UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Sistema Client-Server di un ospedale per la gestione di dati di pazienti, con multithreading.

Docente:
Prof.Roberto Marino

Studente:
Thilina P.A. Perera, 514553

ANNO ACCADEMICO 2023/2024

Indice

1 Introduzione **3**

2 Architettura del Sistema **4**
2.1 Client-Server 4
2.2 MongoDB 5
2.2.1 connessione al db 5
2.2.2 Data Model 6

3 Client **8**
3.1 Comandi 9
3.2 le operazioni crud lato client 11
3.3 ClientUtils.c 11

4 Server **11**
4.1 Multithreading 12
4.2 sezione critica e struct Oid_mutex 16
4.3 Operazioni CRUD 16
4.4 Logging 16

5 Conclusioni **17**

1 Introduzione

Il nostro progetto si concentra sulla realizzazione di un server multi-threaded, progettato per gestire connessioni concorrenti da parte di client remoti. Il server si collega ad un Database mongo che contiene un Datamodel da noi progettato per il caso studio, e gestirà le diverse operazioni CRUD richieste dai client. Nel corso di questa relazione, esploreremo: l'architettura del client-server,la sua connessione al database MongoDB, il processo di gestione delle richieste dei client, la procedura di logging delle operazioni, e il sistema di comandi lato client, il datamodel scelto e le possibilità che ci offre rispetto ad un database relazionale rigido. Inoltre, discuteremo delle scelte effettuate durante la progettazione e le sfide affrontate durante lo sviluppo.

2 Architettura del Sistema

2.1 Client-Server

Le Componenti Principali del Sistema sono:

- **Client**:
  - Il client è l'interfaccia utente attraverso la quale gli utenti interagiscono con il sistema. È responsabile della presentazione dei dati all'utente e dell'invio delle richieste al server
  - Comunica con il server tramite socket TCP/IP.
  - Permette all'utente di costruire le proprie richieste grazie ad un sistema di comandi attraverso l'interfaccia a linea di comandi, discusso nel paragrafo 3.1 Comandi.
- **Server**:
  - *Il server è il motore centrale del sistema, responsabile della gestione delle richieste dei client, dell'elaborazione dei dati e della comunicazione con il database MongoDB.*
  - Accetta connessioni multiple dai client utilizzando la programmazione multithreading per garantire un'elaborazione efficiente delle richieste.
  - Si basa sul protocollo TCP/IP per comunicare con i client e utilizza il formato BSON per scambiare dati con il database.

Strumenti utilizzati:

- Mongodb (mongoshell) :
  - Libreria per interfacciarsi al db Mongo con programmi in C.
- MongoC driver :
  - Libreria per interfacciarsi al db Mongo con programmi in C.
- libreria BSON (Binary JSON):
  - Il Database mongo archivia i dati in formato BSON e li mostra sotto all'utente in formato JSON.
  - è un formato leggero e per rappresentare i documenti JSON in forma binaria.
- libreria sys/socket.h:
  - libreria standard per costruire socket e far comunicare un client con un server, nel nostro caso in localhost o LAN.
- Libreria TCP/IP:
  - è il protocollo di rete più utilizzato.

  - <netinet/in.h> : gestione dell'indirizzamento ad internett della comunicazione TCP/IP.
  - <arpa/inet.h> : per la conversione degli indirizzi della comunicazione TCP/IP.

- Pthread (POSIX thread) :
  - Libreria posix per il threading in C, fornisce le funzioni per la gestione di thread, mutex e altri strumenti per le eventuali sezioni critiche.

2.2 MongoDB

MongoDB rappresenta un tipo di database NoSQL orientato ai documenti, a differenza dei database SQL che sono focalizzati sul concetto di relazioni e tabelle classiche. In MongoDB, l'unita do organizzazione dei dati principale è la "Collection" (ovvero le "Tabelle" in SQL) che raggruppa dati sotto forma di "Documenti"("record" e "tuple" in Sql). Un documento è costituito da coppie key-value che formano uno schema dinamico. Ciò implica che i documenti all'interno di una collezione possono variare nei campi e assumere valori come: stringhe, int, date o ulteriori documenti. Non è necessario seguire uno schema rigido essendo un DB non-strutturato, nel nostro progetto questa caratteristica viene sfruttata per ottenere una funzionalità(discussa nel paragrafo 2.2.1 Datamodel) che non poteva essere implementata in un database SQL classico per via della sua rigidità strutturale. MongoDB è particolarmente adatto per gestire dati con numerose caratteristiche, senza necessariamente avere correlazioni tra di esse. Questi dati sono immagazzinati come file BSON (Binary JavaScript Object Notation), derivati dai file JSON (JavaScript Object Notation), ed è la sua versione codificata in binario per un'elaborazione più veloce.

2.2.1 connessione al db

La connessione al db è concessa solo al server e viene implementata con la seguente funzione:

```
1     void initialize_mongodb() {// Funzione di
          inizializzazione del client MongoDB
2         mongoc_init();
3         client =
              mongoc_client_new("mongodb://localhost:27017");
4         if (!client) {
5             fprintf(stderr, "Failed to create MongoDB
                  client\n");
6             exit(EXIT_FAILURE);
7         }
8         mongoc_client_set_appname(client,
              "server-medical-dimension");
9     }
10
```

```
    // Funzione di pulizia del client MongoDB
    void cleanup_mongodb() {
        mongoc_client_destroy(client);
        mongoc_cleanup();
    }
```

funzione che viene richiamata all'avvio del server prima ancora di accettare connessioni da parte del client. Sotto è presente anche un funzione di cleanup che libera le risorse per la connessione al db una volta finita l'esecuzione.

2.2.2 Data Model

Il documento JSON che segue rappresenta il modello di dati con cui abbiamo scelto di organizzare i dati per la gestione delle informazioni sanitarie dei pazienti, con dati che hanno relazioni tra loro,ma vengono rappresentate da annidamenti quindi una rappresentazioni meno rigida e più "logica". Il documento rappresenta tutte le informazioni relative ad un paziente, e le "tabelle logiche" sono:

- **dati_anagrafici (del paziente)**: Questa sezione contiene informazioni anagrafiche essenziali sui pazienti, inclusi nome, cognome, data di nascita, sesso, codice fiscale e residenza. Questi dettagli sono fondamentali perchè forniscono il quadro completo dell'identità del paziente.
- **cartelle_cliniche**: Le cartelle cliniche rappresentano le informazioni sanitari effettive, rappresentano un registro dati di un ricovero avvenuto in un arco temporale per un singolo paziente, offrendo uno spazio strutturato per registrare eventi clinici, dettagli di ricovero e altre informazioni mediche.
- **eventi (clinici)**: gli eventi sono la tabella più particolare del progetto, rapprensentano gli eventi effettivi accaduti in un ricovero, sono composti principalmente da tre campi standard ma grazie al modello di dati e al db scelto, è possibile aumentare il numero di "colonne" di questa tabella, rendendo l'evento più o meno dettagliato in base alle esigenze, facendo coesistere in un db, dei record che presentano dei campi diversi e un numero di campi.

```
{
    _id: ObjectId('65c9ffaaf54d354285626c42'),
    dati_anagrafici: {
      nome: 'Giuliano',
      cognome: 'Venuto',
      data_di_nascita: '01-01-2001',
      sesso: 'M',
      codice_fiscale: 'HRJ7DHRLYSN4NDU4',
      residenza: 'Giardini'
    },
    cartelle_cliniche: [
        {
          numero_cartella: 1,
```

```
            inizio_ricovero: [ '10-12-23', '08:00' ],
            eventi_clinici: [
              {
                tipo: 'somministrazione',
                descrizione: 'Somministrazione di farmaco X',
                data: [ '11-12-23', '10:00' ]
              },
              {
                tipo: 'esame pressione arteriosa',
                descrizione: 'Calcolo pressione paziente',
                data: [ '12-12-23', '10:00' ],
                misurazione: '110 mmHg',
                categoria: 'ottimale'
              }
            ],
            fine_ricovero: [ '12-12-23', '16:00' ]
          }
    ]
}
```

3 Client

Il client è stato implementato utilizzando il linguaggio di programmazione C e la libreria standard di socket per la comunicazione di rete. Il client si connette al server tramite un socket TCP/IP e invia richieste al server per eseguire varie operazioni sul database MongoDB. Il codice del client è suddiviso principalmente in diverse funzioni che gestiscono la connessione al server e l'invio delle richieste. il seguente codice ruguarda la connessione a server (localhost o LAN):

```
#define PORT 7799

// funzione per la connessione al server da parte del client
int connessione_al_server() {
    // Dichiarazione di un array di socket
    int sockets = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in address;

    if (sockets < 0) {
        printf("Errore durante la creazione del socket\n");

        return 1;
    }


    // Impostazione dell'indirizzo del server (a scelta tra localhost e lan)
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(PORT);
    char *ip_localhost = "127.0.0.1";
    char *ip_lan = "192.168.128.215";
    if (inet_pton(AF_INET, ip_localhost, &address.sin_addr) <= 0) {
        printf("Indirizzo del server non valido\n");

        return 1;
    }

    // Connessione al server
    if (connect(sockets, (struct sockaddr *)&address, sizeof(address)) < 0) {
        printf("Errore durante la connessione al server\n");
        return 1;
    }

    printf("Connesso al server.\n");
    return sockets;
}
```

Il client è in grado di eseguire diverse operazioni (CRUD) sul database MongoDB tramite richieste inviate al server, tra cui:

- Lettura di Documenti: Il client può richiedere al server di leggere documenti dal database MongoDB in base a determinati criteri di query.
- Creazione di Documenti: È possibile creare nuovi documenti nel database inviando i dati appropriati al server.
- Aggiornamento di Documenti: Il client può richiedere al server di aggiornare i documenti esistenti nel database effettuando degli inserimenti sulle cartelle cliniche.
- Eliminazione di Documenti: È possibile eliminare documenti dal database inviando le relative istruzioni al server.

Il client presenta un'interfaccia testuale semplice che consente all'utente di inserire comandi e dati da inviare al server. Dopo aver inviato una richiesta, il client visualizza la risposta ricevuta dal server, se presente, per fornire un feedback all'operatore sanitario sul risultato dell'operazione.

3.1 Comandi

Abbiamo creato un insieme di comandi con cui l'utente può interagire con il client. Questi comandi sono salvati in un file chiamato commands.c, e sono organizzati in un array di stringhe che rapprensentano il comando e il comando alternativo (ad esempio -c e create) . Inoltre, nel file commands.c sono presenti altre funzioni utili per il parsing dei comandi. Il file commands.c contiene anche un manuale che illustra all'utente come utilizzare i comandi disponibili. Di seguito è riportato il file:

```
// Funzione di parsing per array di array in C
int parseArrayAndGetIndex(char *commands[][2], int
    numCommands, char *commandToFind) {
    for (int i = 0; i < numCommands; ++i) {
        if (strcmp(commands[i][0], commandToFind) == 0 ||
            strcmp(commands[i][1], commandToFind) == 0) {
            return i + 1;  // Restituisce l'indice
                dell'array in cui il comando    stato
                trovato
        }
    }
    return -1;  // Nessuna corrispondenza trovata
}

const char *cli_manuale = "intero manuale che non riporto
    nella relazione per questioni di spazio, ma verr
    mostrato un immagine della sua vista su terminale";


void printManual() {
    printf("%s\n", cli_manuale);
}
```

```
17
18
19  int commandNumber(char *commandToFind) {
20      // all commands
21      char *commands[][2] = {
22          {"-man", "manuale"},
23          {"-c", "create"},
24          {"-r", "read"},
25          {"-u", "update"},
26          {"-d", "delete"},
27          {"-p", "prova"}
28      };
29
30      int numCommands = sizeof(commands) /
            sizeof(commands[0]);
31
32      int commandIndex = parseArrayAndGetIndex(commands,
            numCommands, commandToFind);
33
34      return commandIndex;
35  }
36  ```
```

di seguito la vista su terminale del manuale accessibile dal client con il comando cli -man o cli manuale:

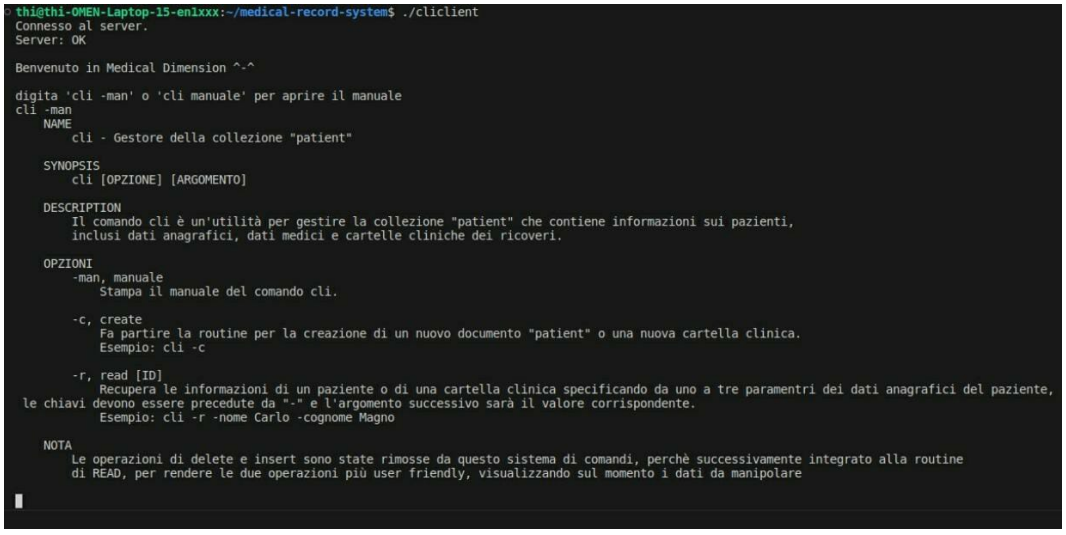

Figura 1: Manuale dei comandi su CLI.

3.2 le operazioni crud lato client

le quattro operazioni crud lato client hanno approcci diversi nel richiedere dati all'utente, i codici sono stati pubblicati sul repository github. qui sotto una breve descrizione

- READ: prima ricerca filtrata per un massimo di tre paramentri -> stampa della lista dei pazienti trovati –>scelta di un paziente e scelta della sezione da visualizzare(anagrafica o cartelle cliniche –> visualizzazione finale dei dati scelti.
- CREATE: richiede all'utente una serie di dati anagrafici in ordine(con controllo sui dati inseriti + conferma del dato) –> stampa dei dati inseriti e richiesta di conferma finale –> creazione di un nuovo paziente(nuovo documento).
- UPDATE: è l'inserimento di un nuovo evento in una cartella clinica esistente oppure creazione di una nuova cartella clinica, richiede prima l'operazione READ fino all'istanza di stampa delle cartelle cliniche, dopo di che viene richiesto se si vuole "effettuare un operazione speciale" –> richiede i dati in ordine e poi li stampa –> inserimento nuovo evento. In particolare ci saranno due cicli dove il primo richiede se si vuole creare un evento, e il secondo situato all'interno del primo ciclo richiede se si vuole aggiunge un nuovo campo e un suo valore corrispondente.
- REMOVE: eliminazione di un paziente dal sistema, richiede prima l'operazione READ fino all'istanza di stampa dei dati anagrafici del paziente, dopo di che viene richiesto se si vuole "effettuare un operazione speciale" –> dopo aver confermato viene chiesto se si vuole proseguire alla rimozione –> il paziente viene rimosso.

3.3 ClientUtils.c

4 Server

Anche il server è stato implementato utilizzando il linguaggio di programmazione C e la libreria standard di socket per la comunicazione di rete. Si tratta di un server multi-threaded in grado di gestire più connessioni simultanee. Il codice del server è suddiviso principalmente in due parti: l'inizializzazione del server e la gestione delle connessioni in entrata.

- Inizializzazione del Server: in questa parte del codice, vengono creati il socket del server e viene eseguita la bind tra i ip del localhost(o LAN) e alla porta specificata. Successivamente, il server inizia ad ascoltare sul socket per le connessioni in entrata, e inizializzazione della connessione al db come descritto precedentemente, un ulteriore inizializzazione è quella della procedura di mutex (che verrà descritta nel paragrafo Multithreading).

– Gestione delle Connessioni: Quando una nuova connessione viene accettata, il server crea un thread dedicato per gestire la comunicazione con il client. Questo thread si occupa di ricevere i dati inviati dal client, interpretarli e eseguire le operazioni richieste.

```
    // Creazione del socket
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        printf("Errore durante la creazione del socket\n");
        return 1;
    }

    // Impostazione delle opzioni del socket
    int opt = 1;
    if (setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR
        | SO_REUSEPORT, &opt, sizeof(opt))) {
        printf("Errore durante l'impostazione delle
            opzioni del socket\n");
        return 1;
    }

    // Impostazione dell'indirizzo locale del socket
    struct sockaddr_in address;
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(7799);

    if (bind(server_socket, (struct sockaddr*)&address,
        sizeof(address)) < 0) {
        printf("Errore durante la bind del socket\n");
        return 1;
    }

    // Inizio ad ascoltare su una porta
    if (listen(server_socket, MAX_CONNECTIONS) < 0) {
        printf("Errore durante la listen del socket\n");
        return 1;
    }
    printf("Server in ascolto..\n");



```

4.1 Multithreading

Il multithreading è un concetto fondamentale che si riferisce alla capacità di un'applicazione di eseguire più thread di esecuzione simultaneamente all'interno di un singolo processo.. Utilizzando il multithreading, è possibile ottenere un miglioramento delle prestazioni e dell'efficienza, in quanto i thread

possono lavorare in parallelo su compiti separati.Tuttavia, la programmazione multithread può essere complessa poiché i thread condividono lo stesso spazio di memoria e possono interferire l'uno con l'altro se non sincronizzati correttamente.

codice che accetta le connessioni e crea un thread indipendente per ogni richiesta:

```
1       ...
2       printf("Server in ascolto..\n");
3
4
5     // Accettazione di connessioni in entrata
6       int client_socket;
7       struct sockaddr_in client_address;
8       int addrlen = sizeof(client_address);
9
10      while (server_running) {
11          client_socket = accept(server_socket, (struct
                sockaddr*)&client_address,
                (socklen_t*)&addrlen);
12          if (client_socket < 0) {
13              printf("Errore durante la accept del
                    socket\n");
14              continue;
15          }
16
17          pthread_t thread;
18          int* new_sock = malloc(sizeof(int));
19          *new_sock = client_socket;//socket di
                comunicazione tra server e client
20
21          if (pthread_create(&thread, NULL,
                handle_connection, (void*)new_sock) < 0) {
22              printf("Errore durante la creazione del
                    thread\n");
23              free(new_sock);
24              continue;
25          }
26
27          pthread_detach(thread);// i thread una volta
                finita l'esecuzione rilasceranno in automatico
                le risorse, senza aspettare un join con altri
                thread
28      }
```

Codice della funzione che viene eseguita come thread, che gestisce le richieste del client:

```
1         void* handle_connection(void* arg) {
2     int client_socket = *(int*)arg;
3     free(arg);
4
5     char ack_msg[1024] = {0};
6     recv(client_socket, ack_msg, sizeof(ack_msg), 0);
7     printf("Client: %s\n", ack_msg); // il messaggio
          dovrebbe essere OK in caso di successo
8
9     // Invio di una conferma al client
10    char* confirm_msg = "OK\n";
11    write(client_socket, confirm_msg, strlen(confirm_msg));
12
13    printf("Connessione accettata\n");
14                            char buffer[2048];
15                            recv(client_socket, buffer,
                                  sizeof(buffer), 0);
16                            char *flagged_query = buffer;
17
18                            char *flag;
19                            char *query;
20                            flag = strtok(flagged_query,
                                  "^");
21                            query = strtok(NULL, "^");
22                            size_t query_size =
                                  sizeof(query);
23                            int query_len = strlen(query);
24
25
26                        // Verifica l'operazione ricevuta
27                        if (strcmp(flag, "READ") == 0) {
28                            // esecuzione routine di read
29                            pthread_exit(NULL);
30                        } else if (strcmp(flag, "CREATE")
                              == 0) {
31                            // esecuzione routine di create
32                            pthread_exit(NULL);
33                        } else if (strcmp(flag, "UPDATE")
                              == 0) {
34                            // esecuzione routine di update
35                            pthread_exit(NULL);
36                        } else if (strcmp(flag, "REMOVE")
                              == 0) {
37                            // Gestisci operazione REMOVE
38                            pthread_exit(NULL);
39                        }
40
41                        // Invio di una conferma al client
42                        write(client_socket, confirm_msg,
                              strlen(confirm_msg));
```

```
43
44                          // Chiusura del socket
45                              close(client_socket);
46                              pthread_mutex_unlock(&lock);
47                              pthread_exit(NULL);
48      return NULL;
49  }
```

4.2 sezione critica e struct Oid_mutex

Nel nostro sistema per evitare problemi dovuti alla concorrenza , basta dare la possibilità ai client di effettuare operazioni di lettura ,aggiornamento e rimozione su un paziente alla volta, questo vuol dire che bisogna ogni volta verificare se un thread sta già effettuando un operazione su un Object id (id che identificano i documenti) specifico, se è già occupato il client deve rimanere in attesa fino a quando non verrà rilasciato il mutex per quel documento. questo è possibile grazie ad una struttura dati che è stata definita per il nostro caso, insieme ad un insieme di funzioni di inizializzazione, occupazione, rilascio e verifica dei mutex.

di seguito viene mostrato lo struct per la gestione della sezione critica:

```
                #define MAX_CONNECTION 30
                #define OID_LENGTH 24

                typedef struct {
                    pthread_mutex_t mutex;
                    int is_locked;
                    char Oid[OID_LENGTH];
                } OidMutex;

```

4.3 Operazioni CRUD

Le operazioni crud lato server si limitano alla creazione di query in BSON con i dati ricevuti controllati lato client, una volta creata la query che rispetti i criteri del datamodel che abbiamo progettato, viene effettuata l'operazione vera e propria sul database Mongo a cui il server è collegato. Anche qui suggeriamo la lettura del codice completo su github.

4.4 Logging

Durante le operazioni CRUD lato server in caso di successo dell'operazione, viene creato un documento di logging che viene inserito in una collezione apparte chiamata "log" che cambia struttura in base all'operazione , dove la struttura più classica è id_documento manipolato , timestamp e tipo di operazione. Nel caso più specifico di un operazione REMOVE , viene anche salvato l'intero documento eliminato (per un eventuale recupero dei dati), mentre per l'operazione UPDATE viene salvato il nuovo campo inserito.

esempio di struttura di un documento log:

```
            {
                _id: ObjectId('65c9ffaaf54d354285626c43'),
                operazione: 'CREATE',
                timestamp: ISODate('2024-02-12T11:23:22.000Z'),
                patient_code: 'HRJ7DHRLYSN4NDU4'
            }
```

5 Conclusioni

Nel corso dello sviluppo del progetto sono state riscontrate problematiche che sono state risolte, cambiando anche approccio su alcune delle funzionalità del sistema. Ma il sistema finale può essere migliorato sotto alcuni punti di vista, sicuramente l'aggiunta di un sistema di accesso autorizzato e una suddivisione in ruoli degli utenti, per rendere alcune operazioni privilegiate. Un intefaccia GUI user-friendly risulterebbe migliore per gli utilizzatori del caso studio, ovvero operatori sanitari. E in generale una suite di comandi più ampia che dia la possibilità di manipolare i dati sfruttando al meglio il tipo di database utilizzato, con ovviamente le dovute procedure di gestione date proprio dalla struttura flessibile di mongo.Un altro approccio rispetto al primo caso potrebbe essere la scelta di un db relazionale classico, con rigidità strutturale alta a discapito delle peculiarità sperimentate nel nostro progetto, che però risulterebbe più conveniente per la gestione di dati in ambito sanitario o casi studi simili.