LabReSiD25 progetto di fine corso

Francesco Paladino matricola 541704 (scienze informatiche)

1 Introduzione

La traccia scelta per l'esame di fine corso consiste nella realizzazione di un Server HTTP concorrente in C che gestisca almeno le richieste GET, POST, PUT, DELETE e le risposte principali (200, 201, 204, 400, 401).
(https://datatracker.ietf.org/doc/html/rfc2616)

2 Stato dell'arte

Prima di affrontare a pieno il problema, e' necessario introdurre i concetti fondamentali per la sua risoluzione.

- connessione di tipo TCP.
- programmazione concorrente.
- fondamentali del protocollo HTTP.

2.1 Protocollo TCP

Il TCP (Transmission Control Protocol), definito nella RFC 9293, e' un protocollo che opera al quarto livello ISO/OSI e si occupa del trasporto delle informazioni all'interno della rete. Ecco le principali caratteristiche:

- e' connection oriented.
- garantisce una connessione affidabile e ordinata.
- controlla il flusso dei dati e la congestione all'interno della rete.

2.1.1 Connection oriented

Per connection oriented intendiamo che prima che avvenga la trasmissione dei dati, viene stabilita tra i due host un canale di comunicazione dedicato grazie al processo di "Three-Way Handshake" che comprende in ordine:

1. l'invio di un pacchetto SYN da parte del client al server che contiene un numero casualmente generato a 32 bit detto Initial Sequence Number (ISN). Questo valore segna l'inizio della sequenza di byte (ovvero dei dati veri e propri) e non parte banalmente da zero per evitare l'attacco di tipo TCP Sequence Prediction.

2. l'invio dal server di un pacchetto ACK che indica l'avvenuta ricezione dell'ISN e l'invio del proprio ISN al client.

3. l'invio di un pacchetto ACK da parte del client per segnalare l'avvenuta ricezione dell'ISN.

Queste informazioni, oltre agli altri parametri della connessione, vengono memorizzate in una struttura dati chiamata Transmission Control Block (TCB) all'interno dell'host.

2.1.2 Connessione affidabile e ordinata

La connessione e' affidabile perche' si conosce se le informazioni sono arrivate a destinazione. Quando il mittente invia un segmento, viene avviato un timer detto RTO (Retransmission Timeout). Se il timer scade prima che arrivi l'ACK corrispondente, il mittente assume che il segmento o il suo ACK di ritorno sia andato perso e lo ritrasmette. Il valore di RTO non è fisso: il protocollo TCP lo calcola dinamicamente adattandolo al tempo di andata e ritorno della rete (Round-Trip Time) per essere efficiente a prescindere dalle performance del mezzo trasmissivo e dalla congestione.

L'ordine e' invece garantito dai numeri di sequenza che permettono di avere i segmenti ordinati anche se consegnati al destinatario in modo casuale. Infine viene anche utilizzato l'algoritmo di Controllo a Ridondanza Ciclica (CRC) per capire se i dati ricevuti dal destinatario sono errati. Alla base di questo algoritmo si utilizza l'operatore logico XOR.

2.1.3 Controllo di flusso e della congestione

All'interno di un segmento TCP, per il controllo di flusso e della congestione troviamo:

- la Receive Window (rwnd).
- la Congestion Window (cwnd).

La Receive Window contiene il numero di byte che il mittente ha a disposizione nel suo buffer. Chi riceve un segmento TCP con questo valore impostato a zero, non inviera' altri pacchetti. Per evitare un blocco permanente, il mittente fa partire un timer e alla scadenza verra' inviato un pacchetto sonda (che equivale ad un segmento TCP con un solo byte di dati) per costringere il destinatario a inviare un nuovo ACK con il valore aggiornato della rwnd.

La Congestion Window invece permette di gestire la congestione con la tecnica detta slow start: all'inizio di una connessione la dimensione massima del

segmento (Maximum Segment Size - MSS) parte da un valore basso che viene incrementato con la ricezione di ACK. Nel momento in cui si verifica la perdita di pacchetti, il valore della cwnd viene dimezzato.

2.2 Programmazione concorrente

Per concorrenza intendiamo la possibilita' di eseguire piu' processi o thread all'interno di una macchina. E' importante distinguerla dal parallelismo, in quanto quest'ultimo e' una sottocategoria di concorrenza in cui vengono svolte piu' attivita' con piu' core della CPU. Durante la concorrenza possono verificarsi alcune problematiche alle risorse condivise:

- race condition: i processi condividono una risorsa e il risultato dell'esecuzione finale dipende dall'ordine di esecuzione dei processi.
- deadlock: un insieme di processi che attende un evento che non occorrera' mai, in quando l'accesso alla risorsa condivisa non e' permessa al processo che fara' succedere quell'evento.
- starvation: il processo a cui vengono assegnate poche risorse non riesce a proseguire.

2.2.1 Comunicazioni interprocesso

Le comunicazioni interprocesso permettono di gestire l'accesso alle risorse condivise. Abbiamo due principali metodi:

- mutex: variabili binarie che permettono di gestire l'accesso ad un solo contendente.
- semafori: variabili n-arie che possono essere incrementate o decrementate. Il processo che decrementa il semaforo si bloccherà appena raggiunto lo zero della variabile. Un processo che incrementa il semaforo invece risveglia tutti i processi che si erano bloccati in fase di decremento.

Le operazioni che riguardano la gestione dei mutex e dei semafori sono atomiche, ovvero azioni che devono essere eseguite in un ciclo di clock indivisibile.

2.3 Fondamentali del protocollo HTTP

Il protocollo HTTP e' un protocollo client-server che utilizza il protocollo TCP (porta 80) per la manipolazione di pagine web. Ecco le principali caratteristiche:

- e' stateless, cio' significa che ogni richiesta HTTP e' indipendente.
- le richieste e le risposte HTTP sono basate su testo. Ogni messaggio e' composto da:
  - la start-line che indica il metodo HTTP e la risorsa richiesta.

- l'header che contiene informazioni sulla richiesta o risposta.
- il body del messaggio che contiene i dati veri e propri. Non e' obbligatorio in ogni messaggio HTTP.

HTTP utilizza inoltre dei codici di stato, ovvero numeri di tre cifre che vengono inviati dal server al client per comunicare l'esito di una richiesta. Ecco i principali:

- 200 indica che la risposta e' andata a buon fine.
- 201 indica che La richiesta è stata soddisfatta restituendo la creazione di una nuova risorsa.
- 204 indica che il server ha processato con successo la richiesta e non restituirà nessun contenuto.
- 400 indica che la richiesta non può essere soddisfatta a causa di errori di sintassi.

2.3.1 Metodi HTTP

- GET: ottiene una risorsa. Eventuali parametri vengono inseriti all'interno dell'URL.
- POST: invia dati per essere processati dal server. I dati vengono inviati nel body della richiesta HTTP.
- PUT: permette di aggiornare una risorsa.
- DELETE: elimina una risorsa.

3 Metodologia ed implementazione

Per la risoluzione del problema, abbiamo implementato un client (client.c) che effettuera' la connessione e la richiesta HTTP, mentre il server (server.c) gestira' le connessioni e le richieste HTTP riferite ad un file di esempio (risorsa.html). in particolar modo, il server:

- gestisce con un mutex l'accesso alla risorsa html.
- e' non bloccante e utilizza epoll(): le connessioni dei client non vengono bloccate e il monitoraggio dei file descriptor avviene in kernel space, garantendo prestazioni maggiori.

3.1 Codice client

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
```

```c
#include <sys/socket.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 1024

void usage(const char *prog_name) {
    fprintf(stderr, "Usage: %s <host> <port> <METHOD> [content]\n",
     prog_name);
    fprintf(stderr, "Methods: GET, POST, PUT, DELETE\n");
    fprintf(stderr, "Example:\n");
    fprintf(stderr, "  %s 127.0.0.1 8080 GET\n", prog_name);
    fprintf(stderr, "  %s 127.0.0.1 8080 PUT '<h1>New Content</h1
    >'\n", prog_name);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[]) {
    if (argc < 4 || argc > 5) {
        usage(argv[0]);
    }

    const char *host = argv[1];
    int port = atoi(argv[2]);
    const char *method = argv[3];
    const char *content = (argc == 5) ? argv[4] : "";

    int sock;
    struct sockaddr_in serv_addr;
    char request[BUFFER_SIZE] = {0};
    char response[BUFFER_SIZE] = {0};

    int req_len = snprintf(request, BUFFER_SIZE,
        "%s / HTTP/1.1\r\n"
        "Host: %s:%d\r\n"
        "\r\n"
        "%s",
        method, host, port, content);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        return -1;
    }

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

    if (inet_pton(AF_INET, host, &serv_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(
    serv_addr)) < 0) {
        perror("Connection Failed");
        return -1;
    }

```

```
    printf("--- Sending Request ---\n%s\n-------------------------\n\
    n", request);
    send(sock, request, req_len, 0);

    printf("--- Server Response ---\n");
    int bytes_read;
    while ((bytes_read = read(sock, response, BUFFER_SIZE - 1)) >
    0) {
        response[bytes_read] = '\0';
        printf("%s", response);
    }
    printf("\n-------------------------\n");

    close(sock);
    return 0;
}
```

3.2 Codice server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/epoll.h>
#include <fcntl.h>
#include <errno.h>
#include <pthread.h>

#define PORT 8080
#define MAX_EVENTS 10
#define BUFFER_SIZE 1024
#define RESOURCE_FILE "risorsa.html"

pthread_mutex_t file_mutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct {
    int fd;
    char read_buffer[BUFFER_SIZE];
    int read_pos;
    char write_buffer[BUFFER_SIZE];
    int write_pos;
    int total_to_write;
} ConnectionState;

void set_nonblocking(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    if (flags == -1) {
        perror("fcntl F_GETFL");
        exit(EXIT_FAILURE);
    }
    if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl F_SETFL O_NONBLOCK");
        exit(EXIT_FAILURE);
    }
```

```
}

void close_connection(int epoll_fd, ConnectionState *state) {
    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, state->fd, NULL);
    close(state->fd);
    free(state);
    printf("Connection closed\n");
}

void prepare_response(ConnectionState *state, int status_code,
    const char *status_text, const char *body) {
    state->total_to_write = snprintf(state->write_buffer,
    BUFFER_SIZE,
        "HTTP/1.1 %d %s\r\n"
        "Server: Server C Paladino\r\n"
        "\r\n"
        "%s",
        status_code, status_text, body ? body : "");
    state->write_pos = 0;
}

void handle_request(ConnectionState *state) {
    char method[16], uri[256], version[16];
    sscanf(state->read_buffer, "%15s %255s %15s", method, uri,
    version);

    printf("Request: %s %s %s\n", method, uri, version);

    pthread_mutex_lock(&file_mutex);

    if (strcmp(method, "GET") == 0) {
        FILE *f = fopen(RESOURCE_FILE, "r");
        if (!f) {
            prepare_response(state, 404, "Not Found", "Resource not
     found.");
        } else {
            char file_buffer[BUFFER_SIZE] = {0};
            fread(file_buffer, 1, BUFFER_SIZE - 1, f);
            fclose(f);
            prepare_response(state, 200, "OK", file_buffer);
        }
    } else if (strcmp(method, "PUT") == 0 || strcmp(method, "POST")
     == 0) {
        const char *body_start = strstr(state->read_buffer, "\r\n\r
    \n");
        if (body_start) {
            body_start += 4;
            const char* mode = (strcmp(method, "PUT") == 0) ? "w" :
     "a";
            FILE *f = fopen(RESOURCE_FILE, mode);
            if(f) {
                fputs(body_start, f);
                fclose(f);
                int status_code = (strcmp(method, "PUT") == 0) ?
    201 : 200;
                const char* status_text = (strcmp(method, "PUT") ==
     0) ? "Created" : "OK";
```

```
86                  prepare_response(state, status_code, status_text, "
    Resource updated.");
87              } else {
88                  prepare_response(state, 500, "Internal Server Error
    ", "Could not write to resource.");
89              }
90          } else {
91              prepare_response(state, 400, "Bad Request", "Missing
    body for PUT/POST.");
92          }
93      } else if (strcmp(method, "DELETE") == 0) {
94          if (remove(RESOURCE_FILE) == 0) {
95              prepare_response(state, 204, "No Content", NULL);
96          } else {
97              prepare_response(state, 404, "Not Found", "Resource not
     found, cannot delete.");
98          }
99      } else {
100         prepare_response(state, 501, "Not Implemented", "Method not
     implemented.");
101     }
102
103     pthread_mutex_unlock(&file_mutex);
104 }
105
106 int main() {
107     int listen_fd, epoll_fd;
108     struct sockaddr_in server_addr;
109     struct epoll_event event, events[MAX_EVENTS];
110
111     listen_fd = socket(AF_INET, SOCK_STREAM, 0);
112     set_nonblocking(listen_fd);
113
114     memset(&server_addr, 0, sizeof(server_addr));
115     server_addr.sin_family = AF_INET;
116     server_addr.sin_addr.s_addr = htons(INADDR_ANY);
117     server_addr.sin_port = htons(PORT);
118
119     if (bind(listen_fd, (struct sockaddr *)&server_addr, sizeof(
    server_addr)) < 0) {
120         perror("bind");
121         exit(EXIT_FAILURE);
122     }
123
124     if (listen(listen_fd, SOMAXCONN) < 0) {
125         perror("listen");
126         exit(EXIT_FAILURE);
127     }
128
129     epoll_fd = epoll_create1(0);
130     event.events = EPOLLIN;
131     event.data.fd = listen_fd;
132     epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &event);
133
134     printf("Server listening on port %d\n", PORT);
135
136     while (1) {
```

```
        int n_events = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
        for (int i = 0; i < n_events; i++) {
            if (events[i].data.fd == listen_fd) {
                struct sockaddr_in client_addr;
                socklen_t client_len = sizeof(client_addr);
                int conn_fd = accept(listen_fd, (struct sockaddr *)&client_addr, &client_len);
                set_nonblocking(conn_fd);

                ConnectionState *state = (ConnectionState *)malloc(sizeof(ConnectionState));
                memset(state, 0, sizeof(ConnectionState));
                state->fd = conn_fd;

                event.events = EPOLLIN;
                event.data.ptr = state;
                epoll_ctl(epoll_fd, EPOLL_CTL_ADD, conn_fd, &event);
                printf("New connection accepted\n");
            } else {
                ConnectionState *state = (ConnectionState *)events[i].data.ptr;
                if (events[i].events & EPOLLIN) {
                    int bytes_read = read(state->fd, state->read_buffer + state->read_pos, BUFFER_SIZE - state->read_pos);
                    if (bytes_read <= 0) {
                        close_connection(epoll_fd, state);
                        continue;
                    }
                    state->read_pos += bytes_read;
                    if (strstr(state->read_buffer, "\r\n\r\n")) {
                        handle_request(state);
                        event.events = EPOLLOUT;
                        event.data.ptr = state;
                        epoll_ctl(epoll_fd, EPOLL_CTL_MOD, state->fd, &event);
                    }
                } else if (events[i].events & EPOLLOUT) {
                    int bytes_written = write(state->fd, state->write_buffer + state->write_pos, state->total_to_write - state->write_pos);
                    if (bytes_written < 0) {
                        if (errno != EAGAIN && errno != EWOULDBLOCK) {
                            close_connection(epoll_fd, state);
                        }
                        continue;
                    }
                    state->write_pos += bytes_written;
                    if (state->write_pos >= state->total_to_write) {
                        close_connection(epoll_fd, state);
                    }
                }
            }
        }
```

```
183     }
184
185     close(listen_fd);
186     pthread_mutex_destroy(&file_mutex);
187     return 0;
188 }
```

4 Presentazione dei risultati

Una volta lanciato il programma server e un terminale client, abbiamo verificato che:

- utilizzando il metodo GET il client riceve il contenuto della pagina html.
- utilizzando il metodo PUT o POST e' possibile da parte del client aggiungere contenuto alla risorsa.
- utilizzando il metodo DELETE si puo' eliminare il file risorsa.

Abbiamo inoltre verificato, oltre alle informazioni del protocollo TCP, che Wireshark riconosce automaticamente che il traffico generato riguarda HTTP con i relativi codici di stato.

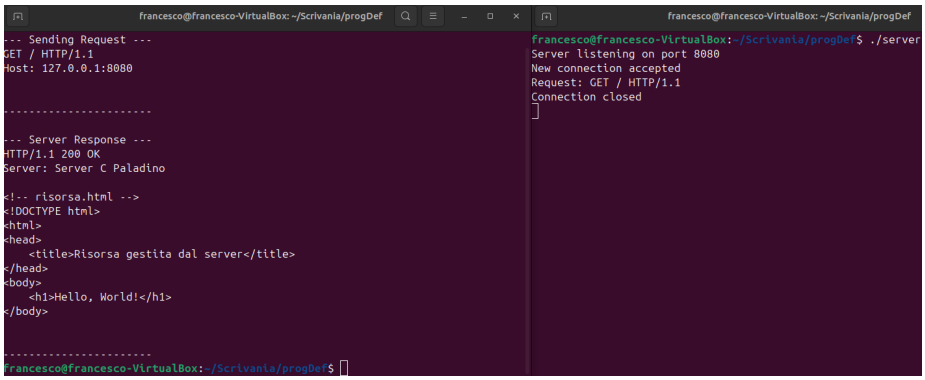

Immagine 1: terminale client che riceve la pagina web col comando GET (a sinistra) e il terminale server (a destra)

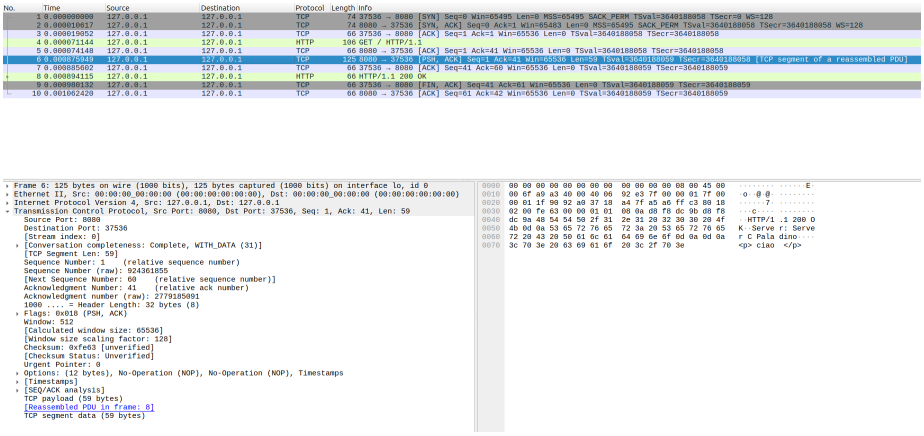

Immagine 2: traffico catturato con Wireshark durante l'esecuzione con successo del comando GET

4.1 Test di carico

Grazie all'utilizzo della funzione concurrent.futures.ThreadPoolExecutor di Python, abbiamo eseguito dei test di carico nei quali (per quanto riguarda le richieste GET):

- sono state eseguite 10000 richieste da 1000 thread in 65 secondi.
- sono state eseguite 10000 richieste da 2000 thread in 61 secondi.
- sono state eseguite 100000 richieste da 2000 thread in 763 secondi.

Abbiamo inoltre notato con Wireshark che, dato l'elevato numero di connessioni, il protocollo TCP riutilizza (lato client) le porte TCP occupate da una connessione nello stato di TIME_WAIT.

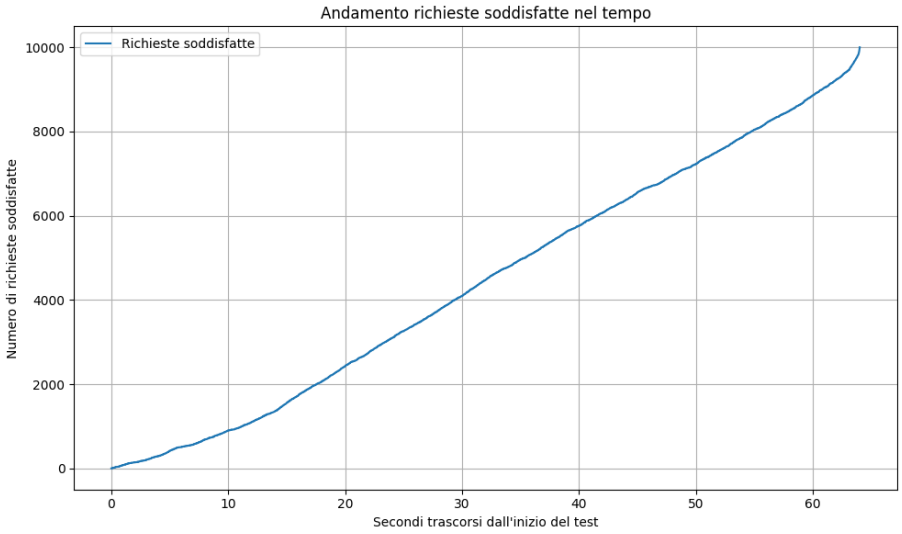

Immagine 3: risultati ottenuti con 1000 thread e 10000 richieste

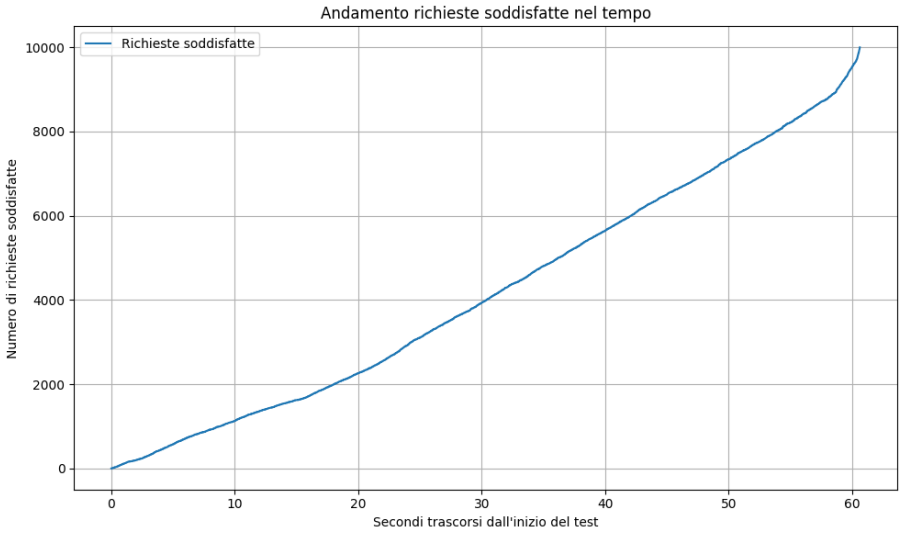

Immagine 4: risultati ottenuti con 2000 thread e 10000 richieste

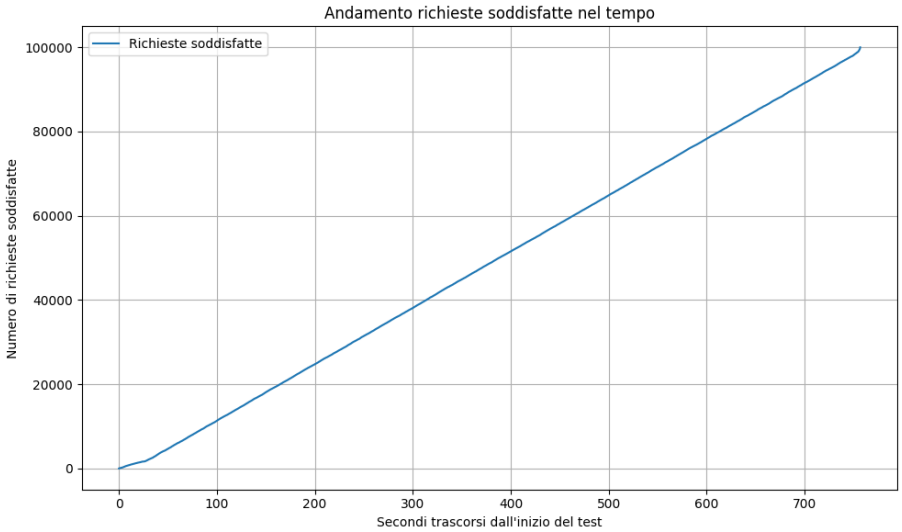

Immagine 5: risultati ottenuti con 2000 thread e 100000 richieste

5 Conclusioni e sviluppi futuri

Abbiamo visto come costruire un server HTTP con le principali richieste GET, POST, PUT e DELETE. Abbiamo inoltre utilizzato un mutex per garantire il corretto accesso ad una pagina html e garantito ottime prestazioni grazie ad epoll(). Tuttavia e' possibile applicare delle migliorie come:

- l'utilizzo di HTTPS, una versione del protocollo originario che introduce una comunicazione sicura grazie all'utilizzo della crittografia. Non sara' possibile quindi leggere da un soggetto terzo il traffico HTTPS tra client e server.
- integrare l'utilizzo del DNS per inserire il nome di dominio del sito e non l'indirizzo IP.