UNIVERSITÀ
DEGLI STUDI
DI MESSINA

Scienze informatiche

Laboratorio di reti e sistemi distribuiti

Relazione Progetto LabReSiD23

Antonio Edoardo Ciliberto

Emanuele Russo

Professore: Roberto Marino

Indice

1. Stato dell'arte

1.1 Introduzione al settore della gestione dei ricambi

La gestione dei ricambi è cruciale per diverse industrie, inclusi settori come l'automotive, l'industria manifatturiera, e il retail. Le soluzioni di gestione dei ricambi consentono di tenere traccia degli inventari, delle vendite e delle operazioni di manutenzione, garantendo che i componenti necessari siano sempre disponibili quando richiesti.

1.2 Soluzioni tradizionali

In passato, la gestione dei ricambi era spesso effettuata manualmente o utilizzando software desktop specializzati. Questi sistemi richiedevano l'installazione su ogni macchina client e non sempre permettevano un accesso centralizzato o aggiornamenti in tempo reale dei dati.

1.3 Sistemi moderni di gestione dei ricambi

Negli ultimi anni, sono emersi sistemi più avanzati che sfruttano le tecnologie web e le applicazioni client-server per migliorare l'efficienza e l'accessibilità della gestione dei ricambi:

- **Applicazioni Web**: Soluzioni basate su web permettono l'accesso da qualsiasi dispositivo connesso a Internet, migliorando la flessibilità e la scalabilità dell'applicazione.

- **Tecnologie Cloud**: L'utilizzo di piattaforme cloud permette di archiviare e gestire grandi quantità di dati in modo sicuro e affidabile, con accesso 24/7 da qualsiasi posizione.

- **Sistemi Client-Server**: L'architettura client-server consente una separazione chiara tra la logica di business (server) e l'interfaccia utente (client), facilitando la gestione e la manutenzione dell'applicazione.

1.4 Strumenti e tecnologie comuni

Nel campo della gestione dei ricambi, sono utilizzate diverse tecnologie per implementare sistemi robusti e efficienti:

- **Database Relazionali**: Come MySQL o PostgreSQL, per la gestione strutturata dei dati dei ricambi.
- **Linguaggi di Programmazione**: Come C, Python, Java, etc., utilizzati per lo sviluppo dell'applicazione client e server.
- **Tecnologie di Comunicazione**: Come HTTP/HTTPS per le comunicazioni web, e TCP/IP per la comunicazione tra client e server tramite socket.
- **Framework e Librerie**: Come Flask, Spring, Django, etc., che facilitano lo sviluppo di applicazioni web e la gestione delle interfacce utente.

1.5 Tendenze e sviluppi futuri

Il settore della gestione dei ricambi è in continua evoluzione. Le tendenze future includono:

- **Integrazione di Intelligenza Artificiale**: Per ottimizzare le previsioni di domanda, la gestione degli inventari e la manutenzione predittiva.
- **Automatizzazione dei Processi**: Utilizzo di robotica e automazione per la gestione fisica degli inventari e la spedizione.
- **Blockchain**: Potenziale utilizzo per la tracciabilità e l'autenticità dei ricambi, migliorando la sicurezza e la gestione della catena di approvvigionamento.

2. Descrizione problema

Realizzare in C un sistema multithreading client/server per la gestione da remoto di un magazzino ricambi secondo il paradigma CRUD usando SQLite.

3. Implementazione

3.1 Implementazione client

3.1.1 Architettura dell'applicazione

il lato client dell'applicazione è stato progettato per gestire interazioni utente e comunicazioni con il server mediante l'uso di socket TCP/IP. L'architettura è basata su un modello di menu interattivo che permette agli utenti di selezionare varie operazioni come aggiunta, visualizzazione, aggiornamento, eliminazione e acquisto di ricambi.

3.1.2 Linguaggi e tecnologie utilizzate

Il client è stato sviluppato in linguaggio C utilizzando le seguenti librerie standard:

- <**stdio.h**> per le funzioni di input/output standard.
- <**stdlib.h**> per la gestione della memoria e delle allocazioni dinamiche.
- <**string.h**> per le operazioni su stringhe come la manipolazione e la comparazione.
- <**unistd.h**> per la gestione del terminale e delle funzioni di sistema.
- <**termios.h**> per la gestione dell'input da tastiera oscurando la digitazione della password
- <**arpa/inet.h**> per la gestione degli indirizzi IP e delle conversioni.

3.1.3 Connessione al server

Il client crea un socket TCP/IP tramite la funzione '**socket()**' per stabilire una connessione con il server. Utilizza '**inet_pton()**' per convertire l'indirizzo IP "**127.0.0.1**"nel formato richiesto dalla struttura '**sockaddr_in**' e '**connect()**' per connettersi alla porta specificata (**8080**).

```
   int sock = 0;
struct sockaddr_in serv_addr;
char buffer[BUFFER_SIZE] = {0};
char mode;
int choice;

// Creazione del socket
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Socket creation error\n");
    return -1;
}

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

// Conversione dell'indirizzo IP
if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    printf("Invalid address/ Address not supported\n");
    return -1;
}

// Connessione al server
if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
    printf("Connection failed\n");
    return -1;
}
```

3.1.4 Modalità di accesso e gestione delle password

Il client gestisce due modalità di accesso: 'Amministratore' e 'Utente'. L'amministratore deve inserire una password per accedere alle funzionalità avanzate. Per la lettura delle password in modo sicuro da tastiera in ambiente Unix-like è stata utilizzata la libreria '**termios**'. Questa assicura che la password non venga mostrata a schermo mentre l'utente la digita, aumentando così la sicurezza. La funzione '**get_password()**' viene utilizzata per leggere la password senza che venga mostrata. Se la password inserita corrisponde a quella predefinita (definita in '**ADMIN_PASSWORD**'), l'accesso è consentito; altrimenti, l'utente ha un numero limitato di tentativi prima di bloccare l'accesso.
Di seguito il codice della funzione per leggere la password senza mostrarla a schermo:

```
        // Funzione per leggere la password senza mostrarla sullo schermo
void get_password(char *password, int size) {
    struct termios oldt, newt;
    int i = 0;
    int ch;

    // Disabilita l'echo
    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);

    // Legge i caratteri uno per uno
    while (i < size - 1 && (ch = getchar()) != '\n' && ch != EOF) {
        password[i++] = ch;
        printf("*"); // Mostra un asterisco per ogni carattere inserito
    }
    password[i] = '\0';

    // Riabilita l'echo
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    printf("\n");
}
```

Di seguito il codice per la modalità di accesso come 'Amministratore' e inserimento password:

```
#define ADMIN_PASSWORD "0000"
printf("Benvenuto al sistema di gestione dei ricambi.\n");
printf("Seleziona la modalita di accesso:\n");
printf("A. Amministratore\n");
printf("U. Utente\n");
printf("Scelta: ");
scanf("%c", &mode);
getchar(); // per consumare il newline lasciato da scanf

if (mode == 'A' || mode == 'a') {
    char password[20]; // Dimensione della password
    int password_attempts = 3; // Numero massimo di tentativi

    while (password_attempts > 0) {
        printf("Inserisci la password: ");
        get_password(password, sizeof(password));

        if (strcmp(password, ADMIN_PASSWORD) == 0) {
            break; // Esci dal loop se la password è corretta
        } else {
            printf("Accesso negato. Password errata. Riprova.\n");
            password_attempts--;
            if (password_attempts == 0) {
                printf("Hai esaurito tutti i tentativi.
                       Il programma verrà chiuso.\n");
                close(sock);
                return -1;
            }
        }
    }
}
```

3.1.5 Menù e scelte dell'utente

il client presenta un menù interattivo basato sulla modalità di accesso selezionata (Amministratore o Utente). Per ogni modalità, vengono elencate diverse opzioni tra cui scegliere, ognuna associata a un numero che l'utente può inserire per selezionare l'azione desiderata. Il menù 'Amministratore' sarà il seguente:

```
if (mode == 'A' || mode == 'a') {
            printf("\n\nMenu Amministratore:\n");
            printf("1. Aggiungi un nuovo ricambio\n");
            printf("2. Visualizza i ricambi\n");
            printf("3. Aggiorna un ricambio\n");
            printf("4. Elimina un ricambio\n");
            printf("5. Esci\n");
            printf("Scelta: ");
            scanf("%d", &choice);
            getchar(); // per consumare il newline lasciato da scanf
            switch (choice) {
                case 1:
                    create_record(sock);
                    break;
                case 2:
                    read_records(sock, mode);
                    break;
                case 3:
                    read_records(sock, mode);
                    update_record(sock);
                    break;
                case 4:
                    read_records(sock, mode);
                    delete_record(sock);
                    break;
                case 5:
                    close(sock);
                    exit(0);
                default:
                    printf("Scelta non valida.\n");
```

Il menù 'Utente' sarà invece:

```
else if (mode == 'U' || mode == 'u') {
       printf("\n\nMenu Utente:\n");
       printf("1. Visualizza i ricambi\n");
       printf("2. Acquista un ricambio\n");
       printf("3. Esci\n");
       printf("Scelta: ");
       scanf("%d", &choice);
       getchar(); // per consumare il newline lasciato da scanf
```

3.1.6 Implementazione delle operazioni sui ricambi

Il client implementa le seguenti operazioni sui ricambi:

- **Funzione per aggiungere un ricambio**: raccoglie nome, descrizione, quantità e prezzo dallo standard input, costruisce una query SQL per inserire i dati nel database remoto utilizzando '**snprintf()**' e invia la query al server tramite '**send()**'.

```
void create_record(int sock);...
   snprintf(query, BUFFER_SIZE * 2,
       "INSERT INTO Ricambi (Nome, Descrizione, Quantità, Prezzo) VALUES
                            ('%s', '%s', %d, %.2f);",
        name, description, quantity, price);
   send(sock, query, strlen(query), 0);

   char response[BUFFER_SIZE] = {0};
   int valread = read(sock, response, BUFFER_SIZE);
   printf("%s\n", response);
```

- **Funzione per visualizzare i ricambi**: invia una query SQL al server per ottenere tutti i ricambi presenti nel database, riceve e stampa la lista dei ricambi.

```
void read_records(int sock, char mode) {
    char query[] = "SELECT * FROM Ricambi;";
    send(sock, query, strlen(query), 0);
    char response[BUFFER_SIZE] = {0};...
```

- **Funzione per aggiornare i ricambi**: permette di selezionare un ricambio tramite ID, scegliere il campo da aggiornare (nome, descrizione, quantità, prezzo) e inviare una query di aggiornamento al server.

```
 void update_record(int sock);...
// Query per verificare se l'ID esiste nel database
 snprintf(query, BUFFER_SIZE, "SELECT * FROM Ricambi WHERE ID=%d;", id);
    send(sock, query, strlen(query), 0);
    char response[BUFFER_SIZE] = {0};
    int valread = read(sock, response, BUFFER_SIZE);
    if (strcmp(response, "") == 0) {
        printf("L'ID specificato non esiste. Inserisci un ID valido.\n");
        continue; // Richiede all'utente di inserire nuovamente l'ID
    } else {
        break; // Esce dal ciclo se l'ID è valido
    }...
```

```
while (1) {
   printf("\n\nScegli il campo da aggiornare:\n");
   printf("1. Nome\n");
   printf("2. Descrizione\n");
   printf("3. Quantità\n");
   printf("4. Prezzo\n");
   printf("5. Fine\n");
   printf("Scelta: ");
   scanf("%c", &choice);
   getchar(); // per consumare il newline lasciato da scanf
   switch (choice) {
       case '1':
           printf("Inserisci il nuovo nome del ricambio: ");
           fgets(name, BUFFER_SIZE, stdin);
           name[strcspn(name, "\n")] = '\0'; // rimuove il newline
           break;
       case '2':
           printf("Inserisci la nuova descrizione del ricambio: ");
           fgets(description, BUFFER_SIZE, stdin);
           description[strcspn(description, "\n")] = '\0';
           break;
       case '3':
           printf("Inserisci la nuova quantità: ");
           scanf("%d", &quantity);
           getchar(); // per consumare il newline lasciato da scanf
           break;
       case '4':
           printf("Inserisci il nuovo prezzo: ");
           scanf("%lf", &price);
           getchar(); // per consumare il newline lasciato da scanf
           break;
       case '5':
           goto end_update;
       default:
           printf("Scelta non valida. Riprova.\n");...
```

- **Funzione per eliminare un ricambio**: simile all'aggiornamento, ma con l'invio di una query di eliminazione al server.

```
 void delete_record(int sock);...
    snprintf(query, BUFFER_SIZE, "DELETE FROM Ricambi WHERE ID=%d;", id);
// Invio della query al server
    send(sock, query, strlen(query), 0);
    // Lettura della risposta dal server
    char response[BUFFER_SIZE] = {0};
    int valread = read(sock, response, BUFFER_SIZE);
    printf("%s\n", response); // Stampa la risposta del server
```

- **Funzione per acquistare un ricambio**: visualizza i ricambi disponibili, permette all'utente di selezionare un ricambio e la quantità desiderata, calcola il prezzo totale, chiede conferma all'utente e invia una query di acquisto al server se confermato.

```
void buy_record(int sock);...
    snprintf(query, BUFFER_SIZE, "PURCHASE %d %d", id, quantity);
// Invia la richiesta di acquisto solo se l'utente conferma
    send(sock, query, strlen(query), 0);
// Lettura della risposta dal server
    char purchase_response[BUFFER_SIZE] = {0};
    len = read(sock, purchase_response, BUFFER_SIZE);
    if (len > 0) {
        purchase_response[len] = '\0';
         printf("%s\n", purchase_response); // Stampa la risposta del server
                                    con il messaggio di acquisto
    } else {
         printf("Errore durante l'acquisto.\n");
        }
    } else {
        printf("Acquisto annullato.\n");
    }
```

3.1.7 Comunicazione con il server

La comunicazione con il server avviene tramite l'invio di query SQL formate dinamicamente in base all'operazione richiesta dall'utente. Le query vengono inviate utilizzando '**send()**' e le risposte del server vengono lette tramite '**read()**'. Questo processo gestisce sia le richieste di operazioni CRUD che la conferma degli acquisti.

```
// Esempio di invio di una query SQL al server
send(sock, query, strlen(query), 0);

// Esempio di lettura della risposta dal server
char response[BUFFER_SIZE];
int valread = read(sock, response, BUFFER_SIZE);
printf("%s\n", response);
```

3.2 Implementazione server

3.2.1 Tecnologie e strumenti utilizzati

Il server è stato sviluppato in linguaggio C utilizzando le seguenti librerie:

- <**stdio.h**>per le funzioni di input/output standard.
- <**stdlib.h**>per la gestione della memoria e delle allocazioni dinamiche.
- <**string.h**>per le operazioni su stringhe.
- <**unistd.h**>per le funzioni di sistema.
- <**pthread.h**>per la gestione dei thread.
- <**sqlite3.h**>per l'interazione con il database SQLite.
- <**netinet/in.h**>e <**arpa/inet.h**>per la gestione delle connessioni di rete.

3.2.2 Creazione del server

Il server utilizza un socket TCP/IP per ascoltare le connessioni sulla porta 8080. Il server è progettato per gestire fino a 100 connessioni simultanee. Quando un nuovo client si connette, il server accetta la connessione e crea un nuovo thread per gestire la comunicazione con il client.

```
#define PORT 8080
#define MAX_CLIENTS 100
#define BUFFER_SIZE 1024
```

```
  // Creazione del socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Configurazione dell'indirizzo
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
```

```
// Binding del socket
if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
    perror("bind failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}

// Ascolto delle connessioni
if (listen(server_fd, MAX_CLIENTS) < 0) {
    perror("listen failed");
    close(server_fd);
    exit(EXIT_FAILURE);
}...
```

3.2.3 Sincronizzazione e mutua esclusione

Per garantire la consistenza dei dati nel database, il server utilizza un mutex (''**pthread_ mutex_t**' '**db_ mutex**'). Ogni volta che un thread accede al database per eseguire una query, il mutex viene bloccato per assicurare che nessun altro thread possa accedere contemporaneamente al database.

```
sqlite3 *db;
pthread_mutex_t db_mutex; // Mutex per la mutua esclusione sul database
...
pthread_mutex_lock(&db_mutex); // Lock per la mutua esclusione sul database
    if (strncmp(buffer, "PURCHASE", 8) == 0) {
        int id, quantity;
         sscanf(buffer, "PURCHASE %d %d", &id, &quantity);
        execute_purchase(id, quantity, sock);
    } else if (strncmp(buffer, "PRICE", 5) == 0) {
        int id, quantity;
        sscanf(buffer, "PRICE %d %d", &id, &quantity);
         calculate_price(id, quantity, sock);
    } else {
         execute_query(buffer, sock);
    }
    pthread_mutex_unlock(&db_mutex);
    // Unlock per la mutua esclusione sul database
    }

close(sock);
pthread_exit(NULL);
}...
```

3.2.4 Gestione delle richieste dei client

Il server è in grado di gestire diversi tipi di richieste dai client:

- **Acquisto di un ricambio**: Identificato dalla stringa "**PURCHASE**", il server verifica la disponibilità del ricambio, aggiorna la quantità e calcola il prezzo totale.
- **Calcolo del prezzo**: Identificato dalla stringa "**PRICE**", il server calcola e ritorna il prezzo totale per una determinata quantità di ricambi.
- **Esecuzione di query SQL**: Gestisce query SQL generiche, con particolare attenzione alle query di tipo '**SELECT**'.

3.2.5 Esecuzione di query SQL

Le query SQL vengono eseguite in due modalità principali:

- **Query di selezione**: Viene utilizzata la funzione '**sqlite3_prepare_v2()**' per preparare la query, '**sqlite3_step()**' per iterare sui risultati e '**sqlite3_column_text()**' per ottenere i valori delle colonne. I risultati vengono inviati al client formattati come testo.
- **Query di modifica**: Viene utilizzata '**sqlite3_exec()**' per eseguire query di inserimento, aggiornamento o cancellazione. In caso di errore, viene inviata una risposta appropriata al client.

3.2.6 Implementazioni delle funzioni

- **Funzione di gestione delle richieste del client**: Riceve i dati dal client, analizza il tipo di richiesta e chiama le funzioni appropriate ('**execute_purchase**', '**calculate_price**' o '**execute_query**'). Ogni accesso al database è protetto da un lock sul mutex '**db_mutex**'.

```
void *handle_client(void *arg) {
    int sock = *(int *)arg;
    char buffer[BUFFER_SIZE] = {0};
    int valread;
    ...
    // Lock per la mutua esclusione sul database
     pthread_mutex_lock(&db_mutex);
    if (strncmp(buffer, "PURCHASE", 8) == 0) {
         int id, quantity;
         sscanf(buffer, "PURCHASE %d %d", &id, &quantity);
        execute_purchase(id, quantity, sock);
    } else if (strncmp(buffer, "PRICE", 5) == 0) {
        int id, quantity;
        sscanf(buffer, "PRICE %d %d", &id, &quantity);
        calculate_price(id, quantity, sock);
    } else {
        execute_query(buffer, sock);
    }
    // Unlock per la mutua esclusione sul database
    pthread_mutex_unlock(&db_mutex);
    }

    close(sock);
    pthread_exit(NULL);
}
```

- **Funzione del processo d'acquisto**: Questa funzione gestisce il processo di acquisto di un ricambio, recupera la quantità e il prezzo corrente del ricambio dal database. Inoltre verifica la disponibilità della quantità richiesta, aggiorna la quantità nel database e calcola e invia il prezzo totale al client.

```
void execute_purchase(int id, int quantity, int client_sock)...
// Prendiamo il prezzo e la quantità attuale del ricambio
    snprintf(query, BUFFER_SIZE, "SELECT Quantità,
            Prezzo FROM Ricambi WHERE ID=%d;", id);
    if (sqlite3_prepare_v2(db, query, -1, &res, 0) != SQLITE_OK) {
        snprintf(response, BUFFER_SIZE,
                "Failed to execute statement: %s", sqlite3_errmsg(db));
        send(client_sock, response, strlen(response), 0);
        return;...
// Controllo se la quantità del prodotto è
    sufficiente per completare l'acquisto
    if (current_quantity < quantity) {
     snprintf(response, BUFFER_SIZE, "Quantità insufficiente in magazzino.");
     send(client_sock, response, strlen(response), 0);
     return;...
// Aggiorno la quantità
    snprintf(query, BUFFER_SIZE, "UPDATE Ricambi
            SET Quantità=Quantità-%d WHERE ID=%d;", quantity, id);
    if (sqlite3_exec(db, query, 0, 0, 0) != SQLITE_OK) {
        snprintf(response, BUFFER_SIZE,
                "Failed to update quantity: %s", sqlite3_errmsg(db));
        send(client_sock, response, strlen(response), 0);
        return;
    }
// Calcolo ed invio del prezzo totale
    double total_price = price * quantity;
    snprintf(response, BUFFER_SIZE, "Acquisto completato.
            Il prezzo totale è: %.2f euro.", total_price);
    send(client_sock, response, strlen(response), 0);
}...
```

- **Funzione del calcolo per il prezzo finale**: questa funzione calcola il prezzo totale per una determinata quantità di ricambi

```
void calculate_price(int id, int quantity, int client_sock) {
    char query[BUFFER_SIZE];
    char response[BUFFER_SIZE];
    sqlite3_stmt *res;
    double price;
    int current_quantity;

// Step 1: Ricavo la quantità e il prezzo corrente
    snprintf(query, BUFFER_SIZE, "SELECT Quantità,
             Prezzo FROM Ricambi WHERE ID=%d;", id);
    if (sqlite3_prepare_v2(db, query, -1, &res, 0) != SQLITE_OK) {
        snprintf(response, BUFFER_SIZE, "Failed to execute statement:
                 %s", sqlite3_errmsg(db));
        send(client_sock, response, strlen(response), 0);
        return;
    }...



// Step 2: Controlla se ci sono abbastanza ricambi
if (current_quantity < quantity) {
    snprintf(response, BUFFER_SIZE, "Quantità insufficiente in magazzino.");
    send(client_sock, response, strlen(response), 0);
    return;
    }

// Step 3: Calcola il prezzo totale
    double total_price = price * quantity;
    snprintf(response, BUFFER_SIZE,
    "Il prezzo totale per %d pezzi è: %.2f euro.", quantity, total_price);
    send(client_sock, response, strlen(response), 0);
}
```

- **Funzione della gestione delle query**: Gestisce le query SQL generiche. Se la query è di tipo '**SELECT**', prepara ed esegue la query, iterando sui risultati e inviandoli al client. Per le altre query, esegue direttamente la query e invia una risposta al client.

```
void execute_query(char *query, int client_sock) {
    char *err_msg = 0;
    sqlite3_stmt *res;

    if (strstr(query, "SELECT") != NULL) {
        int rc = sqlite3_prepare_v2(db, query, -1, &res, 0);
        if (rc != SQLITE_OK) {
            char response[BUFFER_SIZE];
            snprintf(response, BUFFER_SIZE, "Failed to execute statement:
                    %s", sqlite3_errmsg(db));
            send(client_sock, response, strlen(response), 0);
            return;
        }...
```

3.3 Database SQLite

SQLite è una libreria C che fornisce un database SQL leggero e full-featured. A differenza dei tradizionali sistemi di gestione di database relazionali (RDBMS), SQLite è incorporato all'interno dell'applicazione che utilizza il database. Nel contesto della gestione dei ricambi il database, chiamato '**magazzino.db**', viene utilizzato per memorizzare i dati relativi ad essi.

3.3.1 Creazione database

Questo script crea il database ed una tabella chiamata '**Ricambi**' con i seguenti campi:

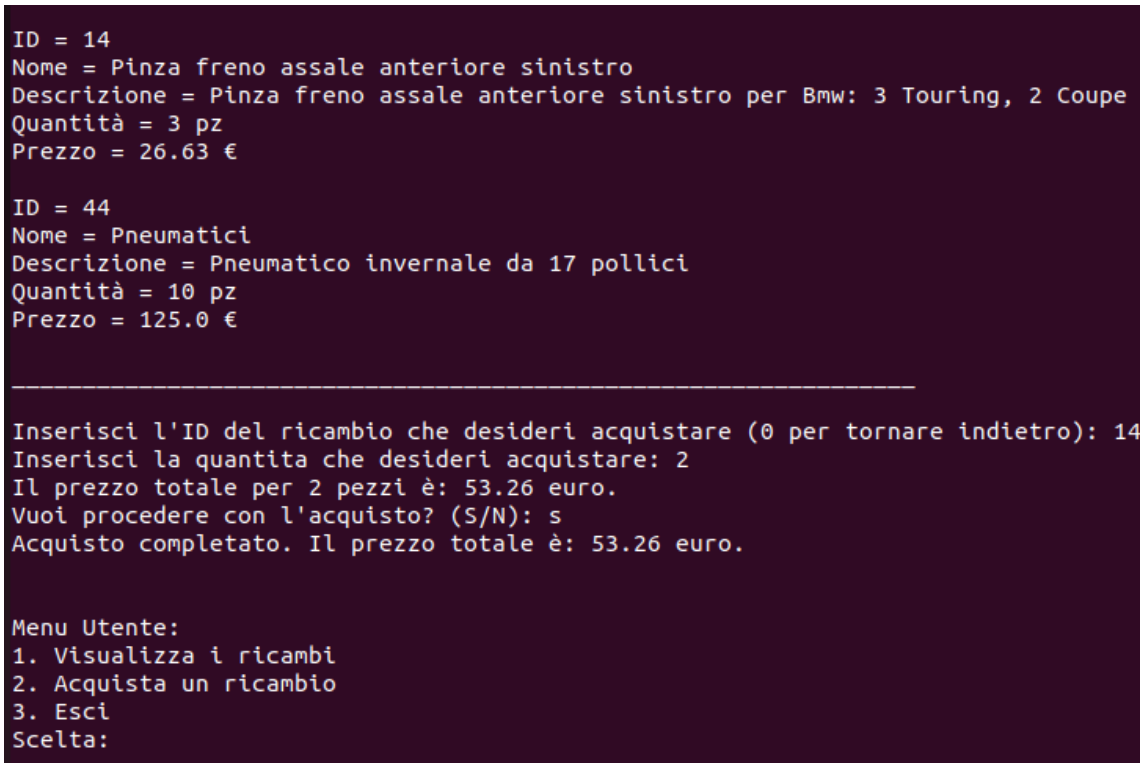
- **ID**: Chiave primaria.
- **Nome**: Nome del ricambio.
- **Descrizione**: Descrizione del ricambio.
- **Quantità**: Quantità disponibile in magazzino.
- **Prezzo**: Prezzo unitario del ricambio.

```
...
int rc = sqlite3_open("magazzino.db", &db);
...
const char *sql = "CREATE TABLE IF NOT EXISTS Ricambi("
                      "ID INTEGER PRIMARY KEY AUTOINCREMENT, "
                      "Nome TEXT, "
                      "Descrizione TEXT, "
                      "Quantità INTEGER, "
                      "Prezzo REAL);";
}
```

4. Risultati sperimentali

I risultati ottenuti dalla realizzazione del sistema multithreading client/server per la gestione da remoto di un magazzino ricambi, confermano uno degli obiettivi principali del progetto, ovvero, quello di garantire la gestione concorrente delle richieste client tramite un approccio multithreading.
Per verificare il corretto funzionamento del multithreading, sono stati eseguiti test in cui, ad esempio, più client tentano di acquistare simultaneamente lo stesso ricambio dal database andando a superare la quantità di ricambi presenti nel magazzino.

Risultati dell'acquisto correttamente riuscito da parte del primo utente:



Risultati dell'acquisto fallito da parte del secondo utente a causa della terminazione del ricambio:

```
ID = 14
Nome = Pinza freno assale anteriore sinistro
Descrizione = Pinza freno assale anteriore sinistro per Bmw: 3 Touring, 2 Coupe
Quantità = 3 pz
Prezzo = 26.63 €

ID = 44
Nome = Pneumatici
Descrizione = Pneumatico invernale da 17 pollici
Quantità = 10 pz
Prezzo = 125.0 €

------------------------------------------------------------

Inserisci l'ID del ricambio che desideri acquistare (0 per tornare indietro): 14
Inserisci la quantita che desideri acquistare: 2
Il prezzo totale per 2 pezzi è: 53.26 euro.
Vuoi procedere con l'acquisto? (S/N): s
Quantità insufficiente in magazzino.


Menu Utente:
1. Visualizza i ricambi
2. Acquista un ricambio
3. Esci
Scelta:
```

Grazie all'uso di meccanismi di mutua esclusione **(mutex)**, ogni thread accede in modo sicuro alla sezione critica in cui viene modificato il database. Il primo thread che acquisisce il mutex controlla la disponibilità del ricambio e, se sufficiente, procede con l'aggiornamento del database. Il secondo thread, una volta acquisito il mutex, verifica nuovamente la disponibilità e aggiorna il database in base alle unità residue.

Questo garantisce la coerenza dei dati nel database SQLite e l'affidabilità delle operazioni CRUD, anche in scenari di elevata concorrenza.

In generale le principali operazioni CRUD sono state progettate per garantire la mutua esclusione, prevenendo **race condition**.

Nello specifico tramite la funzione *'**handle_client**'*:

- **CREATE**: L'operazione di aggiunta di nuovi ricambi nel magazzino è protetta da mutex per evitare duplicati o inserimenti incoerenti quando più thread tentano di creare contemporaneamente lo stesso ricambio.
- **READ**: La lettura dei dati è generalmente non bloccante, tuttavia il controllo per la mutua esclusione viene effettuato anche per questa operazione soprattutto nel momento in cui bisogna l'operazione di lettura deve essere sincronizzata con quella di scrittura.
- **UPDATE**: L'aggiornamento delle informazioni sui ricambi, come l'aumento o la diminuzione delle scorte, avviene in modo esclusivo per garantire che le modifiche siano atomiche.
- **DELETE**: La rimozione di ricambi dal database è gestita con mutex per evitare che operazioni di lettura o aggiornamento concorrenti accedano a dati ormai non validi.

5. Conclusioni e sviluppi futuri

il progetto di gestione da remoto di un magazzino di ricambi ha dimostrato l'efficacia di un sistema client-server basato su socket TCP/IP per la gestione centralizzata e l'interazione remota con un database. Garantendo,soprattutto, la gestione concorrente delle risorse tramite il multithreading. Il progetto pone solide basi per delle future implementazioni, come:

- **Implementazione di una GUI più intuitiva.**
- **Servizio di resi e rimborsi dei prodotti acquistati.**
- **Implementazione di una sezione 'carrello' per acquisti multipli.**
- **Implementazione di Login/Registrazione.**