

Realizzazione di un server concorrente in C con Redis

Presentazione di :
Passari Ludovico, Musarra Tubbi Mattia

Università degli Studi di Messina

6 luglio 2025



1 Introduzione

2 Descrizione del problema

3 Progettazione dell'architettura

4 Risultati

5 Conclusioni





Figura 1: Enter Caption

La domanda

La domanda che ci siamo posti è stata :

Come fanno i WebServer moderni a gestire in maniera efficace grandi volumi di utenti ?



Tecnologie Scelte

- Linguaggio C
- Sockets con `epoll()`
- Redis per la persistenza veloce
- Thread pool per la concorrenza

Obiettivi

- Alta efficienza e reattività
- Architettura modulare
- Bassa latenza
- Facilità di scalabilità



Obiettivo

L'obiettivo che ci siamo prefissati di raggiungere è la realizzazione di un server capace di rispondere a delle richieste effettuate da diversi client, utilizzando REST a delle richieste relative alla gestione di una libreria.

Definizione :

Un **sistema distribuito** è un insieme composto da più calcolatori o processori interconnessi che comunicano tra loro, solitamente sfruttando i servizi messi a disposizione da una rete.



Descrizione :

In un'architettura client-server, esiste un host sempre attivo, chiamato **server**, che fornisce servizi a richieste provenienti da molti altri host, chiamati **client**. Il server gestisce una risorsa e fornisce un servizio manipolando tale risorsa per i suoi client. I client, a loro volta, richiedono i servizi.

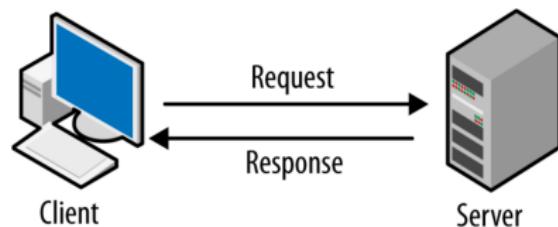


Figura 2: Enter Caption



Ruoli e Comunicazione:

- **Server:** È un host "always-on" con un indirizzo IP fisso(solitamente) e ben noto. Quando un server riceve una richiesta da un client, risponde inviando l'oggetto richiesto.
- **Client :** È un host che non ha un ruolo subordinato al server, effettua semplicemente delle richieste alle quali il server risponderà.
- **Comunicazione tramite scambio di Messaggi:** La comunicazione tra client e server avviene tramite lo scambio di messaggi. Il client invia un messaggio contenente la richiesta e il server esegue il lavoro e risponde.



Diverse tipologie di server :

- **Server Iterativi (o Bloccanti)** : è un server single-thread che gestisce una richiesta per volta. Utilizza socket in modalità bloccante: quando una chiamata di I/O (come `read()` o `accept()`) è eseguita, il server rimane in attesa finché non è completata. Questo modello è semplice ma inefficiente in scenari con molti client simultanei.
- **Server Non Bloccanti (con Polling)** : i socket sono configurati in modo che le operazioni di I/O (lettura/scrittura) ritornino immediatamente. Se i dati non sono pronti, la chiamata fallisce temporaneamente, permettendo al server di continuare a processare altre operazioni. Questo approccio può richiedere polling esplicito, cioè controllare ripetutamente se i socket sono pronti.
- **Server Concorrenti** : è progettato per gestire più richieste contemporaneamente. La concorrenza può essere implementata tramite la creazione di nuovi processi (es. `fork()`), thread (es. `pthread`), oppure utilizzando un event loop asincrono. Questo approccio migliora la scalabilità ma richiede una gestione attenta delle risorse e della sincronizzazione.
- **Server basati su I/O Multiplexing (Select, Epoll)**: L'I/O multiplexing è una tecnica che consente a un singolo thread di monitorare contemporaneamente più "file descriptor".

Un **server iterativo** è un server single thread che processa una richiesta alla volta.

Perchè questo comportamento ?

Questo comportamento è dovuto all'utilizzo di chiamate bloccanti (blocking calls) di rete. Funzioni come accept() (che attende una connessione) e read() (che attende dati) sono bloccanti per impostazione predefinita. Ciò significa che se la risorsa non è pronta, il sistema operativo sospende il thread in esecuzione e passa il controllo della CPU ad altri processi, un meccanismo noto come context switch.

Svantaggi

Scalabilità è limitata perché ogni client deve attendere il completamento delle operazioni del client precedente. Se un client è in attesa di dati, il server non può gestire un altro client finché il primo non ha terminato. Questo porta a una risposta lenta ed è poco applicabile in contesti di produzione.

Vantaggi

Il codice è più semplice e lineare e debugging più semplice.

Utilizzano socket settati in modalità **non bloccante**. Le funzioni di lettura e scrittura ritornano immediatamente.

Perchè si fa questo ?

Per gestire la ricezione dei dati, si adotta una strategia di polling, ovvero un ciclo infinito **while (1)** che invoca continuamente le funzioni di lettura, anche se non ci sono dati pronti. Si fa questo per controllare ad ogni ciclo se ci sono dati pronti da leggere oppure se è possibile scrivere su un socket.

Svantaggi

I server con attesa attiva (busy waiting) comportano un carico molto maggiore sulla CPU. In quanto sprecano cicli di CPU per svolgere computazione "inutile". Infatti ad ogni ciclo è come se si interrogasse il socket per verificarne lo stato.

Vantaggi

Usare questo approccio rende il server più reattivo in quanto non si hanno, ad esempio, gli overhead dovuti ai context switch.



A differenza dei server iterativi, un **server concorrente** è progettato per gestire più richieste simultaneamente.

Comportamento

Al momento di ricevere una richiesta, un server concorrente crea un task figlio (un processo o un thread) incaricato di fornire il servizio richiesto e si rimette immediatamente in attesa di nuove richieste. Questo permette, su sistemi multitasking o multicore, di soddisfare più richieste contemporaneamente.

Svantaggi

Bisogna tenere conto dell'overhead costituito dalla gestione dei thred oppure(a maggior ragione) dei processi figli.

I concetti di **parallelismo** e di **concorrenza** possono portare a delle ambiguità di significato.

Precisazione

Mentre la concorrenza è il concetto generale di un sistema che esegue attività multiple interallacciate (anche su un singolo core), il parallelismo implica l'esecuzione simultanea su più core. Un server multithreaded o multiprocesso può sfruttare il parallelismo su macchine multicore.

Server basati su I/O Multiplexing : `select()`

```
1 int select(int nfds, fd_set *restrict readfds, fd_set *restrict writefds,  
           fd_set *restrict errorfds, struct timeval *restrict timeout);
```

Funzionamento

La funzione di sistema `select()` agisce su insiemi di file descriptor (readfds, writefds, exceptfds) e un parametro `nfds` (il valore massimo del descrittore di file più 1). Permette di spostare il monitoraggio dei file descriptor dallo userspace al kernel space.

Complessità computazionale

Ha una complessità computazionale di $O(N)$, dove N è il numero totale di file descriptor monitorati. Questo la rende meno efficiente con un gran numero di connessioni. Inoltre permette di monitorare un numero limitato di file descriptor(1024).

È bloccante o non bloccante ?

La sua natura bloccante dipende da `*timeout`: se impostato a zero è non bloccante, se nullo è bloccante (attende indefinitamente), altrimenti si blocca per il tempo specificato.



Descrizione

`epoll()` è una funzione di sistema di Linux progettata per superare i limiti di `select()` e per monitorare in modo efficiente **eventi** di I/O su un gran numero di **file descriptor**. Rappresenta la tecnologia più avanzata disponibile su sistemi Linux per gestire connessioni multiple con un singolo thread.

Complessità computazionale

`epoll()` risolve i problemi di complessità si `select` introducendo una complessità computazionale di $O(1)$ per il recupero dei file descriptor pronti. Questo è possibile perché `epoll()` non itera su tutti i file descriptor ogni volta, ma **conosce già a priori quali sono pronti**. Ciò si traduce in una scalabilità notevolmente superiore, permettendo di gestire teoricamente fino a un milione di connessioni.



epoll() : Principio di funzionamento

`epoll()` opera a basso livello attraverso strutture dati e meccanismi ottimizzati nel kernel. Le strutture dati chiave sono due:

- **Interest List (albero)**: Un albero efficiente che tiene traccia di tutti i FD registrati con `epoll_ctl()`. Questa struttura permette ricerche efficienti per aggiungere, modificare o rimuovere FD.
- **Ready List (linked-list)**: Una lista che mantiene sempre disponibili i FD pronti per operazioni di I/O (lettura, scrittura, eccezioni) e che viene aggiornata dal kernel tramite un meccanismo di interrupt quando si verificano eventi. La Ready List è accessibile in $O(1)$ durante la chiamata a `epoll_wait()`.

epoll() : API

L'utilizzo di `epoll()` si basa su tre chiamate di sistema fondamentali:

- `epoll_create1(flags)`: Questa funzione crea una nuova istanza epoll e restituisce un file descriptor che la identifica. Il flag `EPOLL_CLOEXEC` può essere usato per chiudere automaticamente il FD durante una chiamata `exec()`.
- `epoll_ctl(epoll_fd, op, fd, event)`:
 - `epoll_fd`: Il descrittore restituito da `epoll_create1()`.
 - `op`: L'operazione da eseguire (`EPOLL_CTL_ADD` per aggiungere, `EPOLL_CTL_MOD` per modificare, `EPOLL_CTL_DEL` per rimuovere).
 - `fd`: Il file descriptor da gestire.
 - `event`: Una struttura `epoll_event` che specifica gli eventi da monitorare (`EPOLLIN` per lettura, `EPOLLOUT` per scrittura, `EPOLLET` per Edge-Triggered).
- `epoll_wait(epoll_fd, events, maxevents, timeout)` :
 - `epoll_fd`: Il descrittore dell'istanza epoll.
 - `events`: Un array di strutture `epoll_event` dove verranno scritti gli eventi rilevati.
 - `maxevents`: La capacità di questo array.
 - `timeout`: tempo massimo di attesa in millisecondi (-1 per attendere indefinitamente).

Il workflow di epoll si articola in tre fasi:

- **Registrazione dei FD:** Quando un FD viene aggiunto con `epoll_ctl()`, il kernel lo inserisce nell'Interest List e vi collega una callback per tracciare gli eventi.
- **Attesa degli Eventi:** Quando `epoll_wait()` viene chiamato, il processo viene sospeso finché non ci sono eventi o il timeout scade. Il kernel verifica la Ready List e copia gli eventi pronti nello spazio utente.
- **Notifica degli Eventi :** Quando un evento si verifica su un FD, la callback associata viene attivata, il FD viene aggiunto alla Ready List, e il processo in `epoll_wait()` viene risvegliato.

epoll() : Modalità di funzionamento

epoll() supporta due modalità di notifica degli eventi:

- **Level-Triggered (LT)**: Questa è la modalità di default. Notifica l'applicazione finché il file descriptor è pronto per l'operazione (ad esempio, ci sono dati non letti nel buffer). L'applicazione continua a ricevere notifiche finché non ha processato tutti i dati o l'evento non è più presente.
- **Edge-Triggered (ET)**: Notifica l'applicazione solo quando c'è un cambiamento di stato (e.g., da "nessun dato" a "dati disponibili"). Questa modalità è più complessa da gestire perché l'applicazione deve essere sicura di leggere tutti i dati disponibili dopo ogni notifica, altrimenti potrebbe non ricevere ulteriori notifiche per eventi successivi sullo stesso FD finché un nuovo cambiamento di stato non si verifica.

Cosa è REST ?

REST (Representational State Transfer) è uno stile architettonico per la progettazione di servizi web, non un protocollo o uno standard rigido.

Perché è importante? REST sfrutta l'infrastruttura esistente del Web (HTTP, URI, formati standard) per creare servizi che sono naturalmente scalabili e facili da integrare.

Il concetto di risorsa

Una **risorsa** è qualsiasi informazione che può essere identificata con un nome.

Esempi di Risorse:

- Un utente: `/users/123`
- Una collezione di prodotti: `/products`

Caratteristiche delle Risorse:

- **Identificate univocamente** da URI (Uniform Resource Identifier)
- **Rappresentate** attraverso diversi formati (JSON, XML, HTML)

Analogia: Si può pensare ad una risorsa come a un documento in un archivio. L'URI è l'indirizzo del documento, mentre la rappresentazione è il modo in cui visualizzi il contenuto (fotocopia, scan digitale, riassunto).

La domanda sorge spontanea...

Come vengono scambiate tra client e server le rappresentazioni delle risorse ?

Descrizione

HTTP (HyperText Transfer Protocol) è un protocollo di comunicazione utilizzato per trasferire informazioni su Internet, in particolare tra un client (di solito un browser web) e un server web.

- È un protocollo basato su messaggi testuali human readable, eppure è abbastanza potente da supportare l'intera infrastruttura digitale moderna.

- **1989-1991:** Tim Berners-Lee al CERN inventa il World Wide Web insieme alla prima versione del protocollo. Era incredibilmente semplice: supportava solo il comando GET per recuperare documenti HTML.
- **1996 - HTTP/1.0 - La Prima Standardizzazione :** Introduce header, metodi POST e HEAD, codici di stato. Il Web inizia a diventare più interattivo, non più solo lettura di documenti statici.
- **1997 - HTTP/1.1 :** Connessioni persistenti chunked encoding, cache avanzata. Questa versione ha dominato il Web per quasi 20 anni, dimostrando la solidità del design originale.

HTTP : Struttura di una richiesta

Ogni richiesta HTTP è composta dalle seguenti parti :

- **Request Line (La Prima Riga)** : contiene tre informazioni come il metodo (quale operazione), l'URI (su quale risorsa lo vuoi fare), e la versione del protocollo (standard da rispettare). È come dire "Voglio leggere il documento numero 123 nella cartella prodotti, e diciamo che lo scambio di messaggi avviene usando le regole HTTP versione 1.1".

```
1 GET /products/123 HTTP/1.1
```

- **Headers** : sono campi di testo che fanno parte di ogni richiesta o risposta HTTP. Forniscono informazioni aggiuntive sul messaggio stesso, come tipo di contenuto, lunghezza, lingua, dati di autenticazione, cache, e altro ancora.

```
1 Host: api.negozio.com
2 User-Agent: Mozilla/5.0 (Chrome/91.0)
3 Accept: application/json
4 Authorization: Bearer eyJ0eXAi...
```

- **Body (Il Contenuto, Quando Presente)**: contiene i dati veri e propri che vuoi inviare. Non tutte le richieste hanno un body - GET normalmente non ne ha bisogno, mentre POST e PUT lo utilizzano per inviare informazioni al server.

```
1 {
2   "nome": "Smartphone XYZ",
3   "prezzo": 299.99,
4   "categoria": "elettronica"
5 }
```

HTTP : Response Message

Dopo ogni richiesta, il server invia una risposta strutturata che segue un formato preciso:

- **Status Line** : La prima riga comunica immediatamente l'esito dell'operazione: versione del protocollo, codice numerico di stato, e descrizione testuale. È come l'oggetto di un'email che ti dice subito se contiene buone o cattive notizie.

```
1 HTTP/1.1 200 OK
```

- **Response Headers** : Gli header della risposta forniscono metadati sulla risposta: che tipo di contenuto stai ricevendo, quanto è grande, se puoi memorizzarlo nella cache, informazioni di sicurezza.

```
1 Content-Type: application/json
2 Content-Length: 1024
3 Cache-Control: max-age=3600
4 Set-Cookie: sessionId=abc123; HttpOnly
```

- **Response Body (Il Contenuto Richiesto)**: Il body contiene i dati effettivi richiesti dal client. Può essere HTML per una pagina web, JSON per un'API, un'immagine, un video, o qualsiasi altro tipo di contenuto digitale.

```
1 {
2   "id": 123,
3   "nome": "Smartphone XYZ",
4   "prezzo": 299.99,
5   "disponibile": true
6 }
```

I **Codici di stato** in HTTP sono elementi fondamentali del protocollo, utilizzati per indicare il risultato di un tentativo del server di comprendere e soddisfare una richiesta. Facendo riferimenti ad RFC 2616.

La prima cifra del codice di stato definisce la classe della risposta. Esistono 5 classi principali:

- **1xx: Informational (Informative)** : indicano che il server ha ricevuto la richiesta e sta continuando il processo.
- **2xx: Success (Successo)** : Indica che l'azione è stata ricevuta, compresa e accettata con successo.
 - **200 OK**: La richiesta è riuscita.
 - **201 Created** : La richiesta è stata soddisfatta e ha portato alla creazione di una nuova risorsa.
 - **202 Accepted**: La richiesta è stata accettata per l'elaborazione, ma l'elaborazione non è stata completata.
- **3xx: Redirection (Reindirizzamento)**: Indica che è necessario intraprendere ulteriori azioni da parte del client per soddisfare la richiesta.
- **4xx: Client Error (Errore del Client)** : Indica casi in cui il client sembra aver commesso un errore.
- **5xx: Server Error (Errore del Server)** : Indica casi in cui il server è consapevole di aver commesso un errore o è incapace di eseguire la richiesta.

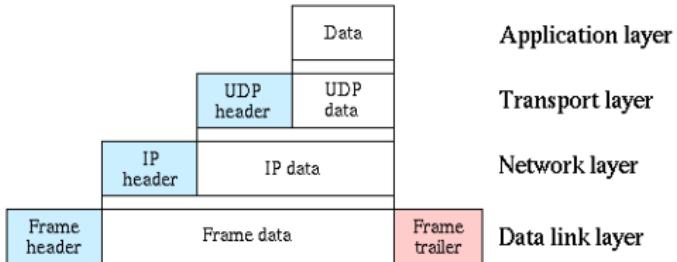


Figura 3: Esempio di incapsulamento

Apprendimento

Questa immagine rappresenta il processo di incapsulamento dei dati nei protocolli di rete. I dati dell'applicazione vengono progressivamente incapsulati da header (intestazioni) a ogni livello: prima il UDP header (livello trasporto), poi l'IP header (livello rete), e infine il frame header e trailer (livello data link). Ogni livello aggiunge le proprie informazioni di controllo prima dell'invio sulla rete. Una volta raggiunto il destinatario, avverrà il processo inverso, ovvero il decapsulamento.



Protocollo di trasporto

Il **protocollo di trasporto** fornisce la comunicazione **end-to-end** tra le due terminali su cui girano gli applicativi, regola il flusso delle informazioni, pu' o fornire un trasporto affidabile, cio'e con recupero degli errori o inaffidabile. I protocolli principali di questo livello sono il TCP e l'UDP.

Le principali differenze tra TCP e UDP sono :

- **TCP** : affidabile, orientato alla connessione.
- **UDP** : non affidabile, senza connessione, ma più veloce.



Trasmission Control Protocol

È un protocollo **orientato alla connessione** che provvede un trasporto affidabile per un flusso di dati bidirezionale fra due stazioni remote. Il protocollo si prende carico di tutti gli aspetti del trasporto dei dati, come l'acknowledgement, il timeout, la ritrasmissione, ecc.. È usato dalla maggior parte delle applicazioni.

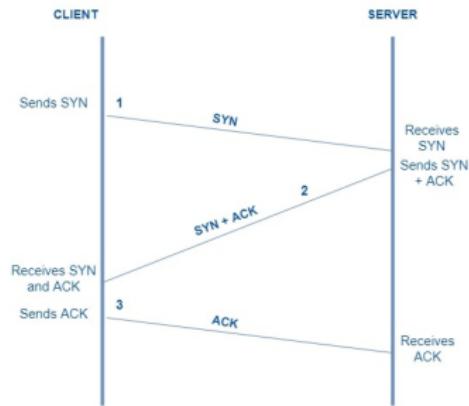


Figura 4: Handshake

Interazione delle Applicazione con lo Stack di rete

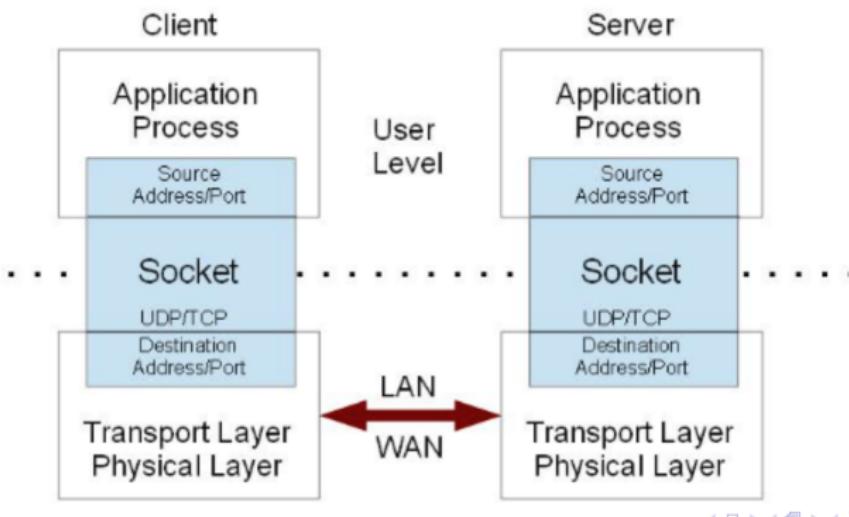


Figura 5: L'immagine mostra come un'applicazione client e un'applicazione server comunichino a livello utente tramite socket, appoggiandosi ai protocolli TCP/UDP, e utilizzando la rete (LAN o WAN) per il trasporto fisico dei dati.



Socket

Un **socket** è un oggetto software che permette l'invio e la ricezione di dati tra due endpoint, questi possono essere due host remoti o due processsi/threads locali. Possiamo immaginarlo come una sorta di presa o porta virtuale: quando due programmi vogliono scambiarsi dati, ciascuno "si collega" a un socket, e da quel momento possono inviare e ricevere informazioni come se stessero parlando attraverso un cavo invisibile

Network socket

Un socket network è un meccanismo usato per scambiare informazioni tra entità computazionali differenti che sfrutta lo stack TCP/IP attraverso un'interfaccia di rete fisica o virtuale.

Everything is a file!

I socket sono trattati come file in Unix e Linux perché essi si fondano sull'idea "che tutto è un file". Utilizzare un approccio simile semplifica l'utilizzo e l'iterazione con le risorse, come dispositivi hardware, file su disco, pipe ecc..

Questa visione uniforme semplifica enormemente il modo in cui si interagisce con il sistema. Per esempio, per leggere informazioni da un sensore o scrivere su un dispositivo hardware, Linux ti permette di "leggere" o "scrivere" su un file speciale, come se stessi aprendo un normale file di testo.

Questo porta a due aspetti da considerare :

- **Uniformità** : L'accesso avviene attraverso operazioni su file.
- **Astrazione** : Un file è un'astrazione che rappresenta una sorgente o destinazione di dati.

Creazione di una socket

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

Approfondimento

Quando avviene una chiamata di sistema, viene interrotto il kernel e gli viene passato il controllo. La funzione sys_Socket() del kernel è responsabile della creazione del socket con l'indirizzo(AF_INET fa riferimento a IPv4) e tipo di famiglia specificati(SOCK_STREAM indica che stiamo creando una socket TCP).



Interfaccia Socket : bind

```
1 if (bind(server_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){  
2     perror("bind failed");  
3     exit(EXIT_FAILURE);  
4 }  
5 }
```

Spiegazione

Viene associato al socket l'indirizzo e la porta specificati. Tale informazione è contenuta nel secondo parametro, un puntatore alla struttura sockaddr che contiene le informazioni sull'indirizzo IP e la porta a cui associare la socket. Il terzo parametro specifica la dimensione di quella struttura (sizeof(serv_addr)), necessaria per sapere quanti byte leggere. Se il binding fallisce, viene stampato un messaggio di errore, il socket viene chiuso e il programma termina.



```
1     if (listen(server_fd, MAX_CLIENTS) < 0){  
2         perror("listen failed");  
3         exit(EXIT_FAILURE);  
4     }  
5 }
```

Spiegazione

La funzione listen mette in ascolto il socket per connessioni in entrata, con una coda massima di MAX_CLIENTS(10000) connessioni pendenti. Se fallisce viene stampato un messaggio di errore, il socket viene chiuso e il programma termina.

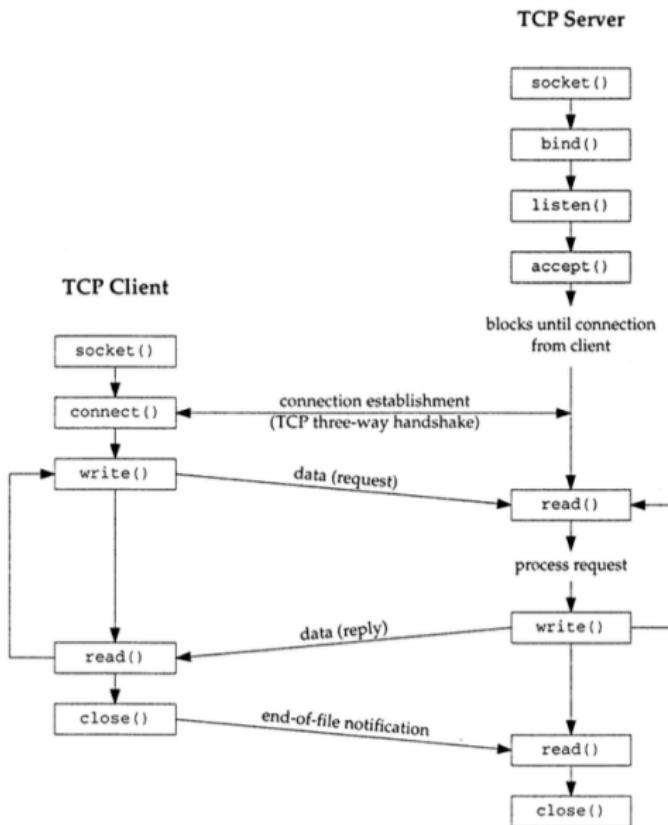
Interfaccia Socket : accept

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

Spiegazione

Accetta una connessione in entrata, prende come parametri il file descriptor del server, il secondo parametro è un puntatore a una struttura sockaddr dove verranno salvate le informazioni sul client che si connette (IP e porta). Il terzo parametro è un puntatore alla dimensione di quella struttura, che accept() aggiorna con la dimensione effettiva dell'indirizzo del client.





Architettura generale

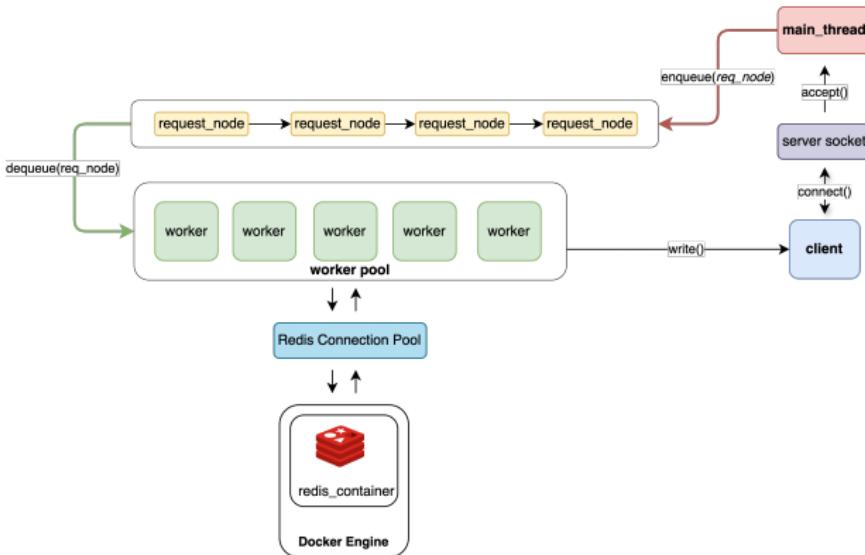


Figura 7: Architettura progettata

Moduli del codice

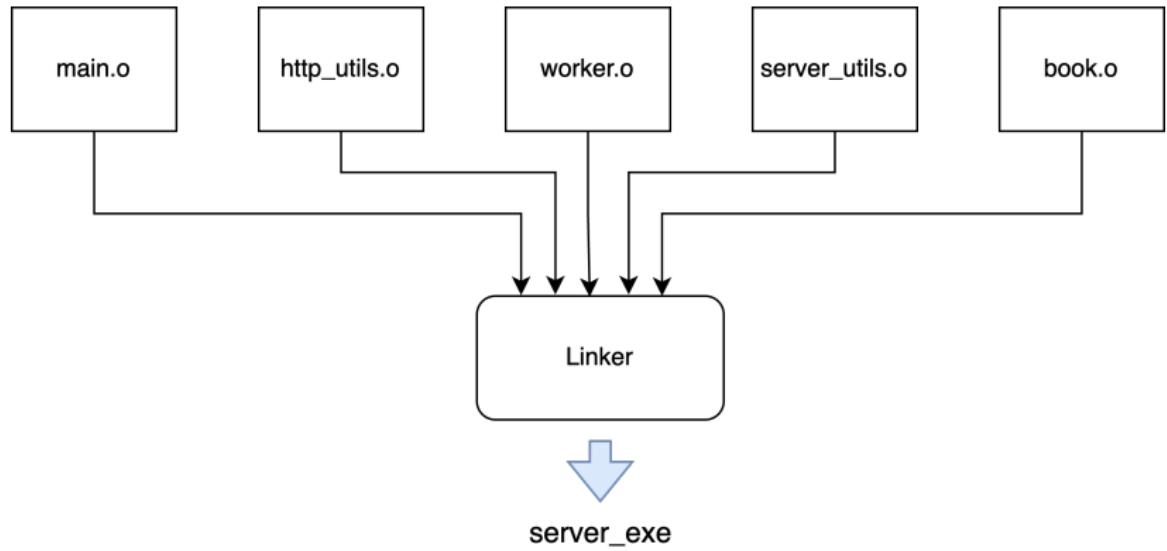


Figura 8: Divisione modulare del codice



Algorithm 1 Definizione Struttura HTTPRequest

- 1: **Struttura** HTTPRequest:
- 2: method: HTTPMethod
- 3: method_str: String[MAX_METHOD_LEN]
- 4: uri: String[MAX_URI_LEN]
- 5: path: String[MAX_URI_LEN]
- 6: version: String[MAX_VERSION_LEN]
- 7: headers: Array[HTTPHeader, MAX_HEADERS]
- 8: header_count: Integer
- 9: body: Pointer to Char
- 10: body_length: Size_t
- 11: content_length: Size_t

Algorithm 2 Definizione Struttura HTTPResponse

```
1: Struttura HTTPResponse:  
2: version: String[MAX_VERSION_LEN]  
3: status_code: HttpStatus  
4: status_message: String[MAX_STATUS_MESSAGE_LEN]  
5: headers: Array[HTTPHeader, MAX_HEADERS]  
6: header_count: Integer  
7: body: Pointer to Char  
8: body_length: Size_t  
9: raw_response: Pointer to Char  
10: raw_response_size: Size_t
```

Algorithm 3 Definizione Struttura Client Request Node

- 1: **Struttura** ClientRequestNode:
- 2: client_fd: Integer
- 3: request: HTTPRequest
- 4: next: Pointer to ClientRequestNode

Algorithm 4 Definizione Struttura Request Queue

```
1: Struttura RequestQueue:  
2: front: Pointer to ClientRequestNode  
3: rear: Pointer to ClientRequestNode  
4: size: Integer  
5: maxSize: Integer  
6: mutex: PthreadMutex  
7: notEmpty: PthreadCond  
8: notFull: PthreadCond  
9: emptySlots: Semaphore  
10: fullSlots: Semaphore  
11: totalProduced: Integer  
12: totalConsumed: Integer
```

- ▷ Primo elemento (dequeue)
- ▷ Ultimo elemento (enqueue)
- ▷ Numero elementi attuali
- ▷ Dimensione massima coda
- ▷ Mutex per accesso esclusivo
- ▷ Condition variable coda non vuota
- ▷ Condition variable coda non piena
 - ▷ Semaforo per slot vuoti
 - ▷ Semaforo per slot pieni
 - ▷ Totale elementi prodotti
 - ▷ Totale elementi consumati



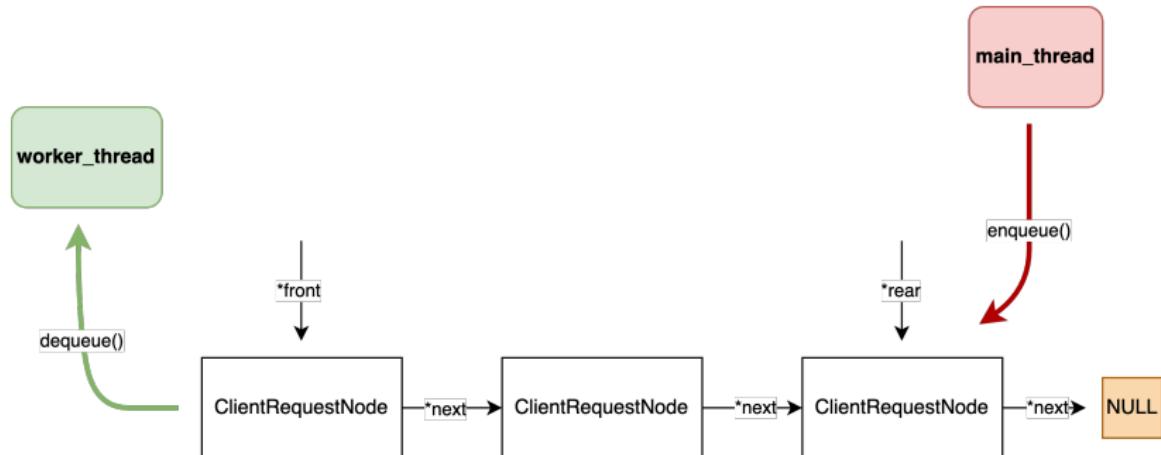


Figura 9: La rappresentazione mostra come il main_thread accetta le richieste e le accoda per la gestione, mentre uno dei thread worker effettua le operazioni di gestione della coda

Algorithm 5 Definizione Struttura Worker Pool

```
1: Struttura WorkerPool:  
2: threads: Pointer to PthreadT  
3: num_threads: Integer  
4: queue: Pointer to RequestQueue  
5: process_function: Pointer to Function(Pointer) → Pointer  
6:  
7: Variabile Globale:  
8: worker_pool: Pointer to WorkerPool
```



]

Algorithm 6 Algoritmo Principale del Server

```
1: function MAIN                                ▷ Inizializzazione pool Redis
2:     redis_pool ← malloc(sizeof(WorkerPool))
3:     INIT_REDISH_POOL(10, redis_pool)           ▷ Inizializzazione server
4:     server_fd ← INITIALIZE_SERVER(SERVER_PORT)
5:     if server_fd < 0 then
6:         return EXIT_FAILURE
7:     end if                                    ▷ Configurazione epoll
8:     epoll_fd ← INIT_EPOLL_INSTANCE
9:     ADD_FD_TO_EPOLL_INSTANCE(server_fd, epoll_fd, EPOLLIN)
10:    events: Array[EpollEvent, MAX_EVENTS]
11: end function
```

Continua ...



Main Event Loop

Algorithm 7 Ciclo Principale degli Eventi

```
1: function MAIN(continuazione)                                ▷ Main event loop
2:   while true do
3:     num_events ← EPOLL_WAIT(epoll_fd, events, MAX_EVENTS, -1)
4:     if num_events = -1 then
5:       if errno = EINTR then
6:         continue                                         ▷ Segnale ricevuto, continua
7:       end if
8:       perror("epoll_wait failed")
9:       break
10:      end if
11:      if num_events > 0 then
12:        result ← PROCESS_EPOLL_EVENTS(server_fd, epoll_fd, events, num_events)
13:        if result < 0 then
14:          print("Errore nel processare gli eventi")    ▷ Decisione se continuare o uscire
15:        end if
16:      end if
17:    end while                                         ▷ Cleanup
18:    CLEANUP_RESOURCES(server_fd, epoll_fd)
19:    return EXIT_SUCCESS
20: end function
```



Algorithm 8 Funzione di Inizializzazione Server

```
1: function INITIALIZE_SERVER(port)
2:   print("Avvio del server...")
3:   worker_pool ← WORKER_POOL_INIT(10, worker_thread)           ▷ Inizializzazione worker pool
4:   server_fd ← INIT_SERVER_SOCKET(port)
5:   if server_fd < 0 then
6:     print("Errore nell'inizializzazione del socket del server")
7:     return -1
8:   end if
9:   print("Server in ascolto sulla porta ", port)
10:  return server_fd                                         ▷ Ritorna il file descriptor del server
11: end function
```



Algorithm 9 Funzione di Inizializzazione Socket Server

```
1: function INIT_SERVER_SOCKET(port)
2:   server_fd: Integer
3:   serv_addr: SocketAddrIn                                ▷ Struttura per IP e porta
4:   server_fd ← SOCKET(AF_INET, SOCK_STREAM, 0)           ▷ Creazione socket
5:   if server_fd < 0 then
6:     exit(EXIT_FAILURE)
7:   end if                                              ▷ Configurazione indirizzo server
8:   serv_addr.sin_family ← AF_INET
9:   serv_addr.sin_addr.s_addr ← INADDR_ANY
10:  serv_addr.sin_port ← htons(port)                      ▷ Binding socket all'indirizzo
11:  result ← BIND(server_fd, &serv_addr, sizeof(serv_addr))
12:  if result < 0 then
13:    exit(EXIT_FAILURE)
14:  end if                                              ▷ Messa in ascolto
15:  if LISTEN(server_fd, MAX_CLIENTS) < 0 then
16:    perror("listen failed")
17:    exit(EXIT_FAILURE)
18:  end if
19:  SET_NONBLOCKING(server_fd)
20:  return server_fd
21: end function
```

Definizione

Con il termine I/O Multiplexing intendiamo la tecnica che permette a un singolo thread di monitorare multipli file descriptor per le operazioni di :

- Lettura
- Scrittura
- Eccezioni

Vantaggi

L'utilizzo dell'I/O Multiplexing è importante per diverse ragioni :

- Permette di evitare il busy waiting, ovvero quando un processo attende attivamente l'avverarsi di una condizione eseguendo continuamente controlli in un ciclo senza fare altro. Uso inefficiente di CPU.
- Permette di usare un singolo thread per gestire connessioni concorrenti.
- E infine, com'è possibile intuire dai due punti precedenti, ottimizzare l'utilizzo delle risorse.



Algorithm 10 Funzione di Inizializzazione Epoll

```
1: function INIT_EPOLL_INSTANCE    ▷ Crea una nuova istanza epoll e ritorna il  
   file descriptor  
2:     epoll_fd ← EPOLL_CREATE1(0)  
3:     if epoll_fd = -1 then  
4:         perror("epoll_create1")  
5:         exit(EXIT_FAILURE)  
6:     end if  
7:     return epoll_fd  
8: end function
```

Riga 2: Crea una nuova istanza epoll tramite epoll_create1(0). Il parametro 0 usa le impostazioni di default.

Righe 3-6: Gestione degli errori - se la chiamata fallisce (ritorna -1), stampa l'errore e termina il programma.

Riga 7: Ritorna il file descriptor dell'istanza epoll creata

Algorithm 11 Funzione per Aggiungere File Descriptor a Epoll

```
1: function ADD_FD_TO_EPOLL_INSTANCE(fd_to_monitor, epoll_instance,  
   event_type)  
2:   event: EpollEvent  
                                ▷ Configurazione evento  
3:   event.events ← event_type  
4:   event.data.fd ← fd_to_monitor  
                                ▷ Aggiunta file descriptor all'istanza epoll  
5:   result ← EPOLL_CTL(epoll_instance, EPOLL_CTL_ADD,  
   fd_to_monitor, &event)  
6:   if result = -1 then  
7:     perror("epoll_ctl: add server")  
8:     exit(EXIT_FAILURE)  
9:   end if  
10:  return 0  
11: end function
```

Spiegazione

Il multi-threading è una tecnica di programmazione che permette a un programma di eseguire più thread (flussi di esecuzione) contemporaneamente all'interno dello stesso processo. Un thread è un'unità leggera di esecuzione che condivide lo spazio di memoria con altri thread dello stesso processo, ma mantiene il proprio stack e registri del processore. In C, il multi-threading viene implementato principalmente attraverso la libreria POSIX Threads (pthread) su sistemi Unix/Linux.

Vantaggi

Il primo è il parallelismo reale, che permette di sfruttare processori multi-core per eseguire operazioni simultaneamente, riducendo drasticamente i tempi di esecuzione. La responsività, mentre un thread gestisce l'interfaccia, altri thread possono eseguire operazioni pesanti in background senza bloccare l'interazione con l'utente.

Svantaggi

- Sincronizzazione
- debug
- deadlock

Inizializzazione Worker Pool

Algorithm 12 Funzione di Inizializzazione Worker Pool

```
1: function WORKER_POOL_INIT(num_threads, process_func)           ▷ Crea un pool di thread worker
2:   if num_threads ≤ 0 or process_func = NULL then
3:     return NULL
4:   end if
5:   pool ← MALLOC(sizeof(worker_pool_t))
6:   if pool = NULL then
7:     return NULL
8:   end if
9:   pool.threads ← MALLOC(sizeof(pthread_t) × num_threads)
10:  if pool.threads = NULL then
11:    FREE(pool)
12:    return NULL
13:  end if
14:  pool.queue ← CREATEQUEUE(20)
15:  if pool.queue = NULL then
16:    FREE(pool.threads)
17:    FREE(pool)
18:    return NULL
19:  end if
20:  pool.num_threads ← num_threads
21:  pool.process_function ← process_func                                ▷ Crea i thread worker
22:  for i = 0 to num_threads - 1 do
23:    thread_id ← i
24:    if PTHREAD_CREATE(pool.threads[i], NULL, process_func, thread_id) ≠ 0 then
25:      pool.num_threads ← i                                         ▷ Aggiorna numero thread creati
26:      WORKER_POOL_DESTROY(pool)
27:      return NULL
28:    end if
29:  end for
30:  return pool
31: end function
```

Algorithm 13 Funzione Thread Worker

```
1: function WORKER_THREAD(arg)                                ▷ Funzione eseguita da ogni thread worker
2:   SLEEP(3)
3:   c ← GET_REDIS_CONNECTION
4:   if c = NULL then
5:     return
6:   end if
7:   incoming_request ← MALLOC(sizeof(client_request_node_t))
8:   while true do
9:     if DEQUEUE_NODE(worker_pool.queue, incoming_request) then
10:      response ← PROCESS_REST_REQUEST(incoming_request.request, c)
11:      if response ≠ NULL then
12:        response_string ← GET_RESPONSE_STRING(response)
13:        if response_string ≠ NULL then
14:          PRINTF("\n— Risposta raw —\n")
15:          PRINTF(response_string)
16:        end if
17:        SEND(incoming_request.client_fd, response_string, strlen(response_string), 0)
18:        FREE_HTTP_RESPONSE(response)
19:      end if
20:    end if
21:    PRINTQUEUE(worker_pool.queue)
22:  end while
23:  return NULL
24: end function
```

Nascono dei problemi...

Cosa succede se il `main_thread` trova la coda `request_queue` piena ?

Cosa succede se un `worker_thread` non trova elementi da prelevare dalla coda ?

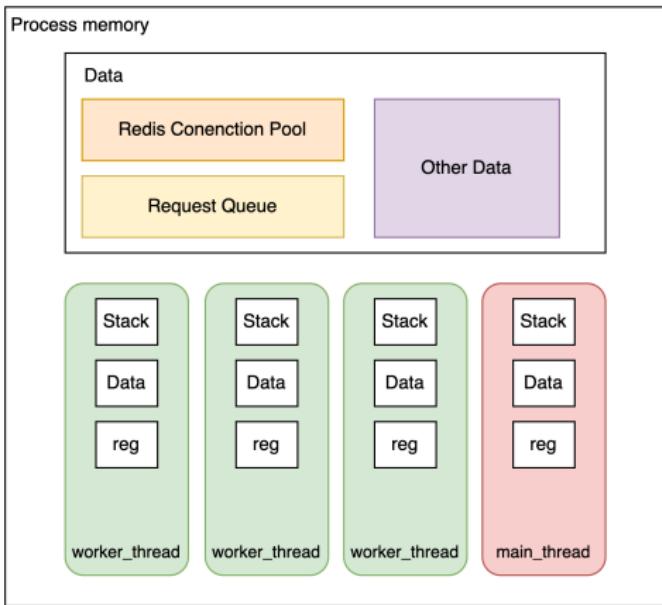


Figura 10: Rappresentazione semplificata della memoria



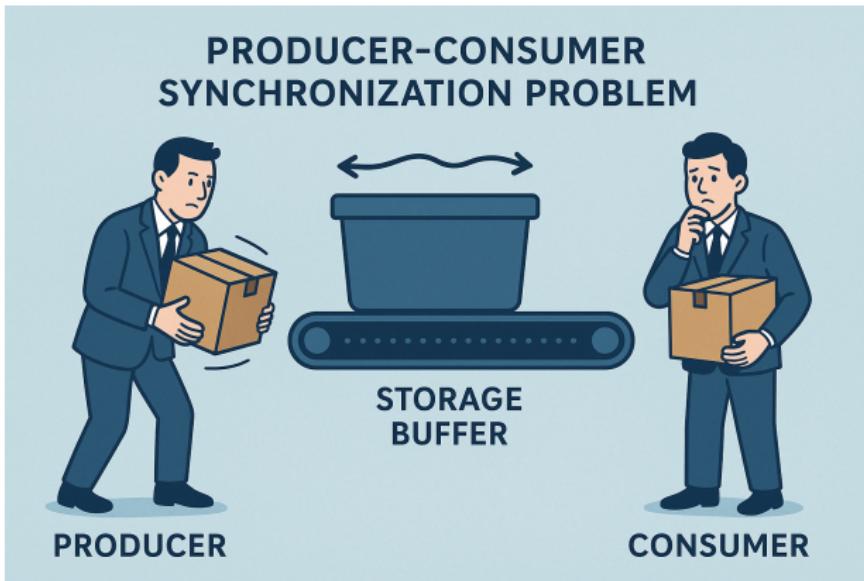


Figura 11: Produttore-consumatore

Descrizione del Problema:

- Un **produttore** genera dati o elementi.
- Un **consumatore** utilizza o elabora questi dati.
- I **dati** vengono **scambiati** tramite un **buffer condiviso** di dimensione finita (o limitata).

Necessità di Sincronizzazione

La sincronizzazione è cruciale per prevenire **race conditions** e garantire l'integrità dei dati presenti nella porzione di **memoria condivisa**.

Il comportamento deve :

- **Buffer Pieno**: Il produttore deve essere bloccato (sospeso) se il buffer è pieno, aspettando che il consumatore liberi spazio.
- **Buffer Vuoto** : Il consumatore deve essere bloccato (sospeso) se il buffer è vuoto, aspettando che il produttore generi nuovi dati.
- **Accesso Concorrente**: Poiché il buffer è una risorsa condivisa, l'accesso simultaneo da parte del produttore e del consumatore può portare a incoerenze dei dati. È necessaria la mutua esclusione per assicurare che solo un thread alla volta possa accedere al buffer.



Il problema viene tipicamente risolto utilizzando una combinazione di primitive di sincronizzazione:

- **Mutex (Mutua Esclusione):** Un mutex è impiegato per garantire che l'accesso al buffer condiviso avvenga in mutua esclusione, impedendo che produttore e consumatore scrivano o leggano contemporaneamente dalla stessa area di memoria del buffer, prevenendo così le condizioni di gara.
- **Semafori:** I semafori sono usati per la sincronizzazione tra il produttore e il consumatore, gestendo le condizioni di "buffer pieno" e "buffer vuoto".
- **Variabili di Condizione (Condition Variables) :** Le variabili di condizione offrono un meccanismo più esplicito per la sincronizzazione, permettendo ai thread di attendere che una specifica condizione si verifichi.

Algorithm 14 Funzione Enqueue Thread-Safe

```
1: function ENQUEUE(q, request)      ▷ Inserisce un elemento nella coda con sincronizzazione
2:   if q = NULL then
3:     return false
4:   end if
5:   SEM_WAIT(q.emptySlots)           ▷ Attendi slot vuoto
6:   PTHREAD_MUTEX_LOCK(q.mutex)      ▷ Accesso esclusivo
7:   while isFullUnsafe(q) do
8:     PTHREAD_COND_WAIT(q.notFull, q.mutex)
9:   end while
10:  newNode ← MALLOC(sizeof(client_request_node_t))
11:  newNode.request ← request
12:  newNode.next ← NULL
13:  if isEmpty(q) then
14:    q.front ← newNode
15:    q.rear ← newNode
16:  else
17:    q.rear.next ← newNode
18:    q.rear ← newNode
19:  end if
20:  q.size ← q.size + 1
21:  PTHREAD_COND_SIGNAL(q.notEmpty)    ▷ Segnala elemento disponibile
22:  PTHREAD_MUTEX_UNLOCK(q.mutex)       ▷ Rilascia mutex
23:  SEM_POST(q.fullSlots)             ▷ Segnala slot occupato
24:  return true
25: end function
```

- **Sincronizzazione multipla:** Combina semafori per il controllo degli slot, mutex per l'accesso esclusivo e condition variables per l'attesa efficiente.
- **Doppio controllo della capacità:** Usa sia semafori che condition variables per gestire la coda piena, evitando race conditions complesse.
- **Inserimento FIFO thread-safe:** Alloca dinamicamente i nodi e li inserisce mantenendo l'ordine, con segnalazione agli altri thread una volta completato.

Algorithm 15 Funzione Dequeue Thread-Safe

```
1: function DEQUEUE(q, request)      ▷ Rimuove un elemento dalla coda con sincronizzazione
2:   if q = NULL or request = NULL then
3:     return false
4:   end if
5:   SEM_WAIT(q.fullSlots)           ▷ Attendi elemento disponibile
6:   PTHREAD_MUTEX_LOCK(q.mutex)    ▷ Accesso esclusivo
7:   while isEmpty(q) do
8:     PTHREAD_COND_WAIT(q.notEmpty, q.mutex)
9:   end while
10:  if isEmpty(q) then
11:    PTHREAD_MUTEX_UNLOCK(q.mutex)
12:    SEM_POST(q.fullSlots)
13:    return false
14:  end if
15:  temp ← q.front
16:  request ← temp.request
17:  q.front ← q.front.next
18:  if q.front = NULL then
19:    q.rear ← NULL
20:  end if
21:  q.size ← q.size - 1
22:  FREE(temp)
23:  PTHREAD_COND_SIGNAL(q.notFull)  ▷ Segnala slot disponibile
24:  PTHREAD_MUTEX_UNLOCK(q.mutex)   ▷ Rilascia mutex
25:  SEM_POST(q.emptySlots)        ▷ Segnala slot vuoto
26:  return true
27: end function
```



- **Attesa sincronizzata:** Usa semafori per attendere elementi disponibili e condition variables per gestire code vuote senza busy-waiting
- **Rimozione FIFO thread-safe:** Estrae l'elemento dalla testa della coda, aggiorna i puntatori front/rear e libera correttamente la memoria
- **Segnalazione simmetrica:** Notifica agli altri thread la disponibilità di slot vuoti tramite condition variables e semafori per mantenere il bilanciamento

Algorithm 16 Funzione di Elaborazione Eventi Epoll

```
1: function PROCESS_EPOLL_EVENTS(server_fd, epoll_fd, events, num_events) ▷ Elabora gli
   eventi epoll ricevuti
2:   for i = 0 to num_events - 1 do
3:     current_fd ← events[i].data.fd
4:     if current_fd = server_fd then                                ▷ Nuova connessione in arrivo
5:       if HANDLE_NEW_CONNECTION(server_fd, epoll_fd) < 0 then
6:         PRINTF("Errore nell'accettare la connessione")
7:       end if
8:     else                                              ▷ Dati pronti per la lettura da un client
9:       result ← HANDLE_CLIENT_DATA(current_fd)
10:      if result < 0 then
11:        PRINTF("Errore nella gestione dei dati del client")
12:      end if
13:    end if
14:   end for
15:   return 0
16: end function
```

Algorithm 17 Funzione di Gestione Dati Client

```
1: function HANDLE_CLIENT_DATA(client_fd)           ▷ Gestisce i dati ricevuti da un client
2:   receiving_buffer ← MEMSET(BUFFER_SIZE, 0)
3:   received_data_size ← RECV(client_fd, receiving_buffer, BUFFER_SIZE - 1, 0)
4:   if received_data_size > 0 then                   ▷ Dati ricevuti con successo
5:     receiving_buffer[received_data_size] ← '\0'
6:     request ← CREATE_HTTP_REQUEST
7:     if request = NULL then
8:       return -1
9:     end if
10:    if PARSE_HTTP_REQUEST(receiving_buffer, request) ≠ 0 then
11:      FREE_HTTP_REQUEST(request)
12:      return -1
13:    end if
14:    newNode ← MALLOC(sizeof(client_request_node_t))
15:    if newNode = NULL then
16:      return -1
17:    end if
18:    newNode.request ← request
19:    newNode.client_fd ← client_fd
20:    newNode.next ← NULL
21:    ENQUEUE_NODE(worker_pool.queue, newNode)
22:    return 0
23:  else if received_data_size = 0 then             ▷ Client disconnesso
24:    CLOSE(client_fd)
25:    return 1
26:  else                                         ▷ Errore nella recv
27:    if errno ≠ EAGAIN and errno ≠ EWOULDBLOCK then
28:      CLOSE(client_fd)
29:      return -1
30:    end if
31:    return 0
```



Le origini

Redis (REmote DIctionary Server) è stato creato da Salvatore Sanfilippo nel 2009, inizialmente per migliorare la scalabilità di un suo progetto personale. Nato come un semplice key-value store in memoria, Redis si è evoluto rapidamente in un database NoSQL molto performante, grazie al supporto di strutture dati avanzate come liste, set e hash. Dopo l'adozione da parte di aziende come GitHub e Twitter, è diventato uno dei database in-memory più usati al mondo. Nel 2015 Sanfilippo ha donato il progetto alla comunità, e dal 2020 ha lasciato la leadership attiva.

Redis è un database in-memory che memorizza i dati principalmente nella RAM, garantendo altissime prestazioni in lettura e scrittura.

Funziona secondo un modello **key-value**: ogni valore è associato a una chiave univoca, ma supporta anche strutture dati complesse come stringhe, liste, set, hash, sorted set, bitmap e hyperloglog. Questo permette una gestione più flessibile e potente delle informazioni.

Redis è single-threaded, ma estremamente veloce perché riduce al minimo il tempo di blocco tra le operazioni e sfrutta al massimo l'accesso diretto alla memoria.

Redis offre anche diverse modalità di persistenza: può salvare periodicamente lo stato su disco (**snapshotting**) o registrare ogni modifica in un file di log (**AOF - Append Only File**), rendendolo adatto anche a contesti in cui non si può perdere nessun dato. Inoltre, Redis supporta funzionalità avanzate :

- **replica** per alta disponibilità.
- **clustering** per scalabilità.
- **pubblicazione/sottoscrizione** di messaggi.
- **scripting** con Lua.

Queste caratteristiche lo rendono molto più di una semplice cache: è un vero e proprio motore di dati in tempo reale.



Algorithm 18 Connessione a Redis

```
1: function CONNECT _ REDIS                                ▷ Stabilisce connessione a Redis
2:   c ← REDISCONNECT("127.0.0.1", 6379)
3:   if c = NULL or c.err then
4:     if c ≠ NULL then
5:       PRINTF("Errore connessione: " + c.errstr)
6:       REDISFREE(c)
7:     end if
8:     return NULL
9:   end if
10:  return c
11: end function
```

Algorithm 19 Salvataggio Libro

```
1: function SAVE _ BOOK(c, book)                          ▷ Salva un libro usando HSET
2:   key ← SNPRINTF("book:%d", book.id)
3:   reply ← REDISCOMMAND(c, "HSET %s id %d title %s author %s price %.2f",
4:   key, book.id, book.title, book.author, book.price)
5:   if reply = NULL then
6:     return -1
7:   end if
8:   FREEREPLYOBJECT(reply)
9:   return 0
9: end function
```

Algorithm 20 Caricamento Libro

```
1: function LOAD_BOOK(c, book_id)                                ▷ Carica un libro usando HGETALL
2:   key ← SNPRINTF("book:%d", book_id)
3:   reply ← REDISCOMMAND(c, "HGETALL %s", key)
4:   if reply = NULL then
5:     return NULL
6:   end if
7:   if reply.type = REDIS_REPLY_ARRAY and reply.elements > 0 then
8:     book ← MALLOC(sizeof(Book))
9:     if book = NULL then
10:      FREEREPLYOBJECT(reply)
11:      return NULL
12:    end if
13:    for i = 0 to reply.elements - 1 step 2 do
14:      field ← reply.element[i].str
15:      value ← reply.element[i + 1].str
16:      if field = "id" then
17:        book.id ← ATOI(value)
18:      else if field = "title" then
19:        STRNCPY(book.title, value, sizeof(book.title) - 1)
20:      else if field = "author" then
21:        STRNCPY(book.author, value, sizeof(book.author) - 1)
22:      else if field = "price" then
23:        book.price ← ATOF(value)
24:      end if
25:    end for
26:  end if
27:  FREEREPLYOBJECT(reply)
28:  return book
29: end function
```



Algorithm 22 Aggiornamento Prezzo

```
1: function UPDATE_BOOK_PRICE(c, book_id, new_price) ▷ Aggiorna il prezzo usando HSET
2:   key ← SNPRINTF("book:%d", book_id)
3:   reply ← REDISCOMMAND(c, "HSET %s price %.2f", key, new_price)
4:   if reply = NULL then
5:     return -1
6:   end if
7:   FREEREPLYOBJECT(reply)
8:   return 0
9: end function
```

Algorithm 23 Funzione di Eliminazione Libro

```
1: function DELETE_BOOK(c, book_id) ▷ Elimina un libro dal database Redis
2:   key ← SNPRINTF("book:%d", book_id)
3:   reply ← REDISCOMMAND(c, "DEL %s", key)
4:   if reply = NULL then
5:     PRINTF("Errore eliminazione libro")
6:     return -1
7:   end if
8:   PRINTF("Libro eliminato: %s", key)
9:   FREEREPLYOBJECT(reply)
10:  return 0
11: end function
```

Test - Create

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (6)', 'Test Results', and a refresh icon. On the right, status information is displayed: '201 Created' with a green background, '24 ms', '308 B', and a globe icon. Below these are buttons for 'Copy', 'Find', 'Delete', and 'Edit'. The main area has a 'JSON' dropdown menu with options like '[]', 'JSON', 'XML', etc., followed by 'Preview' and 'Visualize' buttons. The JSON payload is shown in a code editor:

```
1 {  
2     "id_book": 1,  
3     "title": "Il signore degli anelli",  
4     "author": "autore1",  
5     "price": 120.0  
6 }
```

Figura 12: Create

```
1 post a /add/book  payload :{  
2     "id_book": 1,  
3     "title" : "Il signore degli anelli",  
4     "author" : "autore1",  
5     "price" : 120.0  
6 }  
7 }
```



Test - Read

The screenshot shows a REST API testing interface with the following details:

- Body:** JSON response content:

```
1 {  
2   "id_book": 1,  
3   "title": "Il signore degli anelli",  
4   "author": "autore1",  
5   "price": 120.00  
6 }
```
- Headers:** Headers tab (partially visible)
- Status:** 200 OK
- Time:** 4 ms
- Size:** 300 B
- Icon:** A globe icon indicating international reach.
- Tools:** A toolbar with icons for copy, search, and refresh.

Figura 13: Read

```
1 get a 8080/get/books payload :{  
2   "id_book" : 1  
3 }  
4
```



Test - Update

Body Cookies Headers (6) Test Results ⚡

200 OK • 2 ms • 268 B • 🔍

{ } JSON ▾ Preview ⚡ Visualize ▾

```
1 {  
2   "id_book": 1,  
3   "title": "",  
4   "author": "",  
5   "price": 30.00  
6 }
```

A screenshot of a REST API testing interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (6)', and 'Test Results'. On the right, a status bar shows '200 OK', '2 ms', and '268 B' with icons for search and refresh. Below these are buttons for 'Copy', 'Edit', 'Search', and 'Close'. The main area displays a JSON payload with line numbers 1 through 6. Line 1 starts a brace {}, line 2 contains "id_book": 1, line 3 contains "title": "", line 4 contains "author": "", line 5 contains "price": 30.00, and line 6 ends with a brace }.

Figura 14: Update

```
1 PUT a /update/book payload : {  
2   "id_book" : 1,  
3   "price" : 30.0  
4 }  
5
```



Test - Delete

Body Cookies Headers (6) Test Results ⓘ

204 No Content • 4 ms • 275 B • ⓘ

{ } JSON ▾ Preview ⚡ Visualize ▾

```
1 {  
2     "id_book": 1,  
3     "title": "",  
4     "author": "",  
5     "price": 0.00  
6 }
```

Figura 15: Delete

```
1 DELETE a /delete/book payload {  
2     "id_book": 1  
3 }  
4
```



Test - Errore

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (5)', 'Test Results', and a refresh icon. On the right, status information is displayed: '404 Not Found' in red, '2 ms', '207 B', and a circular progress bar. Below the tabs, there are buttons for 'Preview' and 'Visualize'. The main area shows a JSON response with three lines of code:

```
1 {  
2   "error": "Endpoint non trovato"  
3 }
```

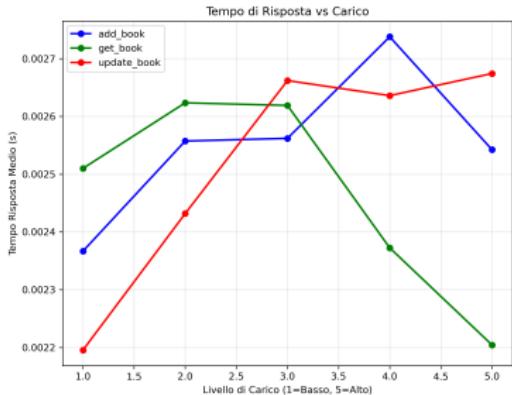
Figura 16: Esempio non trovato

```
1 POST a /aggiungi/book payload : {  
2     "id_book": 1,  
3     "title" : "Libro",  
4     "author" : "autore1",  
5     "price" : 120.0  
6 }  
7
```

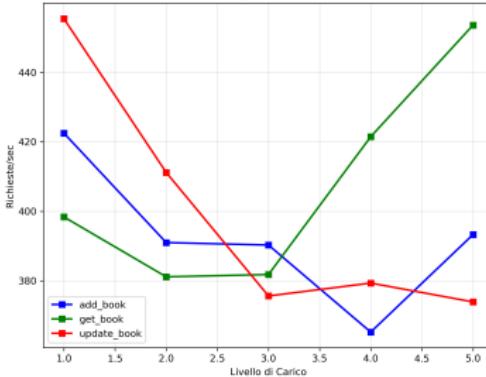


Test di carico

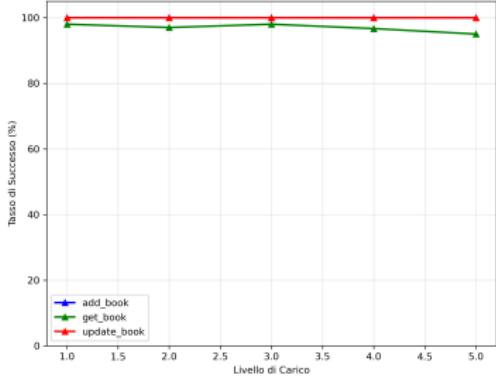
Test di Carico Server REST - Risultati



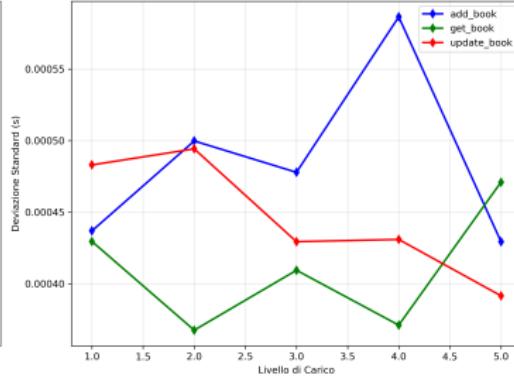
Throughput vs Carico



Tasso di Successo vs Carico

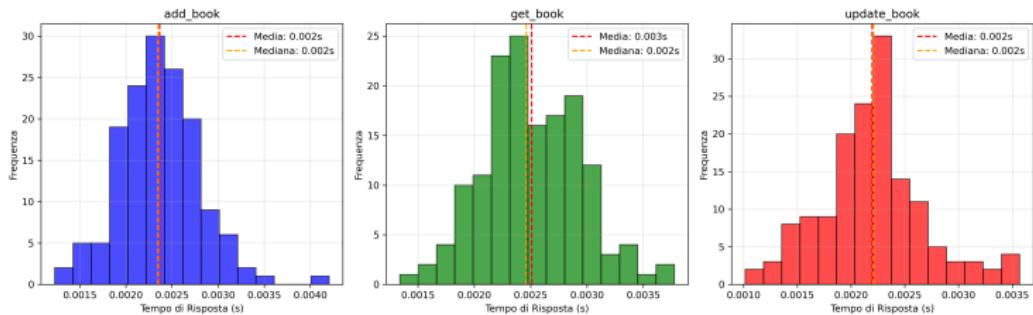


Variabilità Tempi di Risposta

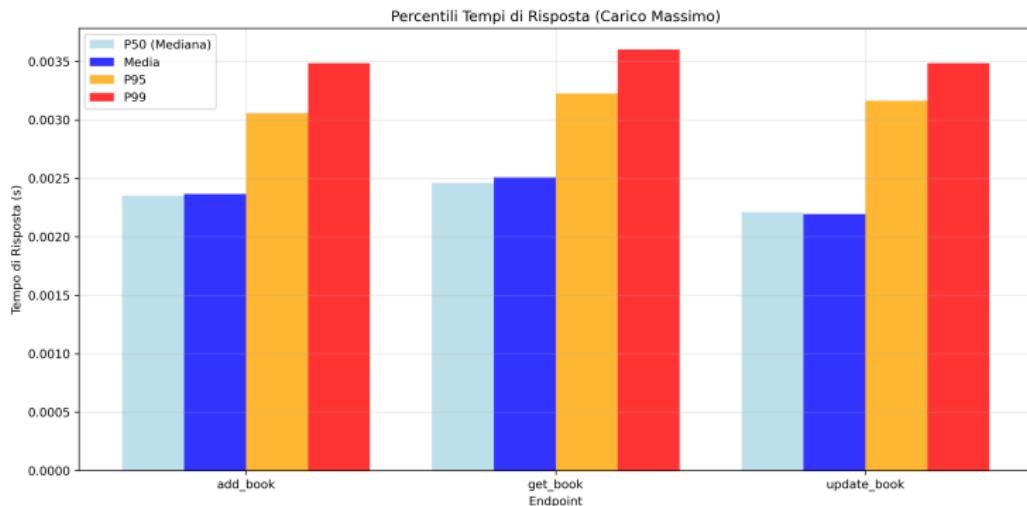


Test di carico

Distribuzione Tempi di Risposta (Carico Massimo)



Test di carico



Possibili miglioramenti

- **Gestione Robusta degli errori** : Descrizione accurata degli allert di errori, evitare situazioni di segmentation fault, garantire il principio di fault tollerance.
- **Sistema di Logging** : Per garantire un accesso sicuro al sistema tramite un nome utente e password.
- **Feature di sicurezza** : La versione corrente usa HTTP, si potrebbe pensare di passare ad HTTPS, implementare un meccanismo di autenticazione degli utenti...
- **Adattabilità al carico di lavoro** : si potrebbero implementare algoritmi capaci di stimare il carico di ogni worker e lanciarne altri all'occorrenza.