

Progetto

Andrea Caltagirone

17 febbraio 2026

- Introduzione al problema
- Stato dell'arte
- Metodologia usata
- Risultati sperimentali
- Conclusione

Introduzione al problema

Il problema chiede di creare un server concorrente in C che gestisca almeno le richieste GET, POST, PUT, DELETE, per fare ciò sono stati utilizzati vari elementi affrontati durante il corso come i socket ed i thread.

Stato dell'arte: Socket

L'interfaccia socket in linux è un pilastro della programmazione di rete essi vengono utilizzati per stabilire una comunicazione bidirezionale tra due endpoint, alcune funzioni importanti per il loro funzionamento sono

- La funzione socket è utilizzata per creare un nuovo socket
- La funzione listen per mettere per esempio il socket di un server in ascolto per possibili connessioni di client
- La funzione accept per accettare le connessioni in entrata una volta che il socket è stato messo in ascolto attraverso listen
- La funzione connect per connettere un socket ad un server remoto
- Le funzioni send e recv che vengono utilizzate per i socket orientati alle connessioni per inviare e ricevere dati
- Le funzioni sendto e recvfrom invece vengono utilizzate per socket senza connessione sempre per inviare e ricevere dati (a differenza di send e recv bisognerà specificare l'indirizzo della destinazione o della sorgente tra i parametri)

Nei sistemi moderni un processo può essere costituito da più unità di esecuzione, chiamate thread, ciascuna delle quali viene eseguita nel contesto del processo e condivide lo stesso codice e gli stessi dati globali. Alcuni dei vantaggi nell'utilizzo dei thread sono la maggiore facilità nel condividere informazioni tra di essi piuttosto che tra i processi ed inoltre il multi-threading è un modo per rendere i programmi più veloci quando sono presenti più processori/core.

Stato dell'arte: Thread

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);` è il prototipo della funzione `pthread_create` che permette di creare un thread e fargli avviare l'esecuzione della funzione "start_routine".
- `pthread_join` che permette di attendere la terminazione di un thread specificato
- `pthread_exit` che termina il thread corrente

Metodologia

- server.c: gestione socket e loop principale del server
- handle_client.c: parsing richieste HTTP e instradamento ai metodi
- method.c: implementazione di GET, POST, PUT, DELETE
- auth.c; autenticazione Basic Auth con decodifica Base64
- utility.c: funzioni di supporto
- header.h: dichiarazioni di funzioni e variabili globali
- Makefile: compilazione modulare con linking finale

Metodologia: server.c - Inizializzazione

La prima parte di server.c si occupa di includere librerie definire costanti e inizializzare le variabili

- Inclusione librerie: stdio, stdlib, string, unistd, arpa/inet, pthread
- Definizione costanti: BUFFER_SIZE (4096), PORT (8080), BACKLOG (10)
- Dichiaraione variabili: server_fd (socket server), new_sock (socket client)
- Struttura sockaddr_in per indirizzo del server
- Compilazione regex per parsing HTTP

Metodologia: server.c - Setup Socket

Successivamente viene effettuato il setup del socket

- `socket(AF_INET, SOCK_STREAM, 0)` : *crea socket TCPIPv4*
- Configurazione struttura address: famiglia IPv4,
`INADDRANY`(*accetta da qualsiasi interfaccia*), *porta 8080*
- `bind()`: associa il socket del server con la struttura sockaddr
- `listen(server_fd, BACKLOG)` :
imposta il socket in ascolto con codadi 10 connessioni
- Loop infinito: `accept()` blocca fino all'arrivo di un client

Metodologia: server.c - Setup Socket

Una volta effettuate le inizializzazioni per gestire i client dopo averlo accettato con accept verrà avviato un thread per ognuno di essi

- Allocazione dinamica memoria per file descriptor del client (malloc)
- accept(): accetta connessione e ottiene socket del client
- pthread_create(thread_id, NULL, handle_client, new_sock): crea thread dedicato
- Il thread esegue la funzione handle_client passando il socket come parametro
- pthread_detach(thread_id): rilascia automaticamente risorse alla terminazione
- Il server ritorna ad accept() per accettare nuovi client

La funzione
handleclient si occupa di richiamare la funzione corretta per ogni tipo di richiesta

Recupero file descriptor client dal parametro e liberazione memoria

Dichiarazione buffer (4096 byte), array method e path per parsing

read(client_fd, buffer, BUFFER_SIZE - 1) : lettura richiesta HTTP dal client

regexec(re_http, buffer, 3, matches, 0) : verifica match con pattern HTTP

Estrazione metodo (GET/POST/PUT/DELETE) da matches[1]

Estrazione path del file da matches[2]

Metodologia: handle_client - Autenticazione

Successivamente effettua un controllo per verificare se il client stia cercando di accedere a cartelle protette

- Controllo presenza stringa "admin" nel path richiesto
- Se presente, ricerca header "Authorization:" nella richiesta
- Estrazione credenziali e chiamata a `checkauth()` per verifica
- Se autenticazione fallisce: `sendResponse(401, "Unauthorized")`
- Se autenticazione OK: prosegue con gestione richiesta
- Path non protetti: nessun controllo, accesso diretto

Metodologia: handle_client - Routing

In fine in base alla richiesta http specifica richiamerà la funzione corretta che si occuperà di essà

- Serie di if per verificare il valore di method
- GET: chiamata a `handle_get(client_fd, path)`
- POST: estrazione body dalla richiesta, chiamata `handle_post(client_fd, path, body)`
- PUT: estrazione body, chiamata `handle_put(client_fd, path, body)`
- DELETE: chiamata a `handle_delete(client_fd, path)`
- Metodo non supportato: `send_response(400, "BadRequest")`
- Chiusura socket client e terminazione thread

Metodologia: handle_client - Routing

In fine in base alla richiesta http specifica richiamerà la funzione corretta che si occuperà di essà

- Serie di if per verificare il valore di method
- GET: chiamata a `handle_get(client_fd, path)`
- POST: estrazione body dalla richiesta, chiamata `handle_post(client_fd, path, body)`
- PUT: estrazione body, chiamata `handle_put(client_fd, path, body)`
- DELETE: chiamata a `handle_delete(client_fd, path)`
- Metodo non supportato: `send_response(400, "BadRequest")`
- Chiusura socket client e terminazione thread

Metodologia: handle_get

funzione che si occupa della richiesta GET:

- Costruzione percorso completo:

$\text{WEB_ROOT}(\cdot/\text{www}/) + \text{pathrichiesto}$

- $\text{get_extension}()$: estrae estensione file dal path
- $\text{get_mime_type}()$: determina MIME type in base all'estensione
- $\text{open(file_path, O_RDONLY)}$: apertura file in sola lettura
- Se file non esiste: $\text{send_response}(404, "NotFound")$
- $\text{fstat}()$: ottiene informazioni file, in particolare dimensione
- Allocazione memoria dinamica per contenuto file (malloc)
- $\text{read}()$: lettura contenuto completo del file
- $\text{send_response}(200, \text{mime_type}, \text{file_content}, \text{file_size})$

Metodologia: handle_post

funzione che si occupa della richiesta POST:

- Costruzione percorso completo del nuovo file
- `open(file_path, O_WRONLY|O_CREAT|O_TRUNC, 0644)` : *creazionefile*
- `O_WRONLY` : *apertura in scrittura*
- `O_CREAT` : *crea file se non esiste*
- `O_TRUNC` : *se esiste lo sovrascrive*
- `0644`: permessi rw-r-r-
- `write(fd, body, strlen(body))` : *scrittura contenuto body nel file*
- `close()`: chiusura file descriptor
- `send_response(201, "Created")` : *risposta al client*

Metodologia: handle_put

funzione che si occupa della richiesta PUT:

- Identico a `handle_post` nell'implementazione
- Sovrascrive file esistente o lo crea se non esiste
- `send_response(204, "")` : risposta senza body (`NoContent`)

Metodologia: handle_delete

funzione che si occupa della richiesta PUT: Funzione che si occupa delle richieste DELETE

- Costruzione percorso completo del file da eliminare
- `unlink(file_path)` : *eliminafiledalfilesystem*
- Se file non esiste: `send_response(404, "NotFound")`
- Se eliminazione OK: `send_response(204, "")`

funzione get_extension(path)

- Utilizza strrchr(path, '.') per trovare ultima occorrenza di '.'
- Ritorna stringa vuota se non trova estensione, altrimenti ext+1

Funzione get_mime_type(ext):

- Associa estensioni a MIME type

Metodologia: send_response

funzione `send_response` che si occupa di costruire il messaggio con cui rispondere alla richiesta del client

- Parametri: `client_fd`, `status_code`, `content_type`, `body`, `body_len`
- Switch su `status_code` per determinare `status_text` (es. 200 "OK", 404 "NotFound")
- Costruzione header HTTP con `snprintf()`:
 - Prima riga: `HTTP/1.1 [status_code] [status_text]`
 - `Content-Type`: `[content_type]`
 - `Content-Length`: `[body_len]`
 - Se 401: aggiunge `WWW-Authenticate: Basic realm="Protected Area"`
 - `Connection: close`
- `write(client_fd, header, header_len)`: invio header
- `write(client_fd, body, body_len)`: invio body se presente

Metodologia: Makefile

Il makefile viene utilizzato per compilare e linkare i vari file .c per creare l'eseguibile

- CC = gcc: definizione compilatore
- CFLAGS = -Wall -Wextra -pthread -std=c11: opzioni compilazione
- LDFLAGS = -pthread: opzioni linking
- SRCS: lista file sorgente (*server.c, handle_client.c, method.c, utility.c, auth.c*)
- OBJS = \$(SRCS:.c=.o): trasformazione .c in .o
- TARGET = server: nome eseguibile finale
- Target 'all': dipende da \$(TARGET), compila tutto
- Target \$(TARGET): linka tutti gli OBJS
- Pattern rule %.o: %.c header.h: compila ogni .c in .o
- Target 'clean': rimuove oggetti ed eseguibile

Risultati

- GET: risposta 200 OK - file servito correttamente con MIME type appropriato
- POST: risposta 201 Created - file test.txt creato con successo
- PUT: risposta 204 No Content - file test.txt sovrascritto
- DELETE: risposta 204 No Content - file test.txt eliminato

Obiettivi

- Server HTTP concorrente pienamente funzionante
- Gestione corretta di GET, POST, PUT, DELETE
- Implementazione risposte HTTP: 200, 201, 204, 400, 401
- Sistema di autenticazione Basic Auth per path protetti
- Architettura modulare con Makefile per compilazione
- Utilizzo efficace di socket e thread POSIX