggiungiCliente(int) boolean
preparaOrdine(int) void
postoDisponibile(int) boolean
postiDisponibili() int
1
N
Tavolo
Tavolo(int)
maxSedieTavolo int
clientiSeduti int
getNumeroClienti() int
postoDisponibile() boolean
liberaPosto() void
getMaxSedieTavolo() int
addCliente() boolean
tavoloVuoto() boolean

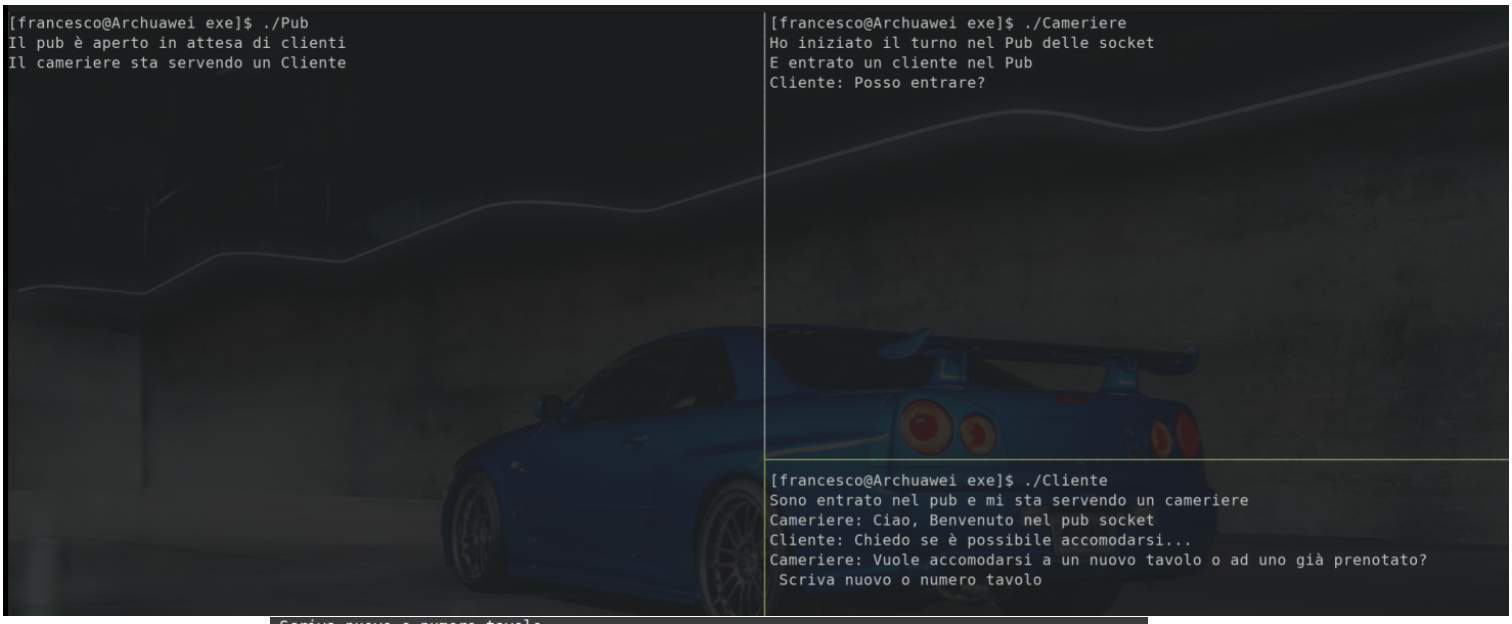
4.2 Diagramma delle Sequenze

Il diagramma di sequenza mostra il flusso di messaggi e le interazioni tra il client, il cameriere e il pub durante le operazioni principali. Questo diagramma aiuta a visualizzare come le richieste e le risposte vengono scambiate tra le diverse entità del sistema.

Per esssere quanto più completi possibili andremo ad analizzare diversi casi che si possono verificare, con tanto di `Esecuzione` e, `Diagramma delle sequenze`.

4.2.1 Caso 1






```
 Scriva nuovo o numero tavolo
nuovo
Cameriere: Accomodati al tavolo numero: 1
Cliente: Richiedo il menu al cameriere
Cameriere: Menu Pub
                        1) Fourier small menu
                        2) ADSL large menu
                        3) Fibra medium menu
                        4) ALOHA single hamburger
                        5) Ethernet vegan menu small
                        6) Berkeley Socket menu
                        7) UDP BBQ menu
                        8) DNS Nuggets x6
                        9) DNS Nuggets x12
                        10) LAN Wrap menu
                        11) MultiplexingCola
                        12) Gran P2P Bacon menu
                        13) ISO Water
                        14) RFID Fanta

Inserisci il numero della portata desiderata (da 1 a 14) e conferma con INVIO.
 Per terminare l'ordine, digita 'exit'.
Portata: 2
Portata: 3
Portata: 14
Portata: exit
Hai consegnato il tuo ordine al cameriere, ora verrà consegnato al pub per la preparazione.
Cameriere: Ordine pronto e consegnato
```

```
Cliente: Posso entrare?
Porto il menu al tavolo 1
ho acquisito l'ordine e lo trasmetto al pub
Pub: Ordine pronto al tavono n: 1
itiro il menu e lo consegno al tavolo
Cliente: Grazie mille e arrivederci
Il cliente ha abbandonato il pub
```

```
Sto preparando l'ordine del tavolo 1
Ordine pronto per essere consegnato al tavolo 1 dal cameriere
Cameriere: Cliente ha liberato il tavolo n: 1
Si è liberato il posto al tavolo n: 1
```

Il diagramma fornito potrebbe variare se invece di `nuovo` il cliente inserisca `un numero di tavolo specifico`:

```
 Scriva nuovo o numero tavolo
2
Cameriere: Accomodati al tavolo numero: 2
Cliente: Richiedo il menu al cameriere
Cameriere: Menu Pub
                        1) Fourier small menu
                        2) ADSL large menu
                        3) Fibra medium menu
                        4) ALOHA single hamburger
                        5) Ethernet vegan menu small
                        6) Berkeley Socket menu
                        7) UDP BBQ menu
                        8) DNS Nuggets x6
                        9) DNS Nuggets x12
                        10) LAN Wrap menu
                        11) MultiplexingCola
                        12) Gran P2P Bacon menu
                        13) ISO Water
                        14) RFID Fanta
```

Il che cambia poco, poichè il resto dell'esecuzione è identico.

Esempio di alcune situazioni limite:
Il Client si disconnette improvvisamente mentre è ancora connesso al Cameriere.
Il Client richiede un tavolo inesistente, generando un errore che porta alla disconnessione del client.
Il Client tenta di ordinare una portata non valida (ad esempio un numero al di fuori del range 1-14).
Il Client sbaglia la sintassi di una scelta (errore nella scrittura di *nuovo*.

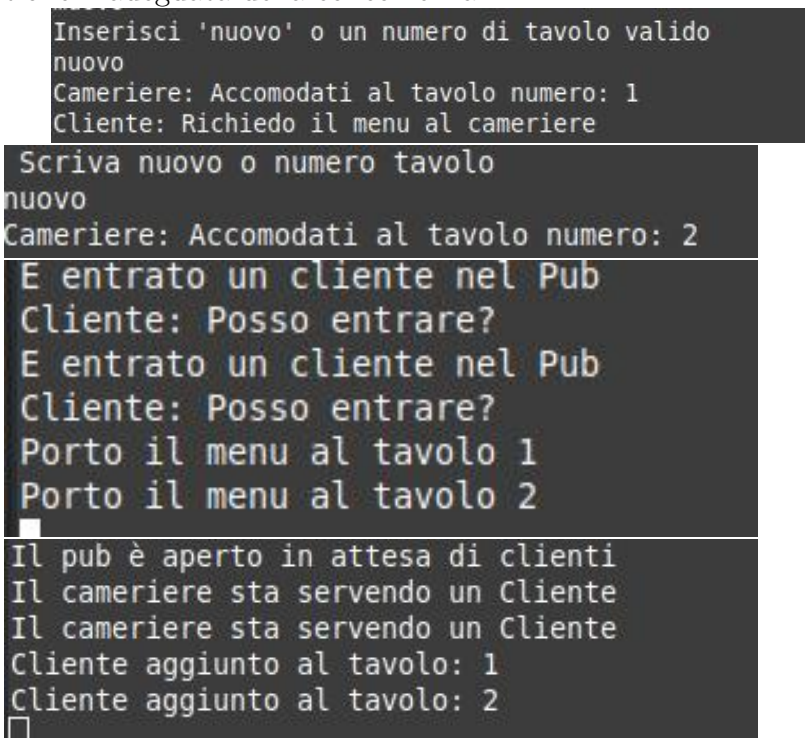
```
 Scriva nuovo o numero tavolo
nuevo
Inserisci 'nuovo' o un numero di tavolo valido
```

4.2.2 Caso 2

Un aspetto cruciale della nostra architettura è la capacità di gestire la concorrenza, ossia di funzionare correttamente con più client collegati contemporaneamente senza compromettere l'integrità del sistema o la qualità del servizio offerto. Questo è particolarmente importante in scenari reali dove è comune che più utenti interagiscano con il sistema nello stesso momento.

Quando due o più client si collegano contemporaneamente, la nostra architettura è progettata per gestire le richieste in modo efficiente e isolato, garantendo che le operazioni di un client non interferiscano con quelle degli altri. Un esempio significativo di questa capacità si

verifica quando più client richiedono simultaneamente di essere assegnati a un nuovo tavolo. Il nostro server è in grado di processare queste richieste in parallelo, assegnando correttamente i tavoli disponibili senza sovrapposizioni o blocchi. Questo evita situazioni in cui due client potrebbero essere assegnati allo stesso tavolo o in cui uno dei client rimanga bloccato a causa di una gestione inadeguata della concorrenza.



Definizioni

- **TCP/IP** è un insieme di protocolli di comunicazione utilizzati per interconnettere dispositivi in rete. Garantisce un trasporto affidabile dei dati e che siano consegnati nell'ordine giusto. Questo è fondamentale per molte applicazioni, specialmente quando è cruciale che i dati non vengano persi o danneggiati durante il transito.

- **Le socket** sono interfacce di programmazione che permettono la comunicazione tra due nodi in una rete, utilizzando protocolli come TCP/IP. Sono usate per creare connessioni bidirezionali tra client e server.

- Un **"socket in modalità non bloccante"** è un socket che, quando esegue operazioni di I/O (come inviare o ricevere dati), non costringe il programma a fermarsi e aspettare finché l'operazione non è completata. In altre parole, il programma può continuare a eseguire altre operazioni anche se l'operazione di I/O non è ancora terminata.

- Un **file descriptor** è un identificatore univoco che il sistema operativo assegna a ogni file o risorsa aperta, come un file, un socket o una pipe, per gestire le operazioni di input/output. In pratica, è un numero intero che il sistema utilizza per tenere traccia di queste risorse durante l'esecuzione di un programma.

- Un **indirizzo IP** (Internet Protocol address) è un identificatore univoco assegnato a ciascun dispositivo connesso a una rete che utilizza il protocollo IP. Esso consente di localizzare e identificare il dispositivo all'interno della rete. Gli indirizzi IP possono essere sia IPv4 (es. 192.168.0.1) che IPv6 (es. 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

- Una **porta** del socket è un numero che identifica un canale di comunicazione specifico su un dispositivo di rete. Le porte sono utilizzate insieme all'indirizzo IP per indirizzare correttamente i dati all'interno di una rete. Ogni porta è associata a un servizio o applicazione specifica e può variare da 0 a 65535.

- La funzione **fork()** è una chiamata di sistema utilizzata per creare un nuovo processo figlio che è una copia del processo padre. Nel contesto di un server, questa tecnica è spesso utilizzata per gestire connessioni multiple in modo simultaneo.

5 Manuale Utente per Compilazione ed Esecuzione del Progetto

Per iniziare a lavorare con il progetto, segui i passaggi descritti di seguito:

1. Accedi al repository GitHub del progetto tramite il seguente link: **GitHub Repository**.
2. Clicca sul pulsante `Code` e seleziona `Download ZIP`.
3. Dopo aver scaricato il file ZIP, decomprimilo in una directory a tua scelta.
4. Apri un terminale e spostati nella directory principale del progetto usando il comando `cd`.
5. Successivamente, accedi alle singole cartelle del progetto per la compilazione ed esecuzione, utilizzando i seguenti comandi:

   - **Per il Pub:**

     ```
     g++ main.cpp -o Pub
     ./Pub
     ```

   - **Per il Cameriere:**

     ```
     g++ main.cpp -o Cameriere
     ./Cameriere
     ```

   - **Per il Cliente:**

     ```
     g++ main.cpp -o Cliente
     ./Cliente
     ```

     Nota Bene: Per testare la concorrenza dell'architettura, si consiglia di eseguire il comando relativo al cliente su più terminali contemporaneamente.

6. Nel repository GitHub sono già presenti file eseguibili precompilati, che non richiedono ulteriore compilazione. Questi possono essere eseguiti direttamente tramite il seguente comando:

```
cd exe
./Pub
./Cameriere
./Cliente
```

Seguendo questi passaggi, sarai in grado di compilare ed eseguire correttamente i componenti del progetto. La procedura di compilazione ed esecuzione è disponibile nel file `README` disponibile su GITHUB.