Università degli Studi di Messina

**Dipartimento di Scienze Matematiche e Informatiche,
Scienze Fisiche e Scienze della Terra
Corso di Laurea in Informatica
(Laurea Triennale L-31)**

Realizzazione di un Server REST concorrente in C che utilizza come backend Redis

Elaborato di :
Ludovico Passari
Mattia Musarra Tubbi

Docente:
Dott. Roberto Marino

Anno Accademico 2024-2025

Indice

1 Introduzione

Lo sviluppo di applicazioni web moderne richiede architetture capaci di gestire un elevato numero di richieste simultanee mantenendo prestazioni ottimali e garantendo la coerenza dei dati. In questo contesto, i servizi REST rappresentano uno standard consolidato per la comunicazione tra client e server, offrendo un'interfaccia uniforme e scalabile per l'accesso alle risorse.

1.1 Obiettivi

Il presente report documenta la progettazione e implementazione di un server REST concorrente sviluppato in linguaggio C, sfruttando Redis come sistema di backend per la persistenza e gestione dei dati. La scelta del linguaggio C, pur presentando maggiori complessità rispetto a linguaggi di più alto livello, consente un controllo granulare sulle risorse di sistema e prestazioni superiori, caratteristiche fondamentali per applicazioni ad alta concorrenza. L'architettura proposta integra diversi paradigmi di programmazione concorrente per massimizzare l'utilizzo delle risorse hardware disponibili. L'implementazione sfrutta thread multipli per gestire simultaneamente le richieste HTTP in arrivo, mentre Redis funge da layer di persistenza ad alte prestazioni, garantendo operazioni di lettura e scrittura rapide attraverso la sua natura in-memory e le sue strutture dati ottimizzate. Il progetto affronta diverse sfide tecniche significative, tra cui la gestione della concorrenza senza compromettere l'integrità dei dati, l'implementazione di alcune utility per interagire con HTTP in C, l'integrazione con l'API di Redis(hiredis), e l'ottimizzazione delle prestazioni sotto carico elevato. L'obiettivo principale è dimostrare come sia possibile realizzare un'infrastruttura server efficiente e scalabile utilizzando tecnologie a basso livello, ottenendo prestazioni competitive rispetto a soluzioni più tradizionali basate su framework di alto livello. Il lavoro si propone inoltre di esplorare le best practices per lo sviluppo di sistemi concorrenti in C, evidenziando vantaggi, limitazioni e trade-off delle scelte architetturali adottate. Questo documento presenta l'analisi dei requisiti, le decisioni progettuali, i dettagli implementativi e una valutazione sperimentale delle prestazioni del sistema sviluppato, fornendo una panoramica completa del processo di sviluppo e dei risultati ottenuti.

2 Descrizione del problema

Nel prsente report verrà trattato la progettazione e l'implementazione di un server REST in linguaggio C per accogliere e inviare risposte HTTP. Il sever implementato utilizza come layer di persistenza dei dati il database in-memory Redis.

3 Stato dell'arte

3.1 Paradigma client-server

Il paradigma client-server rappresenta un modello architetturale distribuito che definisce la modalità di interazione tra entità software attraverso una rete di comunicazione. In questa architettura, il client assume il ruolo di richiedente di servizi, mentre il server fornisce risorse e servizi in risposta alle richieste ricevute. Questa separazione delle responsabilità costituisce il principio fondamentale che governa tutte le interazioni del sistema. L'architettura client-server si basa su una distribuzione asimmetrica delle funzionalità, dove il server centralizza la logica di business, la gestione dei dati e i servizi condivisi, mentre il client si occupa della presentazione delle informazioni e dell'interfaccia utente. Questa distinzione permette di ottimizzare l'utilizzo delle risorse e di scalare il sistema in modo efficiente secondo le necessità operative.

3.2 Separazione delle Responsabilità

La caratteristica distintiva del paradigma client-server risiede nella netta separazione delle responsabilità tra le due entità. Il server assume la responsabilità della gestione centralizzata dei dati, implementando meccanismi di persistenza, controllo dell'accesso e validazione delle operazioni. Questa centralizzazione garantisce coerenza dei dati e semplifica la manutenzione del sistema, poiché la logica di business risiede in un unico punto di controllo. Il client, dal canto suo, si concentra sulla presentazione delle informazioni e sull'interazione con l'utente finale. Questa specializzazione permette di ottimizzare ciascun componente per il proprio dominio specifico, risultando in un'architettura complessivamente più efficiente e mantenibile.

3.3 Modello di Comunicazione

La comunicazione tra client e server segue un modello basato su richieste e risposte, dove il client inizia sempre l'interazione inviando una richiesta specifica al server. Il server elabora la richiesta, esegue le operazioni necessarie e restituisce una risposta contenente i risultati dell'elaborazione o informazioni sullo stato dell'operazione. Questo modello di comunicazione è intrinsecamente asincrono, permettendo al client di continuare altre operazioni mentre attende la risposta del server. La natura stateless delle interazioni HTTP amplifica questa caratteristica, rendendo ogni richiesta indipendente dalle precedenti e semplificando la gestione della concorrenza.

3.4 Scalabilità e Prestazioni

Il paradigma client-server offre eccellenti caratteristiche di scalabilità attraverso diversi meccanismi. La scalabilità verticale permette di potenziare le risorse hardware del server per gestire carichi di lavoro crescenti, mentre la scalabilità orizzontale consente di distribuire il carico tra server multipli attraverso tecniche di load balancing e clustering. La centralizzazione delle risorse computazionali nel server permette di ottimizzare l'utilizzo

dell'hardware, concentrando potenza di calcolo e capacità di storage dove sono maggiormente necessarie. I client possono operare con risorse hardware limitate, delegando le operazioni computazionalmente intensive al server.

3.5 Sicurezza Centralizzata

La centralizzazione dei dati e della logica di business nel server facilita l'implementazione di politiche di sicurezza coerenti e complete. I meccanismi di autenticazione, autorizzazione e auditing possono essere implementati una sola volta nel server e applicati uniformemente a tutti i client che accedono al sistema. Questa approccio riduce significativamente la superficie di attacco del sistema, poiché i dati sensibili rimangono centralizzati e protetti da meccanismi di sicurezza dedicati. I client accedono solamente alle informazioni strettamente necessarie per le loro funzioni specifiche.

3.6 Collo di Bottiglia del Server

La centralizzazione delle risorse nel server può creare potenziali colli di bottiglia quando il numero di client simultanei supera la capacità di elaborazione del server. Questo problema diventa particolarmente critico in scenari ad alta concorrenza, dove il server deve gestire migliaia di richieste simultanee mantenendo tempi di risposta accettabili.

3.7 Evoluzione verso Architetture Moderne

Il paradigma client-server tradizionale ha subito significative evoluzioni per adattarsi alle esigenze delle applicazioni moderne. L'introduzione di architetture a microservizi ha frammentato il server monolitico in servizi specializzati e indipendenti, migliorando la scalabilità e la mantenibilità del sistema.

3.8 Epoll: Gestione Avanzata degli Eventi nei Sistemi Linux

Epoll rappresenta un meccanismo di notifica degli eventi specificamente progettato per i sistemi Linux, introdotto nel kernel 2.5.44 per superare le limitazioni prestazionali dei meccanismi tradizionali di polling come select e poll. Questa tecnologia implementa un approccio scalabile per il monitoraggio simultaneo di migliaia di file descriptor, caratteristica essenziale per lo sviluppo di applicazioni server ad alte prestazioni. L'architettura di epoll si basa su una struttura dati interna al kernel ottimizzata per la gestione efficiente di grandi insiemi di file descriptor. A differenza dei meccanismi precedenti che richiedevano la scansione lineare di tutti i descriptor a ogni chiamata, epoll mantiene uno stato persistente all'interno del kernel e notifica l'applicazione solamente quando si verificano eventi rilevanti sui descriptor monitorati. Il sistema epoll opera attraverso tre funzioni principali che gestiscono il ciclo di vita delle operazioni di monitoraggio. La funzione `epoll_create` inizializza una nuova istanza epoll, creando una struttura dati dedicata nel kernel per la gestione degli eventi. La funzione `epoll_ctl` permette di aggiungere, modificare o rimuovere file descriptor dal set monitorato, mentre `epoll_wait`

attende la notifica degli eventi e restituisce informazioni sui descriptor che hanno attività pendente.

3.8.1 Meccanismi di Funzionamento

Il meccanismo di gestione degli eventi di epoll si basa su un modello edge-triggered o level-triggered, configurabile per ciascun file descriptor monitorato. Nel modo level-triggered, che rappresenta il comportamento predefinito, epoll notifica l'applicazione ogni volta che un file descriptor è pronto per una operazione di input/output, mantenendo la notifica attiva finché la condizione persiste. Il modo edge-triggered fornisce notifiche solamente quando lo stato di un file descriptor cambia da non-pronto a pronto, richiedendo che l'applicazione gestisca completamente tutti i dati disponibili prima della successiva notifica. Questo approccio riduce il numero di chiamate di sistema necessarie ma richiede una programmazione più attenta per evitare la perdita di eventi. La scelta tra questi due modi influenza significativamente il design dell'applicazione e le prestazioni complessive del sistema. Il modo level-triggered semplifica la programmazione mantenendo semantiche simili a select e poll, mentre il modo edge-triggered offre prestazioni superiori in scenari ad alta concorrenza richiedendo però una gestione più sofisticata degli eventi.

3.8.2 Strutture Dati Interne

L'efficienza di epoll deriva dall'utilizzo di strutture dati ottimizzate all'interno del kernel Linux. Il sistema mantiene una red-black tree per l'indicizzazione rapida dei file descriptor registrati, garantendo operazioni di ricerca, inserimento e cancellazione con complessità logaritmica. Questa struttura permette di gestire efficacemente insiemi di descriptor di dimensioni arbitrarie senza degradazione significativa delle prestazioni. La lista degli eventi pronti viene mantenuta separatamente utilizzando una linked list che contiene solamente i descriptor con attività pendente. Questa separazione elimina la necessità di scansionare l'intero set di descriptor monitorati per identificare quelli attivi, riducendo drasticamente il overhead computazionale delle operazioni di polling. Il kernel implementa meccanismi di callback interni che popolano automaticamente la lista degli eventi pronti quando si verificano attività sui file descriptor monitorati. Questo approccio asincrono garantisce che le informazioni sugli eventi siano sempre aggiornate senza richiedere interventi espliciti dell'applicazione.

3.8.3 Limitazioni e Considerazioni

Epoll rappresenta una tecnologia specifica per i sistemi Linux, limitando la portabilità delle applicazioni che ne fanno uso intensivo. Sistemi operativi alternativi forniscono meccanismi equivalenti con interfacce differenti, come kqueue su sistemi BSD e IOCP su Windows, richiedendo implementazioni specifiche per ciascuna piattaforma.

3.9 Redis: Database In-Memory per Applicazioni Ad Alte Prestazioni

Redis rappresenta un sistema di gestione dati in-memory progettato per fornire accesso estremamente veloce a strutture dati complesse. Sviluppato inizialmente da Salvatore Sanfilippo, Redis si distingue per la sua architettura single-threaded che elimina la necessità di meccanismi di locking complessi, garantendo operazioni atomiche e prestazioni prevedibili sotto carico elevato. Il sistema mantiene tutti i dati in memoria principale, utilizzando tecniche di persistenza opzionali per garantire durabilità. Questa scelta architettuale consente latenze nell'ordine dei microsecondi per operazioni standard, caratteristica essenziale per applicazioni che richiedono tempi di risposta rapidi. Redis supporta strutture dati native specializzate inclusi stringhe, hash, liste, set e sorted set, ciascuna ottimizzata per specifici pattern di accesso e casi d'uso.

4 Metodologie

4.1 Panoramica Architettuale

L'architettura del sistema implementa un modello multi-tier che separa chiaramente le responsabilità tra i diversi componenti, ottimizzando il flusso delle richieste e la gestione delle risorse. Il design si basa su un approccio asincrono che massimizza la concorrenza attraverso l'utilizzo di worker thread specializzati e code di richieste per disaccoppiare la ricezione delle richieste dalla loro elaborazione. Il sistema presenta una struttura gerarchica dove i client inviano richieste HTTP che vengono elaborate attraverso diversi livelli di astrazione. Il server principale funge da coordinatore delle operazioni, gestendo l'accettazione delle connessioni e il routing delle richieste verso il sistema di elaborazione interno. Questa separazione permette di ottimizzare ogni componente per la propria funzione specifica, migliorando l'efficienza complessiva del sistema.

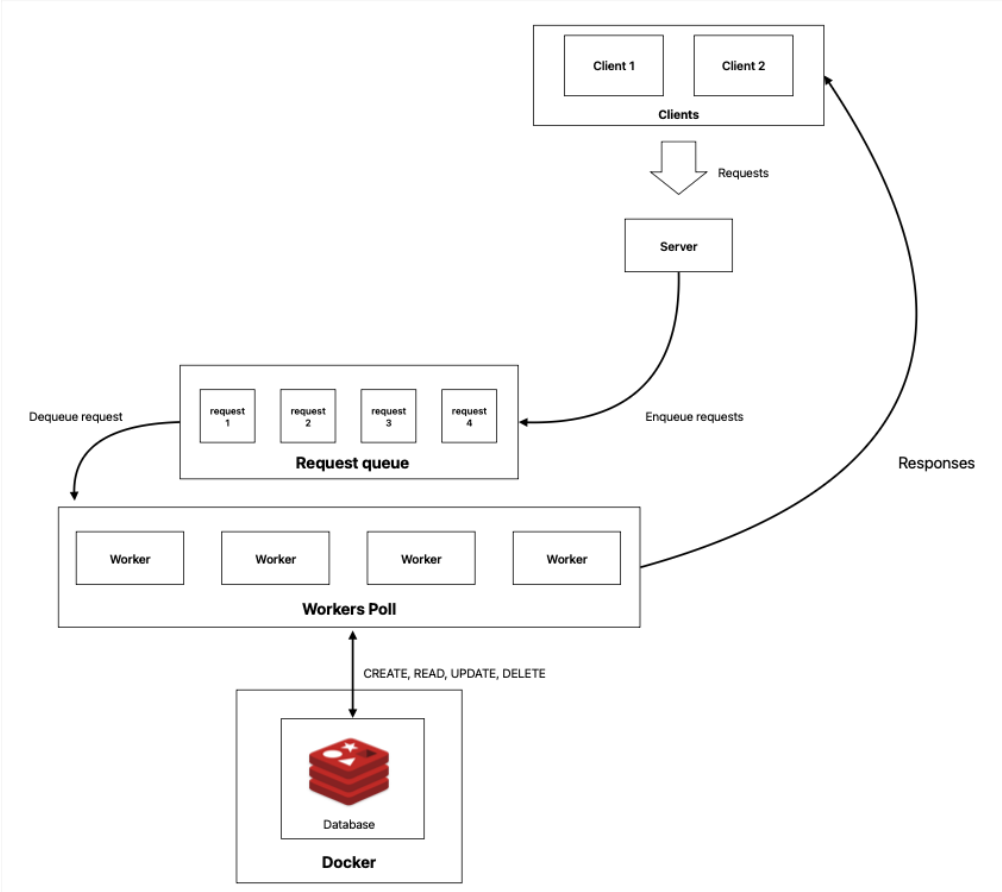

Figura 1: Enter Caption

4.2 Layer Client e Gestione delle Connessioni

Il layer client rappresenta l'interfaccia esterna del sistema, dove applicazioni inviano richieste HTTP attraverso la rete. I client operano in modo completamente indipendente, generando richieste secondo pattern variabili che il sistema deve gestire efficacemente. Il server implementa meccanismi di accettazione delle connessioni ottimizzati per gestire un elevato numero di connessioni simultanee senza compromettere i tempi di risposta. La gestione delle connessioni utilizza tecniche di multiplexing I/O attraverso `epoll` per monitorare simultaneamente migliaia di fie descriptor associati a socket di rete. Questo approccio elimina la necessità di thread dedicati per ogni connessione, riducendo significativamente l'overhead di gestione delle risorse e migliorando la scalabilità del sistema.

4.3 Request Queue e Disaccoppiamento

Il sistema di code delle richieste rappresenta il componente cruciale per il disaccoppiamento tra la ricezione delle richieste e la loro elaborazione. Questa architettura permette al server di accettare richieste a velocità elevata senza essere limitato dalla velocità di elaborazione dei singoli worker.

4.4 Workers Pool e Elaborazione Concorrente e Parallela

Il pool di worker thread costituisce il cuore dell'elaborazione parallela e concorrente del sistema, implementando un modello producer-consumer dove i worker competono per l'acquisizione delle richieste dalla coda condivisa. Ciascun worker opera in modo indipendente, elaborando richieste HTTP complete dalla fase di parsing fino alla generazione della risposta finale. Questa parallelizzazione permette di sfruttare efficacemente le architetture multi-core moderne. I worker implementano la logica di business dell'applicazione REST, gestendo operazioni CRUD complete attraverso l'interfaccia con il database Redis. Ogni worker mantiene connessioni persistenti(attraverso un pool di connessioni pre-inizializzate) verso il container docker contenente Redis per ottimizzare le prestazioni di accesso ai dati, utilizzando connection pooling per bilanciare efficienza e utilizzo delle risorse.

4.5 Integrazione con il Backend Redis

L'architettura integra Redis attraverso containerizzazione Docker, fornendo isolamento completo e gestione semplificata delle dipendenze. Questa scelta architettuale facilita il deployment e la gestione del sistema. La containerizzazione abilita inoltre strategie di scaling orizzontale e vertical scaling del backend di persistenza.

4.6 Struttura Dati Redis : HSET

Redis HSET è una struttura dati che implementa una hash table, permettendo di memorizzare collezioni di coppie campo-valore all'interno di una singola chiave Redis. Questa

struttura è ideale per rappresentare oggetti con proprietà multiple, come ad esempio un profilo utente con nome, email, età e altre informazioni. L'HSET funziona come una mappa o dizionario: ogni hash ha una chiave principale che identifica l'oggetto, e al suo interno può contenere numerosi campi, ognuno con il proprio valore. Ad esempio, una chiave "utente:123" potrebbe contenere i campi "nome", "email" e "età" con i rispettivi valori. L'utilizzo di HSET offre diversi benefici rispetto alla memorizzazione di oggetti serializzati. Permette di aggiornare singoli campi senza dover recuperare e riscrivere l'intero oggetto, riducendo il traffico di rete e migliorando le prestazioni. La struttura è inoltre ottimizzata per l'uso della memoria, utilizzando rappresentazioni compatte per hash di piccole dimensioni.

4.7 Funzione `main()`

```
int main() {
    const int SERVER_PORT = 8080;
    const int MAX_EVENTS = 10;
    struct epoll_event events[MAX_EVENTS];

    redis_pool =  malloc(sizeof(worker_pool_t));
    init_redis_pool(10,redis_pool);

    int server_fd = initialize_server(SERVER_PORT);
    if (server_fd < 0) {
        return EXIT_FAILURE;
    }

    int epoll_fd = init_epoll_istance();
    add_fd_to_epoll_istance(server_fd, epoll_fd, EPOLLIN);

    while (1) {
        int num_events = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);

        if (num_events == -1) {
            if (errno == EINTR) {
                continue;
            }
            perror("epoll_wait failed");
            break;
        }
        if (num_events > 0) {
            if (process_epoll_events(server_fd, epoll_fd, events,
    num_events) < 0) {
                printf("Errore nel processare gli eventi\n");
            }
        }
    }

    cleanup_resources(server_fd, epoll_fd);
    return EXIT_SUCCESS;
}
```

La funzione mostrata nello snippet di codice sopra è lo start point del server. Essa si occupa di inizlizzare le strutture dati necessarie; in particolare si occupa di inizializzare il pool di connessioni al database redis, inizializzare la socket server che si occuperà dell'accettazione di nuovi client. Viene successivamente inizializzata l'istanza di epoll e viene aggiunto ad essa il file descriptor del server in modo tale da monitorarlo per l'accettazione di nuovi client. Dopodichè, in un loop infinito viene chiamata la funzione wait che pone il task in stato di sleep per una quantità di tempo indefinita e fino a quando, uno dei file descriptor monitorati scatena un evento e quindi a quel punto epoll_wait ritornerà il numero di eventi scatenati. Dopodicchè viene invocata, passandogli come argomento il file descriptor relaitivo all'istanza di epoll, la funzione `process_epoll_events()`

4.7.1 Inizializzazione del server

```
int init_server_socket(int port){

    int server_fd;
    struct sockaddr_in serv_addr;

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port);

    if (bind(server_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
    < 0){
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, MAX_CLIENTS) < 0){
        perror("listen failed");
        exit(EXIT_FAILURE);
    }

    set_nonblocking(server_fd);

    return server_fd;

}
```

La funzione ricevuto un numero di porta non fa altro che effettuare la creazione della socket, fare il bind della porta mediante il metodo `bind()` e tramite il metodo `listen()` la mette in ascolto.

```
int initialize_server(int port) {
    printf("Avvio del server...\n");

```

```
    worker_pool = worker_pool_init(10,worker_thread);

    // Inizializza il socket del server
    int server_fd = init_server_socket(port);
    if (server_fd < 0) {
        printf("Errore nell'inizializzazione del socket del server\n");
        return -1;
    }

    // Inizializza epoll
    int epoll_fd = init_epoll_istance();
    if (epoll_fd < 0) {
        printf("Errore nell'inizializzazione di epoll\n");
        close(server_fd);
        return -1;
    }

    // Aggiungi il server socket a epoll
    if (add_fd_to_epoll_istance(server_fd, epoll_fd, EPOLLIN) < 0) {
        printf("Errore nell'aggiunta del server socket a epoll\n");
        close(server_fd);
        close(epoll_fd);
        return -1;
    }

    printf("Server in ascolto sulla porta %d\n", port);

    return server_fd; // Ritorna il file descriptor del server
}
```

Mediante l'uso di questa funzione prendiamo come parametro un numero di porta e restituiamo un intero che rappresenta il file descriptor del server. Viene inizializzato un pool di 10 thread worker, i quali andranno ad eseguire la funzione worker_thread per processare i task. Mediante ini_server_socket ci viene ritornato il file descriptor del server. Successivamente controlliamo se la creazione del socket è fallita e in caso stampa un messaggio di errore e termina la funzione. Fatto ciò viene creata un'istanza di epoll, che ci permette di monitorare eventi su più file descriptor in modo efficiente (**I/O multiplexing**). Verifichiamo che se la creazione di epoll è riuscita, in caso di errore viene chiusa la socket del server, stampato un messaggio di errore e terminata la funzione. Una volta creata la socket del server viene aggiunta all'istanza per monitorare fli eventi di tipo **EPOLLIN** (Dati pronti per essere letti, ovvero nuove connessioni in arrivo). Anche in questo caso viene gestito l'errore nell'aggiunta del socket a epoll.

4.7.2 Inizializzazione istanza epoll

```
int init_epoll_istance(){
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1)
    {
        perror("epoll_create1");
```

```
6           exit(EXIT_FAILURE);
7       }
8
9       return epoll_fd;
10  }
```

Questa funzione viene usata per creare un'istanza di epoll. Essa restistuisce un intero che sarà ul file descriptor dell'istanza epoll. Per fare ciò viene chiamata la chiamata di sistema *epoll_create1()* per creare una nuova istanza. Il parametro 0 indica che non vengono usati flag speciali(si potrebbe passare ad esempio **EPOLL_CLOEXEC per chiudere atuomaticamente l'epoll quanto si esegue un processo figlio**. Come già detto sopra restituisce un file descriptor che rappresenta l'istanza epoll.

4.7.3 Aggiunta di un file descriptor all'istanza di epoll

```
1   int add_fd_to_epoll_istance(int fd_to_monitor, int epoll_istance, int
        event_type){
2
3       struct epoll_event event;
4       event.events = event_type;
5       event.data.fd = fd_to_monitor;
6
7       if (epoll_ctl(epoll_istance, EPOLL_CTL_ADD, fd_to_monitor, &event) ==
         -1)
8       {
9           perror("epoll_ctl: add server");
10          exit(EXIT_FAILURE);
11      }
12
13      return 0;
14  }
```

Questa funzione aggiunge un file descriptor all'istanza epoll per monitorarlo. Prende tre parametri, il file descriptor da monitorare (**fd_monitor**), il file descriptor dell'istanza epoll create precedentemente (**epoll_istance**) e il tipo di evento da monitorare.

Viene dichiarata una struttura epoll_event che contiene le informazioni sull'evento da registrare. Ha due campi principali :

- **events** specifica quali eventi monitorare.
- **data** un'unione che può contenere dati associati all'evento.

Successivamente associamo l'evento che si vuole monitorare e memorizziamo il file descriptor nel campo data della struttura. Questo permette di identificare quale file descriptor ha generato l'evento quando epoll lo segnala.

Non ci resta che aggiungere il file descriptor all'istanza epoll, mediante *epoll_ctl()* che prende come parametri, l'istanza di epoll, l'operazoine di aggiunta (**EPOLL_CTL_ADD**,il file descriptor da aggiungere e un puntatore alla struttura con le informazioni dell'evento.

Se la chiamata a funzione fallisce viene restituito -1, viene stampato un messaggio di errore e terminato il programma.

4.7.4 Processamento eventi epoll

```
int handle_new_connection(int server_fd, int epoll_fd) {
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);

    int new_client_fd = accept(server_fd, (struct sockaddr *)&client_addr
    , &client_len);

    if (new_client_fd < 0) {
        perror("Accept failed");
        return -1;
    }

    printf("Client connesso con successo\n");

    if (set_nonblocking(new_client_fd) < 0) {
        close(new_client_fd);
        return -1;
    }

    if (add_fd_to_epoll_istance(new_client_fd, epoll_fd, EPOLLIN) < 0) {
        close(new_client_fd);
        return -1;
    }

    return 0;
}
```

Questa funzione permette di gestire l'arrivo di una nuova connessione. Mediante la struttura dati **sockaddr_in** andiamo a memorizzare l'indirizzo del client. Altra struttura correlata è **socklen_t** che ci permette di inizializzare la dimensione della struttura sockaddr_in.

Con il metodo *accept()* accettiamo una nuov aconnessione in arrivo sul socket server. I parametri che la funzione prende sono :

- **server_fd** il file descriptor del socker server in ascolto.
- **client_addr** viene rimpieto con l'indirizzo IP e porta del client.
- **client_len** dimensione della struttura client_addr.

In caso la funzione restituisse un valore mnore di 0, allora si è verificato un errore, viene stampato un messaggio di errore e viene ritornato -1.

Nel caso in cui tutto è andato a buon fine viene impostata la socket del client in modalità non bloccante tramite la funzione *set_nonblocking()* e aggiunto il soket all'istanza epoll per il monitoraggio mediante la funzione che abbiamo spiegato antecedentemente *add_to_epoll_istance()*. In caso di errore(anche su set_nonblocking) viene chiuso il socket del client

```
int handle_client_data(int client_fd) {
```

```
    char receiving_buffer[BUFFER_SIZE];
    memset(receiving_buffer, 0, BUFFER_SIZE);

    ssize_t received_data_size = recv(client_fd, receiving_buffer,
    BUFFER_SIZE - 1, 0);

    if (received_data_size > 0) {
        // Dati ricevuti con successo
        receiving_buffer[received_data_size] = '\0';

        http_request_t *request = create_http_request();
        if (!request) {
            printf("Errore nella creazione della richiesta HTTP\n");
            return -1;
        }

        if (parse_http_request(receiving_buffer, request) != 0) {
            printf("Errore nel parsing della richiesta HTTP\n");
            free_http_request(request);
            return -1;
        }

        client_request_node_t* newNode = (client_request_node_t*)malloc(
    sizeof(client_request_node_t));
        if (newNode == NULL) {
            printf("Errore: impossibile allocare memoria per il nuovo
    nodo\n");
            return false;
        }

        newNode->request = *request;
        newNode->next = NULL;
        newNode->client_fd = client_fd;

        enqueue_node(worker_pool->queue, newNode);

        return 0;

    } else if (received_data_size == 0) {
        // Client disconnesso
        printf("Client disconnesso\n");
        close(client_fd);
        return 1; // Indica disconnessione

    } else {
        // Errore nella recv
        if (errno != EAGAIN && errno != EWOULDBLOCK) {
            perror("recv failed");
            close(client_fd);
            return -1;
        }
        return 0; // Non    un errore fatale
    }
```

```
52 }
```

Come prima cosa andiamo a dichiare un buffer per ricevere i dati dal client. Mediante l'uso della funzione *memset()* andiamo ad inizializzare tutto il buffer a zero per evitare dati residui.

Tramite *recv()* andiamo a ricevere i dati dal client. La funzione prende come parametri :

- **client_fd** file descriptor del client.
- **receiving_ buffer** buffer dove memorizzare i dati ricevuti.
- **BUFFER_SIZE-1** dimensione massima da ricevere(lasciamo spazio per inserire il carattere terminatore.
- 0 nessun flag speciale.

Successivamente controlliamo se sono stati ricevuti dati, mediante un if. Infatti se essi sono pervenuti il valore di **recevied_data_size** sarà maggiore di 0. In questo caso andiamo ad aggiungere il carattere terminatore alla fine dei dati ricevuti per garantire che sia una stringa C valida.

Ora non ci resta che gestire il messaggio ricevuto che è una richiesta HTTP. Andremo ad usare una struttura dati, **http_request_t** per memorizzare la richiesta HTTP. Anche in questo caso è stato aggiunto un controllo in caso di errore (viene restituito -1). Una volta creata la struttura andiamo a richiamare la funzione *parse_http_request()* che prende come parametri il contenuto del buffer e la struttura per memorizzare le richieste HTTP parsata. Se c'è un errore viene liberata la memoria associata alla struttura sopracitata. tramite il metodo *free_http_request()*.

Se la richiesta è stata parsata correttamente viene allocata la memoria per un nuovo nodo della coda di lavoro. Una volta che il nodo è stato creato, viene copiata la richiesta HTTP nel nodo(copia per valore). Fatto ciò viene inizializzato il puntatore al prossimo nodo(NULL) e viene memorizzato il file descriptor del client nel nodo.

Ora che abbiamo tutte le informazioni che ci servono possiamo agigungere il nodo alla coda di lavoro per essere processato dai worker mediante la funzione *enqueue_node()*.

Tornando un attimo sopra nella descrizione del codice se **received_buffer** fosse uguale a 0, allora il client ha chiuso la connessione, viene chiusa la connessione dal lato del server e ritorna 1 per indicare la disconnessione(non un errore).

Invece se ci trovassimo nel caso in cui **received_buffer** fosse -1, allora si è verificato un errore. Controlliamo se gli errori che si sono verificati sono **EAGAIN/EWOULDBLOCK** che si verificano quando non ci sono dati disponibili(situazione che si può verificare nel caso in cui la socket non è bloccante). Non si tratta di errori fatali, indicano solo che in quel dato momento non ci sono dati disponibili. In questo caso viene ritornato 0, ma se gli errori sono diversi da quelli sopracitati, allora siamo in presenza di errori più seri, vengono stampati i dettagli e viene chiusa la socket. Infine viene ritornato -1.

```c
int process_epoll_events(int server_fd, int epoll_fd, struct epoll_event
    *events, int num_events) {
    for (int i = 0; i < num_events; i++) {
        int current_fd = events[i].data.fd;

        if (current_fd == server_fd) {
            // Nuova connessione in arrivo
            if (handle_new_connection(server_fd, epoll_fd) < 0) {
                printf("Errore nell'accettare la connessione\n");
                // Continua comunque con gli altri eventi
            }
        } else {
            // Dati pronti per la lettura da un client
            int result = handle_client_data(current_fd);
            if (result < 0) {
                printf("Errore nella gestione dei dati del client\n");
                // Continua comunque con gli altri eventi
            }
        }
    }

    return 0;
}
```

Tramite l'uso di questa funzione andiamo a ciclare attraverso tutti gli eventi pronti restituiti da epoll_wait().Ci estraiamo il file descriptor associato all'evento corrente e controlliamo se corrisponde al socket del server. In questo caso siamo nella situazione in cui abbiamo una nuova connessione in arrivo e allora andiamo a richiamare la funzione *handle_new_connection()*(che abbiamo descritto sopra) passando il file descriptor del server(necessaria per il metodo *accept()*) e il file descriptor dell'istanza di epoll dove andremo ad aggiungere il client. Viene gestito il caso di errore stampando un messaggio a schermo.

Se l'evento non si è verificato sul file descriptor del server, vuol dire che ci sono dati pronti per la lettura da un client. In questo caso viene chiamata la funziona *handle_client_data()* passando come parametro il file descriptor del client che ha inviato i dati. In caso di errore anche qui stampiamo un messaggio a schermo.

4.8 Gestione della coda di richieste

4.8.1 Strutture dati

```c
typedef struct client_request_node_t{
    int client_fd;
    http_request_t request;
    struct client_request_node_t*next;
}client_request_node_t;

typedef struct {
    client_request_node_t* front;
    client_request_node_t* rear;
    int size;     // Numero di elementi nella coda
    int maxSize;

    // Sincronizzazione
    pthread_mutex_t mutex;    // Mutex per accesso esclusivo
    pthread_cond_t notEmpty;  // Condition variable per coda non vuota
    pthread_cond_t notFull;   // Condition variable per coda non piena
    sem_t emptySlots;         // Semaforo per slot vuoti
    sem_t fullSlots;

    int totalProduced;        // Totale elementi prodotti
    int totalConsumed;        // Totale elementi consumati
    bool shutdownFlag;        // Flag per terminazione

} request_queue_t;
```

Le strutture dati mostrate sopra sono centrali nello sviluppo del sistema. Insieme esse rappresentano una linked list necessaria per accodare le richieste HTTP accettate dal server. La struttura `client_request_node_t` rappresenta un singolo elemento della coda implementata come una lined list. Ogni nodo contiene tre componenti essenziali: il descrittore del file del client (client_fd) che identifica univocamente la connessione TCP, la struttura `http_request_t` che incapsula i dati della richiesta HTTP ricevuta, e il puntatore next che mantiene il collegamento al nodo successivo nella sequenza. La struttura `request_queue_t` implementa una coda FIFO completa con meccanismi di sincronizzazione per l'accesso. I puntatori **front** e **rear** gestiscono rispettivamente la testa e la coda della lista, mentre i campi **size** e **maxSize** monitorano lo stato di occupazione della coda e definiscono la capacità massima del buffer. L'implementazione utilizza meccanismi di sincronizzazione per garantire l'accesso thread-safe. Il mutex mutex fornisce l'accesso esclusivo alle operazioni critiche sulla coda. Le condition variables **notEmpty** e **notFull** permettono ai thread di attendere efficientemente quando la coda si trova in stati specifici, evitando il polling attivo. I semafori emptySlots e fullSlots offrono un controllo aggiuntivo sulla disponibilità delle risorse, implementando il pattern produttore-consumatore classico. I contatori totalProduced e totalConsumed forniscono statistiche operative per il monitoraggio delle performance e il debugging del sistema. Il flag booleano shutdownFlag consente una terminazione coordinata di tutti i

thread quando il server deve arrestarsi, garantendo che le operazioni in corso vengano completate correttamente.

4.8.2 Operazioni standard sulla coda

```
bool enqueue_node(request_queue_t* q, client_request_node_t *node) {
    if (q == NULL) {
        printf("Errore: coda non inizializzata\n");
        return false;
    }

    // Attendi uno slot vuoto (semaforo)
    sem_wait(&q->emptySlots);

    // Acquisisci il mutex per accesso esclusivo
    pthread_mutex_lock(&q->mutex);

    // Verifica se la coda    in shutdown
    if (q->shutdownFlag) {
        pthread_mutex_unlock(&q->mutex);
        sem_post(&q->emptySlots); // Rilascia il semaforo
        return false;
    }

    // Attendi finch  la coda non    piena (usando condition variable)
    while (isFullUnsafe(q) && !q->shutdownFlag) {
        pthread_cond_wait(&q->notFull, &q->mutex);
    }

    if (q->shutdownFlag) {
        pthread_mutex_unlock(&q->mutex);
        sem_post(&q->emptySlots);
        return false;
    }

    // Inserisci nella coda
    if (isEmptyUnsafe(q)) {
        q->front = node;
        q->rear = node;
    } else {
        q->rear->next = node;
        q->rear = node;
    }

    q->size++;
    q->totalProduced++;

    // Segnala che la coda non    vuota
    pthread_cond_signal(&q->notEmpty);

    // Rilascia il mutex
    pthread_mutex_unlock(&q->mutex);
```

```
48
49      // Segnala che c'   un elemento pieno
50      sem_post(&q->fullSlots);
51
52      return true;
53  }
```

La funzione `enqueue_node` rappresenta l'implementazione dell'operazione di inserimento di un nodo in coda ad una lista concatenata. Vengono utilizzati inoltre meccanismi di sincronizzazione per fare in modo che le operazioni sulla struttura dati (che è condivisa non) mantengano la consistenza dei dati. Il semaforo empty slot assicura che il thread venga messo in attesa se non sono presenti degli slot vuoti all'interno della cosa. Se la chiamata a `sem_wait` non blocca l'esecuzione allora viene acqisito il lock che permette di accedere ai thread in mutua esclusione alla struttura dati.

```
1   bool dequeue_node(request_queue_t* q, client_request_node_t* node) {
2
3       if (q == NULL || node == NULL) return false;
4
5       // Attendi un elemento pieno (semaforo)
6       sem_wait(&q->fullSlots);
7
8       // Acquisisci il mutex per accesso esclusivo
9       pthread_mutex_lock(&q->mutex);
10
11      // Attendi finch  la coda non    vuota (usando condition variable)
12      while (isEmptyUnsafe(q) && !q->shutdownFlag) {
13          pthread_cond_wait(&q->notEmpty, &q->mutex);
14      }
15
16      // Se la coda    in shutdown e vuota, termina
17      if (q->shutdownFlag && isEmptyUnsafe(q)) {
18          pthread_mutex_unlock(&q->mutex);
19          sem_post(&q->fullSlots); // Rilascia il semaforo
20          return false;
21      }
22
23
24      client_request_node_t* temp = q->front;
25      //*request = temp->request;
26      *node =  *temp;
27
28
29      q->front = q->front->next;
30
31      // Se la coda diventa vuota
32      if (q->front == NULL) {
33          q->rear = NULL;
34      }
35
36      q->size--;
37      q->totalConsumed++;
38
```

```
39
40      free(temp);
41
42      // Segnala che la coda non    piena
43      pthread_cond_signal(&q->notFull);
44
45      // Rilascia il mutex
46      pthread_mutex_unlock(&q->mutex);
47
48      // Segnala che c'   uno slot vuoto
49      sem_post(&q->emptySlots);
50
51      return true;
52  }
```

Il codice mostrato sopra rappresenta l'implementazione classica di rimozione di un elemento dalla coda FIFO. Per garantire la consistenza della struttura dati sono stati implementati i meccanismi classici del produttore-consumatore. Vengono infatti utilizzati dei semafori e dei mutex opportunamente posizionati per garantire che l'operazione di dequeue avvenga in mutua esclusione e solo se sono presenti dei nodi in coda.

4.9 Gestione dei worker thread

4.9.1 Inizializzazione dei workers

```
1  worker_pool_t* worker_pool_init(int num_threads, void* (*process_func)(
       void*)) {
2      if (num_threads <= 0 || !process_func) return NULL;
3
4      worker_pool_t *pool = malloc(sizeof(worker_pool_t));
5      if (!pool) return NULL;
6
7      pool->threads = malloc(sizeof(pthread_t) * num_threads);
8      if (!pool->threads) {
9          free(pool);
10         return NULL;
11     }
12
13     pool->queue = createQueue(20);
14     if (!pool->queue) {
15         free(pool->threads);
16         free(pool);
17         return NULL;
18     }
19
20     pool->num_threads = num_threads;
21     pool->shutdown = false;
22     pool->process_function = process_func;
23
24     // Crea i thread worker
25     for (int i = 0; i < num_threads; i++) {
26         int thread_id = i;
```

```
27          if (pthread_create(&pool->threads[i], NULL, worker_thread, &
    thread_id) != 0) {
28              // Errore nella creazione del thread
29              pool->num_threads = i; // Aggiorna il numero di thread creati
30              worker_pool_destroy(pool);
31              return NULL;
32          }
33      }
34
35      return pool;
36  }
```

La funzione *worker_pool_init* crea e inizializza un pool di thread worker che possono processare task in modo parallelo. Prendi num_threads numero di thread worker da creare e process_func, un puntatore alla funizone che ogni thread eseguirà per processare i task.

Per prima cosa verifichiamo che il numero di thread sia positivo e che la funzione di proccessing sia valida.

Allochiamo memoria per la struttura del pool e controlla se l'allocazione è riuscita.

Fatto ciò viene allocato un array per memorizzare gli identificatori dei thread.

Crea una coda di capacità 20 per i task da processare. In caso di errore, libera la memoria già allocata.

Infine, vengono inizializzati i campi della struttura e creati i threads worker.

4.9.2 Worker Task

```
1  void* worker_thread(void *arg) {
2
3      sleep(3);
4      redisContext *c = get_redis_connection();
5      if (c == NULL) {
6          return;
7      }
8
9      client_request_node_t * incoming_request = malloc(sizeof(
    client_request_node_t));
10
11     while (1){
12         if(dequeue_node(worker_pool->queue, incoming_request)){
13
14             http_response_t *response = process_rest_request(&
    incoming_request->request, c);
15             if (response) {
16
17                 const char *response_string = get_response_string(
    response);
18                 if (response_string) {
19                     printf("\n--- Risposta raw ---\n");
20                     printf("%s\n", response_string);
21                 }
```

```
22
23                 send(incoming_request->client_fd,response_string,strlen(
    response_string),0);
24                 free_http_response(response);
25             }
26         }
27
28         printQueue(worker_pool->queue);
29     }
30     return NULL;
31 }
```

Questo codice implementa la funzione worker che viene esseguita da ogni thread nel thread pool.

La prima cosa che è stata fatta è l'inizializzazione del thread. Attendiamo 3 secondi per garantire una certa sincronizzazione, viene stabilita una connesssione Redis privata per questo thread e se la connessione fallisce il thread termina.

Una volta stabilita la connessione viene allocata la memoria per memorizzare le richieste estratte dalla coda.

Viene estratto un nodo dalla coda del thread pool e l'elemento viene memorizzato in incoming request.

Passo successivo da fare è il processamento della richiesta HTTP usando la connessione Redis, viene convertita la risposta in stringa e stampata per dubug e infine viene inviata la risposta al client tramite socket e liberata la memoria della risposta.

4.10 Gestione operazioni CRUD

4.10.1 Operazione Create

```
1  void crud_create(const http_request_t *request, http_response_t *response
       , redisContext *c){
2
3      if(strncmp(request->path, "/add/book", 10) != 0) {
4          printf("Endpoint non supportato: %s\n", request->path);
5          set_response_status(response, HTTP_NOT_FOUND);
6          set_response_json(response, "{\"error\": \"Endpoint non trovato
       \"}");
7          return;
8      }
9
10     Book new_book;
11
12     if (parse_book_json(request->body, &new_book)) {
13         printf("Parsing completato con successo!\n\n");
14     } else {
15         printf("Errore durante il parsing del JSON\n");
16
17     }
18
19
```

```
20      if(save_book(c, &new_book) == -1){
21          set_response_status(response, HTTP_INTERNAL_SERVER_ERROR);
22          set_response_json(response, "{\"error\": \ Errore interno al
    server...riprova e sarai pi  fortunato... \"}");
23          add_response_header(response, "X-Custom-Header", "MyValue");
24      }
25
26
27
28
29      set_response_status(response, HTTP_OK);
30      set_response_json(response, request->body);
31      add_response_header(response, "X-Custom-Header", "MyValue");
32
33
34  }
```

Mediante questa funzione andiamo ad implementare l'operazione CREATE per gestire libri tramite API REST.

Il nostro scopo è quello di gestire richieste HTTP POST per creare nuovi libri, validando l'endpoint, parsando i dati JSON e salvandoli in Redis.

Prende come parametri :

- request : richiesta HTTP ricevuta dal client.
- response: oggeto risposta da popolare.
- c : connessione Redis per il salvataggio.

Effettuiamo una verifica sul path per eaccettarci che esso sia "/add/book". Se non corrisponde, ritorna errore 404.

Una volta che ci siamo accertati del path, andiamo a convertire il JSON nel body della richiesta in una struttura **Book**.

Avendo la struttura book, viene effettuato un tentativo di salvare il libro in Redis. Se fallisce, viene impostato errore 500. Se non fallisce ritorna status 200 con il JSON originale.

4.10.2 Operazione Read

```
1  void crud_read(const http_request_t *request, http_response_t *response,
       redisContext *c ){
2      if(strncmp(request->path, "/get/books", 11) != 0) {
3          printf("Endpoint non supportato: %s\n", request->path);
4          set_response_status(response, HTTP_NOT_FOUND);
5          set_response_json(response, "{\"error\": \"Endpoint non trovato
       \"}");
6          return;
7      }
8
9      Book new_book;
```

```
10
11      if (parse_book_json(request->body, &new_book)) {
12          printf("Parsing completato con successo!\n\n");
13      } else {
14          printf("Errore durante il parsing del JSON\n");
15
16      }
17
18      Book *loaded_book = load_book(c, new_book.id) ;
19
20      if(loaded_book== NULL){
21          set_response_status(response, HTTP_INTERNAL_SERVER_ERROR);
22          set_response_json(response, "{\"error\": \ Errore interno al
        server...riprova e sarai pi  fortunato... \"}");
23          add_response_header(response, "X-Custom-Header", "MyValue");
24      }
25
26      char resp_body[256];
27      snprintf(resp_body, 256,
28          "{\n"
29          "    \"id_book\": %d,\n"
30          "    \"title\": \"%s\",\n"
31          "    \"author\": \"%s\",\n"
32          "    \"price\": %.2f\n"
33          "}",
34          loaded_book->id, loaded_book->title, loaded_book->author,
        loaded_book->price);
35
36
37      set_response_status(response, HTTP_OK);
38      set_response_json(response, resp_body);
39      add_response_header(response, "X-Custom-Header", "MyValue");
40
41
42  }
```

Questo codice implementa l'operaizone READ di un CRUD per recuperare informazioni sui libri tramite API REST. Quindi andremo a gestire richieste HTTP GET per recuperare un libro specifico da Redis tramite il suo ID.

Anche in questo caso controlliamo da prima il path. Se non corrisponde, ritorniamo errore 404.

In caso di correttezza del path estraiamo l'ID del libro dal body della richiesta mediante *parse_bok_json()*.

Una volta ottenuto l'id possiamo cercare il libro nel database Redis usando l'ID estratto. Se il libro non viene trovato viene impostato l'errore 500.

Se è stato trovato procediamo con formattare i dati del libro in una stringa JSON e inviamo la risposta al client.

4.10.3 Operazione Update

```c
void crud_update(const http_request_t *request, http_response_t *response
    , redisContext *c ){

     if(strncmp(request->path, "/update/book", 13) != 0) {
        printf("Endpoint non supportato: %s\n", request->path);
        set_response_status(response, HTTP_NOT_FOUND);
        set_response_json(response, "{\"error\": \"Endpoint non trovato
    \"}");
        return;
    }

    Book new_book;

    if (parse_book_json(request->body, &new_book)) {
        printf("Parsing completato con successo!\n\n");
    } else {
        printf("Errore durante il parsing del JSON\n");

    }



    if(update_book_price(c,new_book.id, new_book.price) < 0){
        set_response_status(response, HTTP_INTERNAL_SERVER_ERROR);
        set_response_json(response, "{\"error\": \ Errore interno al
    server...riprova e sarai pi  fortunato... \"}");
        add_response_header(response, "X-Custom-Header", "MyValue");
    }

    char resp_body[256];
    snprintf(resp_body, 256,
        "{\n"
        "    \"id_book\": %d,\n"
        "    \"title\": \"%s\",\n"
        "    \"author\": \"%s\",\n"
        "    \"price\": %.2f\n"
        "}",
        new_book.id, new_book.title, new_book.author, new_book.price);


    set_response_status(response, HTTP_OK);
    set_response_json(response, resp_body);
    add_response_header(response, "X-Custom-Header", "MyValue");

}
```

Questo codice implementa l'operazione UPDATE per modificare libri tramite API REST. Andremo a gestire richiete HTTP PUT per aggiornar eil prezzo di un libro esistente in Redis.

Come fatto per le altre due richieste già viste iniziamo con la validazione del path. Una volta fatto la validizzione effettuiamo il parsing del JSON e mettiamo il body in un struttura Book per ottenere ID e nuovo prezzo.

Mediante l'uso di *update_book_price()* aggiorniamo solo l prezzo del libro. Se fallisce, imposta l'errore a 500.

In caso di esito positivo viene formattata la risposta JSON con i dati del libro e viene inviata la risposta al client.

4.10.4 Operazione Delete

```
void crud_delete(const http_request_t *request, http_response_t *response
    , redisContext *c ){
    if(strncmp(request->path, "/delete/book", 13) != 0) {
        printf("Endpoint non supportato: %s\n", request->path);
        set_response_status(response, HTTP_NOT_FOUND);
        set_response_json(response, "{\"error\": \"Endpoint non trovato
    \"}");
        return;
    }

    Book new_book;

    if (parse_book_json(request->body, &new_book)) {
        printf("Parsing completato con successo!\n\n");
    } else {
        printf("Errore durante il parsing del JSON\n");

    }



    if(delete_book(c,new_book.id) < 0){
        set_response_status(response, HTTP_INTERNAL_SERVER_ERROR);
        set_response_json(response, "{\"error\": \ Errore interno al
    server...riprova e sarai pi  fortunato... \"}");
        add_response_header(response, "X-Custom-Header", "MyValue");
    }

    char resp_body[256];
    snprintf(resp_body, 256,
        "{\n"
        "    \"id_book\": %d,\n"
        "    \"title\": \"%s\",\n"
        "    \"author\": \"%s\",\n"
        "    \"price\": %.2f\n"
        "}",
        new_book.id, new_book.title, new_book.author, new_book.price);


    set_response_status(response, HTTP_OK);
    set_response_json(response, resp_body);
    add_response_header(response, "X-Custom-Header", "MyValue");

}
```

Quest'ultima funzione invece implementa l'operazione DELETE per rimuovere libri tramite API REST, andremo a gestire richieste DELETE HTTP per liminare un libro specifico da Redis usando il suo ID.

Come abbiamo già visto viene effettuato parsing e validazione del path. Estraiamo l'ID del libro da eliminare dal body la richiesta.

Infine chiamiamo la funzione per eliminare il libro dal database mediante la funzione delete_book(). Una volta eliminato il ibro viene restituita la risposta al client.

4.11 Parsing richieste HTTP

La funzione ha il compito di prendere una richiesta HTTP "grezza" (cioè una stringa di testo ricevuta da una connessione di rete) e scomporla nei suoi elementi principali, per poi salvarli in una struttura dati che rappresenta la richiesta HTTP.

```
int parse_http_request(const char *raw_request, http_request_t *request)
    {
    if (!raw_request || !request) {
        return -1;
    }

    // Trova il separatore tra headers e body
    const char *header_end = strstr(raw_request, "\r\n\r\n");
    if (!header_end) {
        // Prova con solo \n se non trova \r\n\r\n
        header_end = strstr(raw_request, "\n\n");
        if (!header_end) {
            return -1;
        }
    }

    // Calcola la lunghezza della parte headers
    size_t header_length = header_end - raw_request;

    // Crea una copia solo della parte headers
    char *headers_copy = malloc(header_length + 1);
    if (!headers_copy) {
        return -1;
    }

    strncpy(headers_copy, raw_request, header_length);
    headers_copy[header_length] = '\0';

    // Parsing delle linee degli headers
    char *line = strtok(headers_copy, "\r\n");
    if (!line) {
        // Prova con solo \n
        free(headers_copy);
        headers_copy = malloc(header_length + 1);
        strncpy(headers_copy, raw_request, header_length);
        headers_copy[header_length] = '\0';
        line = strtok(headers_copy, "\n");
```

```
37          if (!line) {
38              free(headers_copy);
39              return -1;
40          }
41      }
42
43      // Parsing della request line
44      if (parse_request_line(line, request) != 0) {
45          free(headers_copy);
46          return -1;
47      }
48
49      // Parsing degli headers
50      while ((line = strtok(NULL, "\r\n")) != NULL || (line = strtok(NULL,
    "\n")) != NULL) {
51          if (strlen(line) == 0) {
52              break; // Fine degli headers
53          }
54          if (parse_header_line(line, request) != 0) {
55              // Non considerare un errore critico un header malformato
56              continue;
57          }
58      }
59
60      free(headers_copy);
61
62      // Parsing del body se presente
63      const char *body_start = strstr(raw_request, "\r\n\r\n");
64      if (body_start) {
65          body_start += 4; // Salta "\r\n\r\n"
66      } else {
67          body_start = strstr(raw_request, "\n\n");
68          if (body_start) {
69              body_start += 2; // Salta "\n\n"
70          }
71      }
72
73      if (body_start) {
74          size_t total_length = strlen(raw_request);
75          size_t body_start_offset = body_start - raw_request;
76          size_t remaining = total_length - body_start_offset;
77
78          if (request->content_length > 0) {
79              // Usa Content-Length se specificato
80              size_t body_size = (remaining < request->content_length) ?
    remaining : request->content_length;
81              request->body = malloc(body_size + 1);
82              if (request->body) {
83                  memcpy(request->body, body_start, body_size);
84                  request->body[body_size] = '\0';
85                  request->body_length = body_size;
86              }
87          } else if (remaining > 0) {
```

```
            // Se non c'è Content-Length ma c'è del contenuto, prendilo tutto
            request->body = malloc(remaining + 1);
            if (request->body) {
                memcpy(request->body, body_start, remaining);
                request->body[remaining] = '\0';
                request->body_length = remaining;
                request->content_length = remaining;
            }
        }
    }

    return 0;
}
```

4.11.1 Compilazione modulare

```
# Makefile per progetto C
# Struttura directory:
# ./ (root) - contiene main.c
# ./src/ - contiene i file sorgente
# ./include/ - contiene i file header
# ./obj/ - contiene i file oggetto (creata automaticamente)
# ./bin/ - contiene l'eseguibile (creata automaticamente)

# Configurazione compilatore e flag
CC = gcc
CFLAGS = -Wall -Wextra -g
INCLUDES = -Iinclude
LIBS = -Llib -lhiredis

# Directory
SRCDIR = src
INCDIR = include
OBJDIR = obj
BINDIR = bin

# Nome dell'eseguibile
TARGET = $(BINDIR)/main

# Trova tutti i file .c in src/
SOURCES = $(wildcard $(SRCDIR)/*.c)
# Crea lista dei file oggetto corrispondenti
OBJECTS = $(SOURCES:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
# Aggiungi main.o per main.c nella root
OBJECTS += $(OBJDIR)/main.o

# Regola principale
all: $(TARGET)

# Regola per creare l'eseguibile
$(TARGET): $(OBJECTS) | $(BINDIR)
```

```
  $(CC) $(OBJECTS) -o $@ $(LIBS)
  @echo "Compilazione completata: $(TARGET)"

# Regola per compilare main.c dalla root
$(OBJDIR)/main.o: main.c | $(OBJDIR)
  $(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

# Regola per compilare i file .c da src/
$(OBJDIR)/%.o: $(SRCDIR)/%.c | $(OBJDIR)
  $(CC) $(CFLAGS) $(INCLUDES) -c $< -o $@

# Crea la directory obj se non esiste
$(OBJDIR):
  mkdir -p $(OBJDIR)

# Crea la directory bin se non esiste
$(BINDIR):
  mkdir -p $(BINDIR)

# Pulizia dei file generati
clean:
  rm -rf $(OBJDIR) $(BINDIR)
  @echo "File temporanei rimossi"

# Pulizia solo dei file oggetto
clean-obj:
  rm -rf $(OBJDIR)
  @echo "File oggetto rimossi"

# Ricompilazione completa
rebuild: clean all

# Esecuzione del programma
run: $(TARGET)
  ./$(TARGET)

# Debug con gdb
debug: $(TARGET)
  gdb ./$(TARGET)

# Mostra informazioni sul progetto
info:
  @echo "Progetto C"
  @echo "=========="
  @echo "Compilatore: $(CC)"
  @echo "Flag: $(CFLAGS)"
  @echo "Include: $(INCLUDES)"
  @echo "Sorgenti: $(SOURCES) main.c"
  @echo "Oggetti: $(OBJECTS)"
  @echo "Target: $(TARGET)"

# Installa dipendenze automatiche
$(OBJDIR)/%.d: $(SRCDIR)/%.c | $(OBJDIR)
```

```
89    @$(CC) -MM $(CFLAGS) $(INCLUDES) $< | sed 's|$*.o|$(OBJDIR)/$*.o $(
        OBJDIR)/$*.d|g' > $@
90
91  $(OBJDIR)/main.d: main.c | $(OBJDIR)
92    @$(CC) -MM $(CFLAGS) $(INCLUDES) $< | sed 's|main.o|$(OBJDIR)/main.o $(
        OBJDIR)/main.d|g' > $@
93
94  # Include le dipendenze solo se non stiamo facendo clean
95  ifneq ($(MAKECMDGOALS),clean)
96  ifneq ($(MAKECMDGOALS),clean-obj)
97  -include $(OBJECTS:.o=.d)
98  endif
99  endif
100
101 # Dichiara target che non corrispondono a file
102 .PHONY: all clean clean-obj rebuild run debug info
```

4.12 Gestione della Compilazione con Makefile

La compilazione di un progetto C di media complessità richiede un sistema di build efficiente e organizzato. Il Makefile presentato implementa una soluzione robusta per gestire la compilazione automatica di un progetto strutturato in directory multiple, con gestione delle dipendenze e funzionalità avanzate di debug e manutenzione.

4.12.1 Architettura del Progetto

Il progetto adotta una struttura gerarchica ben definita che separa logicamente i diversi componenti del codice sorgente. La directory radice contiene il file principale `main.c`, mentre le directory specializzate ospitano rispettivamente i file sorgente (`src/`), gli header (`include/`), i file oggetto compilati (`obj/`) e l'eseguibile finale (`bin/`). Questa organizzazione facilita la manutenzione del codice, migliora la leggibilità del progetto e permette una gestione più efficiente delle dipendenze.

Il sistema di build è configurato per utilizzare il compilatore GCC con flag di ottimizzazione e debug appropriati. I flag `-Wall` e `-Wextra` garantiscono un alto livello di controllo sulla qualità del codice, evidenziando potenziali problemi durante la compilazione, mentre il flag `-g` include le informazioni di debug necessarie per l'utilizzo di strumenti come GDB.

4.12.2 Meccanismo di Compilazione

Il processo di compilazione segue un approccio incrementale basato su pattern matching e regole automatiche. Il Makefile identifica automaticamente tutti i file sorgente presenti nella directory `src/` utilizzando la funzione `wildcard`, generando dinamicamente la lista dei corrispondenti file oggetto. Questa strategia permette di aggiungere nuovi file sorgente al progetto senza dover modificare manualmente il Makefile.

Il linking finale combina tutti i file oggetto generati, includendo il main compilato separatamente dalla directory radice, e produce l'eseguibile nella directory `bin/`. Il

sistema gestisce automaticamente la creazione delle directory necessarie qualora non esistano, utilizzando il meccanismo delle dipendenze ordinate di Make.

4.12.3 Funzionalità Avanzate

Il Makefile offre diverse funzionalità che semplificano il workflow di sviluppo. Il target `run` permite l'esecuzione diretta del programma dopo la compilazione, mentre `debug` avvia automaticamente GDB sull'eseguibile. Le operazioni di pulizia sono disponibili in due modalità: `clean` rimuove completamente i file generati, mentre `clean-obj` elimina solo i file oggetto mantenendo l'eseguibile.

La funzionalità `info` fornisce una panoramica completa della configurazione del progetto, mostrando le impostazioni del compilatore, le directory utilizzate e i file coinvolti nel processo di build. Il target `rebuild` combina la pulizia completa con una ricompilazione totale, garantendo un build pulito del progetto.

4.12.4 Integrazione con Librerie Esterne

Il Makefile è configurato per supportare l'integrazione con librerie esterne, come dimostrato dall'inclusione della libreria Redis (hiredis) nelle opzioni di linking. Questa configurazione può essere facilmente estesa per includere ulteriori librerie, modificando le variabili `LIBS` e, se necessario, aggiungendo percorsi di ricerca aggiuntivi per i file header.

La struttura modulare del Makefile permette una personalizzazione semplice e intuitiva: le configurazioni principali sono centralizzate nelle variabili iniziali, mentre la logica di compilazione rimane invariata indipendentemente dalle specificità del progetto.

5 Risultati

Una volta realizzata l'implementazione di cui sono state discusse le parti essenziali(l'implementazione completa è possibile trovarla su GitHub) sono stati effettuati dei test che riguardano le operazioni CRUD implementate utilizzando come client Postman per cosrtuire le richieste HTTP.

5.1 Test delle operazioni implementate

5.1.1 Avvio del server

```
ubuntu:final_project$ ./bin/main
Pool Redis inizializzato con 10 connessioni
Avvio del server...
Server in ascolto sulla porta 8080...
Server pronto per accettare connessioni...
```

5.1.2 Create Book

Attraverso Postman è stata effettuata una richiesta POST a `localhost:8080/add/book` inserendo nel body la seguente stringa JSON:

```
{
    "id_book": 1,
    "title" : "Il signore degli analli",
    "author" : "autore1",
    "price" : 120.0
}
```

La risposta visualizzata su Postman e la seguente:

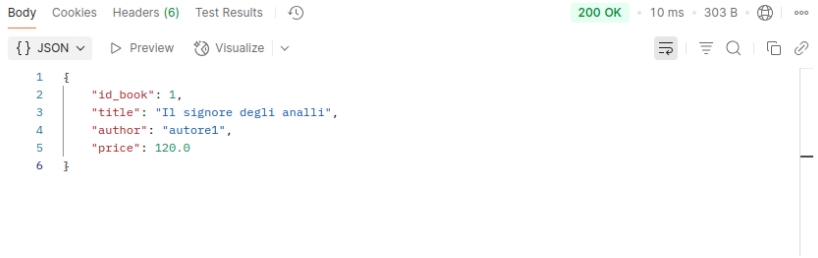

Figura 2: Create Server response

Per come è stato implementato l'endpoint, se la richiesta di creazione di un libro ha successo viene inclusa nella risposta con codice HTTP 200 il json contenete le infromazioni inserite nel Database.

5.1.3 Update Book

Attraverso Postman è stata effettuata una richiesta PUT a `localhost:8080/update/book` inserendo nel body la seguente stringa JSON indicando l'id del libro di cui aggiornare il prezzo ed il nuovo prezzo:

```
{
    "id_book" : 1,
    "price" : 30.0
}
```

Di seguito è mostrata la risposta ottenuta attraverso Postman che come si può notare ha il codice di stato 200. Per come è stato implementato l'endpoint REST se la richiesta di aggiornamento ha successo viene inviato dal server, in risposta, una stringa JSON contenente il nuovo prezzo e l'id del libro si cui abbiamo aggironato il prezzo.

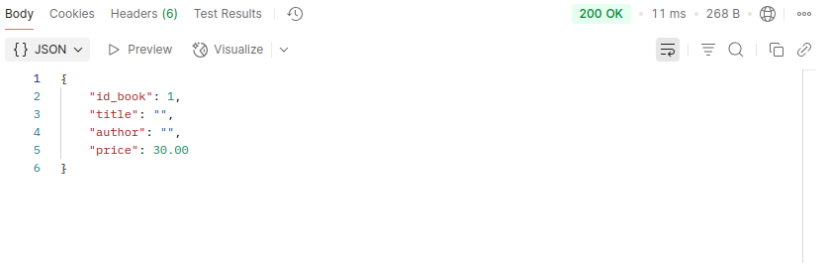

Figura 3: Update Server response

5.1.4 Read book

Attraverso Postman è stata effettuata una richista GET a `localhost:8080/get/books` inserendo nel body la seguente stringa JSON indicando l'id del libro di cui si vogliono ottenere le informazioni :

```
{
    "id_book" : 1
}
```

La risposta mostrata attraverso Postman indica che la richiesta ha avuto successo, questo si denota dal codice di stato HTTP che è 200. Nella risposta è inclusa la stringa JSON contenente la descrizione del libro.

Figura 4: Read Server response

5.1.5 Delete book

Attraverso Postman è stata effettuata una richista DELETE a `localhost:8080/delete/book` inserendo nel body la seguente stringa JSON indicando l'id del libro che si intende eliminare :

```
{
    "id_book": 1
}
```

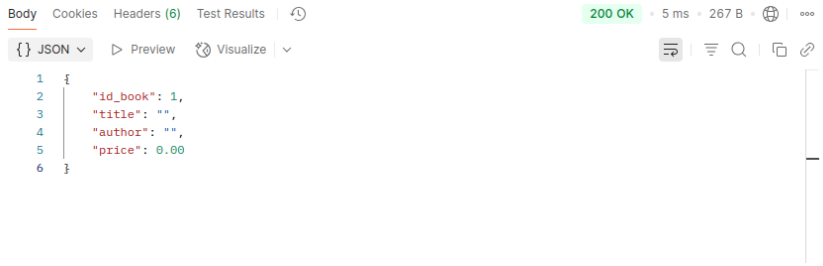

Figura 5: Delete Server response

5.1.6 Test di Carico

Di seguito sono mostrati i risultati del test di carico effettuato sul server sfuttando uno script in python. Lo script testa il server con 5 livelli di carico (1-5) e valuta per ogni operazione il tempo di risposta medio, la percentuale di successo, il throughput, la variabilità dei tempi di risposta rispetto alla media, la distribuzione dei tempi di risposta(sul carico massimo) e dei perncentili. Questi strumenti statistici ci hanno permesso di valutare attraverso dei dati le performace offerte da questa implementazione.

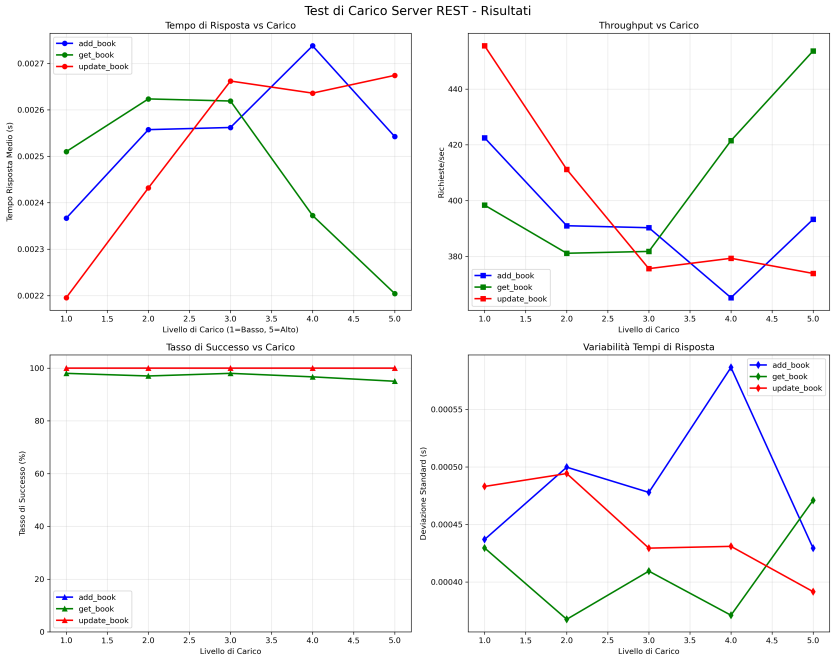

Figura 6: Risultati Test di carico

Quello che si nota in figura 6 dal grafico in alto a sinistra è un andamento crescente dei tempi di rispota all'aumentare del carico per le operazioni di scrittura e aggiornaemento. Mentre per quanto rigurda l'operazione di lettura emerge una tendenza opposta. Da

questo ne deduciamo che le operazioni di scrittura ed aggiornamento sono più sensibili all'aumento del carico sul server. Dal grafico in alto a sinistra nella figura 6 è possibile confermare quanto detto fino ad ora, infatti per le operazioni di lettura si ha un ottima risposta al carico di richieste mantenendo un throughput ottimo.

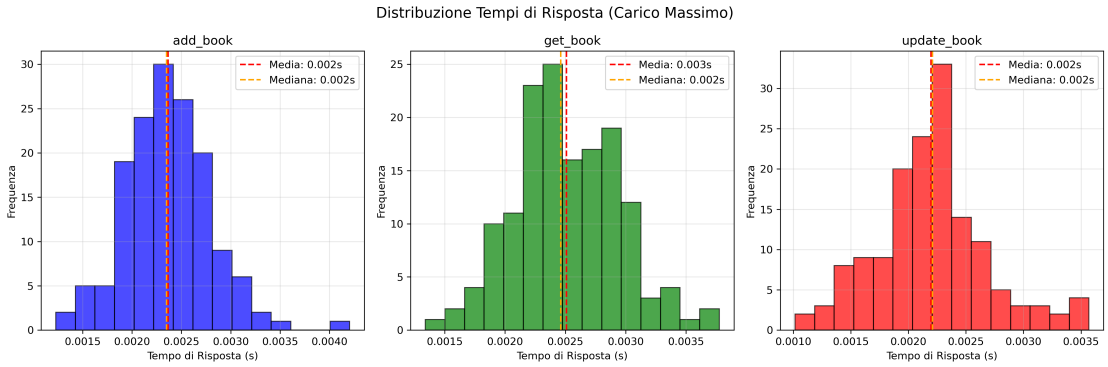

Figura 7: Risultati Test di carico

In figura 7 si nota che la maggior parte delle richieste viene servita in un tempo decisamente ottimo considerando il fatto che si sta parlando di un implementazione a scopo didattico.

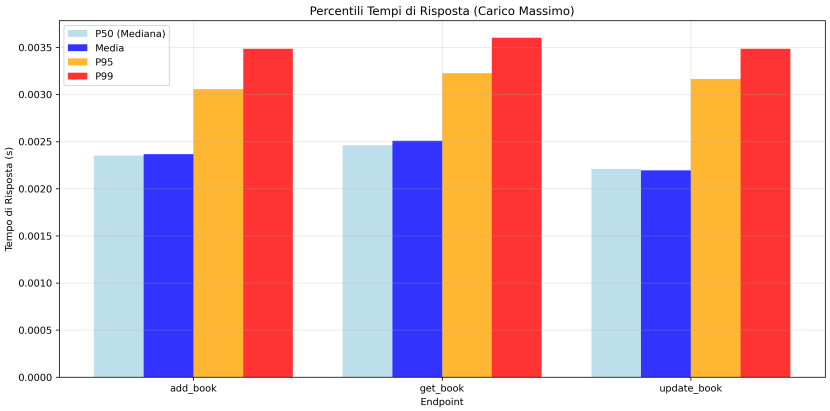

Figura 8: Risultati Test di carico

6 Conclusioni

L'architettura progettata, presenta un elevata scalabilità grazie al fatto che è possibile gestire la quantità di risorse dedicate al processamento delle richieste. In delle implementazioni future infatti porebbe essere il caso di implementare algoritmi capaci di stimare il carico effettivo di ogni worker e in maniera adattiva al carico del server rilasciare o

istanziare nuove risorse. Inoltre la presenza della coda permette al thread principale del server di svincolarsi una volta accettata la richiesta. Altri sviluppi futuri comprendono una gestione degli errori più robusta, permettendo al server di continuare a funzionare anche in casistiche particolari. Tutto questo potrebbe esse supportato anche da un sistema di logging, molto utile per effettuare debuging e fare delle diagniosi sulla base degli eventi registrati dal sistema di logging. In futuro si potrebbe inoltre attenzionare di più gli aspetti legati alla sicurezza. Infatti, di base non è "security by design" pertanto è sicuramente vulnerabili ad attacchi di tipo buffer-overflow, denial of service. Inoltre, questo a prescindere dal server, sarebbe stato possibile proteggere gli endpoint sfuttando strategie come JWT token o altre metodologie che permettono l'accesso agli endpoint REST solo es esclusivamente agli autenti autorizzati. Un'altro aspetto da attenzionare in futuro è la realizzazione di strutture dati più efficienti da memorizzare sul Database. Nel nostro specifico caso applicativo l'attenzione è stata posta sulle tecnologie messe a disposizione dai sistemi Unix-Like per affrontare le problematiche del I/O Multiplexing, la comunicazione di rete attraverso le socket, il multi-threading e la gestione della concorrenza; queste sono le motivazioni per le quali alcuni aspetti, durante la progettazione, sono stati semplificati per concentrarsi sugli obiettivi formativi del progetto.