Relazione Sistemi Operativi

Mattia Bortolaso

15 luglio 2025

Indice

1 Introduzione

Il progetto consiste in un sistema client–server per il calcolo dell'hash crittografico SHA-256 di file arbitrari, realizzato in linguaggio C come esercizio per il corso di Sistemi Operativi. Il sistema utilizza meccanismi di comunicazione *FIFO* (pipe con nome) su ambiente POSIX per lo scambio di messaggi tra un processo server e uno o più processi client.

In sintesi, il client invia al server la richiesta di calcolare l'impronta (digest) SHA-256 di uno o più file specificati. Il server, eseguito in parallelo, riceve le richieste attraverso una FIFO pubblica di ingresso e calcola gli hash richiesti, sfruttando la concorrenza tramite thread e memorizzando i risultati calcolati in una cache interna. Per ogni richiesta, il server invia il risultato (la stringa esadecimale SHA-256 del file) al client tramite un'apposita FIFO di risposta dedicata a quella richiesta. Il client raccoglie tutti gli hash risultanti e li presenta ordinati per dimensione del file.

2 Struttura Generale del Sistema

Il sistema è composto da due programmi principali: un **client** (implementato in `collector_client.c`) e un **server** (implementato in `server.c`). Ciascun componente svolge un ruolo specifico nella soluzione: il client prepara le richieste di hash per i file indicati dall'utente e raccoglie i risultati, mentre il server riceve ed elabora tali richieste in modo concorrente.

2.1 Il Client `collector_client.c`

Il programma client ha il compito di raccogliere l'input (l'elenco di file di cui calcolare l'hash) e di inviare una richiesta per ciascun file al server attraverso la FIFO di ingresso comune. All'avvio, il client crea una struttura dati per memorizzare le richieste (ad esempio un array di `RequestEntry`, contenente percorso del file, nome della FIFO di risposta, hash calcolato e dimensione del file). Per ogni file specificato da riga di comando, il client genera un nome univoco per una FIFO di risposta (tipicamente incorporando il PID del processo e un indice progressivo) e crea effettivamente tale FIFO tramite la chiamata di sistema `mkfifo`.

Una volta pronta la FIFO dedicata, il client invia una richiesta al server aprendo la FIFO di ingresso del server (definita come `/tmp/fifo_in`) in scrittura e scrivendo al suo interno un messaggio contenente il percorso del file e il nome della FIFO client, separati da un delimitatore predefinito (nel progetto si utilizza la stringa `::` come separatore tra nome file e nome FIFO). Il client ripete questa procedura per tutti i file richiesti, creando per ognuno una propria FIFO di risposta e inviando la relativa richiesta al server.

Dopo aver inviato tutte le richieste, il client si mette in attesa dei risultati. Per ogni FIFO di risposta precedentemente creata, il client apre la FIFO in lettura e legge la stringa contenente l'hash restituito dal server (64 caratteri esadecimali rappresentanti il digest SHA-256). Ottenuto l'hash (o un messaggio di errore in caso di problemi), il client chiude e rimuove la FIFO di risposta temporanea. Una volta raccolti tutti i risultati, il client li ordina in base alla dimensione del file

originale (dal più piccolo al più grande, come specificato) utilizzando un semplice ordinamento in memoria, e infine stampa a schermo i percorsi dei file con i rispettivi size in byte e hash SHA-256, in ordine crescente di dimensione.

2.2 Il Server `server.c`

Il programma server è responsabile di gestire le richieste di hash in arrivo in maniera concorrente ed efficiente. All'avvio, il server crea la FIFO pubblica di ingresso (il path `/tmp/fifo_in`) da cui riceverà le richieste dai client. Il server quindi rimane in ascolto su tale FIFO: nel codice ciò è realizzato tramite un *thread* dedicato (`dispatcher_thread`) che esegue un ciclo di lettura bloccante sulla FIFO di ingresso e, ogni volta che riceve uno o più messaggi di richiesta, li scompone e li inserisce in una coda interna di richieste da elaborare.

Per servire le richieste in parallelo, il server crea dinamicamente thread *worker* che si occupano di calcolare l'hash dei file richiesti. Il numero massimo di thread attivi è limitato (ad esempio a 4 thread concorrenti) per controllare le risorse; se tutte le unità di esecuzione sono occupate, le nuove richieste rimangono accodate finché un thread non diventa disponibile. La coda delle richieste in attesa non segue un semplice ordine FIFO di arrivo: il sistema infatti inserisce le richieste in modo ordinato in base alla dimensione del file, così da dare priorità ai file più piccoli (riducendo il tempo di risposta medio per le operazioni brevi).

Il server implementa anche un meccanismo di **cache** dei risultati già calcolati e una struttura per la gestione delle richieste *in corso*. In questo modo, se il server riceve più richieste per lo stesso file, soltanto un thread effettuerà il calcolo dell'hash, mentre gli altri potranno attendere il risultato (evitando duplicazioni di lavoro) oppure ottenere immediatamente l'esito dalla cache se l'hash di quel file era già stato calcolato in precedenza. Al termine dell'elaborazione di ciascuna richiesta, il thread worker scrive il valore hash calcolato nella FIFO di risposta specificata (quella creata dal client per quella richiesta), permettendo così al client richiedente di leggere il risultato. L'architettura server così descritta garantisce che più file possano essere processati in parallelo, ottimizzando le prestazioni grazie al multi-threading e al riutilizzo dei risultati già disponibili.

3 Approfondimenti Tecnici

3.1 Uso delle FIFO e Nomi Temporanei per le Richieste

La comunicazione tra client e server avviene tramite speciali file FIFO (First In First Out) creati nel file system. Il server apre all'avvio una FIFO pubblica nota (`/tmp/fifo_in`) su cui i client possono scrivere le loro richieste. Ogni richiesta inviata attraverso `fifo_in` contiene sia il percorso del file da elaborare sia il nome di una FIFO privata che il client ha creato per ricevere la risposta.

Per ogni file richiesto, il client genera infatti un nuovo endpoint di risposta: ad esempio, il nome può essere del tipo `/tmp/fifo_client_<PID>_<N>`, combinando il proprio PID e un indice. Utilizzando `mkfifo`, il client crea fisicamente questa FIFO e la passa al server all'interno del messaggio di richiesta. In questo modo il server,

una volta calcolato l'hash, potrà aprire la FIFO specificata ed inviare il risultato unicamente a quel client. Dopo aver letto la risposta, il client rimuove (unlink) la FIFO temporanea. Questo approccio con FIFO dedicate ad ogni risposta garantisce che le risposte dei diversi calcoli non si mescolino e permette al client di gestire facilmente più richieste concorrenti (ciascuna avrà un proprio canale di ritorno). Si noti che il server apre la FIFO di ingresso in modalità `O_RDWR` (lettura/scrittura) per evitare di rimanere bloccato in assenza momentanea di scrittori.

3.2 Calcolo dell'hash SHA-256 con OpenSSL

Il calcolo dell'impronta SHA-256 dei file è implementato utilizzando la libreria OpenSSL, che fornisce funzioni ottimizzate per l'hash. In particolare, la funzione ausiliaria `digest_file` del server inizializza un contesto SHA-256 (`SHA256_Init`), legge il contenuto del file in blocchi (ad esempio 1024 byte alla volta usando `read`) aggiornando iterativamente il digest (`SHA256_Update` per ogni blocco), e infine completa il calcolo ottenendo i 32 byte di digest binario (`SHA256_Final`). Questo valore binario viene poi convertito in formato esadecimale leggibile: ogni byte produce due cifre esadecimali, per un totale di 64 caratteri (più il terminatore di stringa). Nel codice, questa conversione viene effettuata da ogni thread worker dopo `digest_file`, utilizzando `sprintf` in un ciclo per formattare ciascun byte in due caratteri. Appoggiarsi a OpenSSL per il calcolo dell'hash assicura correttezza e buone prestazioni, delegando la parte crittografica ad una libreria collaudata.

3.3 Ordinamento delle Richieste per Dimensione del File

Una caratteristica rilevante dell'implementazione è l'ordinamento delle richieste in base alla dimensione dei file da elaborare. Ciò è realizzato nel server all'interno della funzione `enqueue_request`: quando il thread dispatcher inserisce una nuova richiesta nella coda, questa viene collocata in posizione tale che la coda rimanga ordinata in ordine crescente di *filesize*. In pratica, l'array che implementa la coda viene scansionato da fondo a cima finché non si trova la posizione corretta (spostando gli elementi più grandi verso destra per fare spazio). Se due file hanno esattamente la stessa dimensione, si applica un secondo criterio lessicografico sul nome del file (o meglio, sull'intera stringa di richiesta) per garantire un ordine deterministico.

Ordinando le richieste in questo modo, il server privilegia l'esecuzione dei task più brevi: un file di piccole dimensioni presente in coda verrà assegnato a un thread prima di uno molto grande arrivato in precedenza. Questo può migliorare il tempo di risposta medio e l'utilizzo del sistema (evitando che richieste leggere rimangano bloccate dietro elaborazioni lunghe). D'altro canto, si introduce una deviazione dall'ordine di arrivo (non strettamente FIFO), ma in un contesto di calcolo di hash questo è accettabile e mirato a ottimizzare la reattività.

3.4 Gestione della Concorrenza: Thread, Mutex e Variabili di Condizione

Il server impiega meccanismi di sincronizzazione per gestire la concorrenza tra thread. Anzitutto, il numero di thread worker simultanei è limitato da una costante (ad esempio `MAX_THREADS = 4`); il thread dispatcher controlla questo limite mantenendo un contatore atomico (`active_threads`) protetto da mutex. Quando arriva una nuova richiesta, se ci sono già `MAX_THREADS` lavoratori attivi, il dispatcher attende su una `condition variable` (`thread_available`) che viene notificata quando uno dei thread termina, liberando così uno slot. Solo allora il dispatcher può avviare un altro thread per servire la richiesta in coda. I thread worker vengono creati con `pthread_create` e subito `detach`-ati (in modo da liberare automaticamente le risorse al termine).

Per proteggere le strutture condivise, il server utilizza diversi **mutex**: uno per la coda delle richieste (`queue_mutex`), uno per la cache e la lista di richieste in corso (`cache_mutex`), e uno per il contatore di thread attivi (`thread_count_mutex`). Ad esempio, ogni operazione di inserimento o estrazione dalla coda viene effettuata bloccando `queue_mutex`, così da evitare condizioni di gara quando più thread (o il dispatcher stesso) accedono alla coda contemporaneamente. In maniera analoga, `cache_mutex` viene usato per serializzare l'accesso sia alla cache dei risultati già disponibili sia alla struttura che traccia le elaborazioni in corso (descritta di seguito). Le variabili di condizione permettono di sincronizzare efficacemente i thread: oltre a `thread_available` già citata, la `pthread_cond_t queue_not_empty` viene predisposta nel caso in cui si volesse far attendere i worker su una coda vuota (pattern tipico di produttore/consumatore), anche se nel design attuale i thread worker sono lanciati direttamente dal dispatcher.

3.5 Caching dei Risultati e Gestione delle Richieste in Corso

Per evitare ricalcoli ripetuti dello stesso hash e gestire situazioni di richieste duplicate concorrenti, il server implementa due meccanismi complementari: una cache dei risultati già calcolati e una lista delle elaborazioni *in corso*. La cache è una semplice struttura dati (array di `CacheEntry`) che memorizza il percorso del file e la stringa hash calcolata. Ogni volta che sta per essere elaborata una richiesta, il server verifica innanzitutto se il file è presente in cache: in tal caso restituisce immediatamente l'hash memorizzato senza avviare un nuovo calcolo. La cache è protetta da `cache_mutex` per assicurare consistenza in accesso concorrente, e ha una dimensione massima prefissata (nel codice `MAX_CACHE_SIZE`) oltre la quale non vengono aggiunti nuovi elementi (in una possibile estensione si potrebbe usare una politica LRU per rimuovere i meno recenti).

Se l'hash richiesto non è in cache, il server deve procedere a calcolarlo, ma potrebbe accadere che un altro thread stia *già* calcolando l'hash dello stesso file in parallelo. Per gestire tale scenario, il server mantiene una struttura denominata `in_progress_list`: una lista (o array) di elementi `InProgressEntry` ciascuno dei quali traccia un file attualmente in fase di hashing. Quando un thread worker inizia il calcolo di un file, aggiunge una nuova entry in questa lista (impostando il flag `done = 0` e `wait_count = 0`) e inizializza una `pthread_cond_t` locale per quel fi-

le. Se un secondo thread (causato da una richiesta duplicata) trova nella lista un elemento con lo stesso filepath e `done == 0`, invece di calcolare di nuovo l'hash incrementa il `wait_count` di quella entry ed attende su quella condition variable che il calcolo venga completato dal primo thread. Quando il primo thread termina il calcolo, imposta `done = 1`, copia l'hash calcolato nella struttura `InProgressEntry` e risveglia tutti i thread in attesa con `pthread_cond_broadcast`. A quel punto i thread secondari possono riprendere, leggere l'hash pronto dalla struttura condivisa ed evitare completamente il calcolo.

Dopo aver notificato eventuali waiter, il thread primario (o l'ultimo thread che era in attesa) pulisce la entry dalla lista `in_progress_list` (deallocando la condition variable e rimuovendo l'elemento). Anche l'inserimento di nuove entry in-progress è protetto dal `cache_mutex` (usato come lock globale per cache e in-progress data la stretta relazione tra le due strutture). Nel codice è presente anche una logica di garbage collection: se la lista in-progress dovesse saturarsi, vengono eliminate le voci già completate (`done = 1`) per far spazio a nuove richieste.

Grazie a questi meccanismi di cache e sincronizzazione, il server evita sia di ricalcolare hash di file già trattati in passato, sia di calcolare più volte in parallelo lo stesso hash quando richiesto simultaneamente. Ciò migliora sensibilmente l'efficienza in scenari con file ripetuti, riducendo il carico computazionale complessivo.

4 Analisi di Alcuni Frammenti di Codice

Per chiarire ulteriormente l'implementazione, esaminiamo alcuni estratti significativi del codice sorgente, accompagnati dai commenti originali che ne spiegano la logica.

4.1 Preparazione delle Richieste nel Client

Nel codice del client (`collector_client.c`), la preparazione delle richieste avviene all'interno della funzione `main`. Il frammento seguente mostra come, per ciascun file passato come argomento, il client allestisca la struttura dati e la FIFO necessarie e invii poi la richiesta al server. In particolare, si può notare la generazione del nome FIFO unico per la risposta, la creazione della FIFO con `mkfifo`, l'uso della chiamata di sistema `open` per aprire la FIFO di ingresso del server e la scrittura del messaggio formattato contenente percorso del file e nome FIFO:

```
// Prepara richieste
for (int i = 0; i < num_files; i++) {
    strncpy(requests[i].filepath, argv[i + 1], sizeof(requests[i].filepath));
    requests[i].filepath[sizeof(requests[i].filepath) - 1] = '\0';

    snprintf(requests[i].fifo_path, sizeof(requests[i].fifo_path), "/tmp/fifo_c
    if (mkfifo(requests[i].fifo_path, 0666) < 0) {
        perror("mkfifo");
        return 1;
    }
```

```
        struct stat st;
        if (stat(requests[i].filepath, &st) < 0) {
            perror("stat");
            return 1;
        }
        requests[i].filesize = st.st_size;

        // Invia richiesta al server
        int fd_out = open(FIFO_IN, O_WRONLY);
        if (fd_out < 0) {
            perror("open FIFO_IN");
            return 1;
        }

        char message[2048];
        snprintf(message, sizeof(message), "%s::%s", requests[i].filepath, requests
        write(fd_out, message, strlen(message) + 1);
        close(fd_out);
    }
```

Come mostrato, dopo aver scritto il messaggio nella FIFO del server, il client chiude il descriptor (`fd_out`) e prosegue con eventuali altre richieste. La lettura delle risposte (non riportata qui) avviene in un secondo ciclo, nel quale il client apre ciascuna FIFO di risposta e legge l'hash calcolato.

4.2 Funzione `digest_file`

La funzione `digest_file` del server incapsula i dettagli del calcolo dell'hash SHA-256 di un file. Di seguito ne riportiamo l'implementazione in `server.c`. Si può notare l'uso delle routine di OpenSSL (`SHA256_Init`, `SHA256_Update`, `SHA256_Final`) e delle chiamate di sistema POSIX per l'accesso al file (`open`, `read`, `close`). Il codice è documentato con commenti che spiegano passo passo l'operazione:

```
/**
 * Calcola l'hash SHA-256 del file specificato.
 * @param filename Percorso del file di cui calcolare l'hash.
 * @param hash Buffer di 32 byte dove verrà scritto il digest binario risultante.
 */
void digest_file(const char *filename, uint8_t *hash) {
    SHA256_CTX ctx;
    SHA256_Init(&ctx);

    // Apre il file in sola lettura
    int file = open(filename, O_RDONLY);
    if (file == -1) {
        perror("open file");
        return;  // Se il file non si apre, esce (errore gestito dal chiamante)
```

```
    }

    // Legge il file a blocchi e aggiorna il contesto SHA256
    unsigned char buffer[1024];
    ssize_t bytes;
    while ((bytes = read(file, buffer, sizeof(buffer))) > 0) {
        SHA256_Update(&ctx, buffer, bytes);
    }
    close(file);

    // Completa il calcolo dell'hash (digest finale)
    SHA256_Final(hash, &ctx);
}
```

Questa funzione viene invocata dai thread worker per effettuare materialmente il calcolo dell'hash. In caso di errore nell'apertura del file, viene semplicemente stampato un messaggio di errore e restituito il controllo al chiamante (il quale gestirà la situazione, ad esempio inviando un errore al client).

4.3 Funzione `enqueue_request`

Come descritto, il server inserisce le richieste in una coda ordinata per dimensione. La funzione `enqueue_request`, riportata di seguito, realizza questa logica. Si noti l'utilizzo di `pthread_mutex_lock` e `pthread_mutex_unlock` per proteggere l'accesso concorrente alla struttura della coda, e il ciclo `while` che effettua lo spostamento degli elementi più grandi verso la fine dell'array:

```
void enqueue_request(const char* request_str, off_t filesize) {
    pthread_mutex_lock(&queue_mutex);

    if (queue_size >= MAX_QUEUE) {
        fprintf(stderr, "Coda delle richieste piena. Richiesta scartata.\n");
        pthread_mutex_unlock(&queue_mutex);
        return;
    }

    int i = queue_size;
    while (i > 0) {
        if (request_queue[i - 1].filesize > filesize) {
            // Sposta a destra se il file precedente è più grande
            request_queue[i] = request_queue[i - 1];
            i--;
        } else if (request_queue[i - 1].filesize == filesize &&
                   strcmp(request_queue[i - 1].request_str, request_str) > 0) {
            // Sposta a destra se le dimensioni sono uguali e ordine alfabetico mag
            request_queue[i] = request_queue[i - 1];
            i--;
```

```
        } else {
            break;
        }
    }

    strncpy(request_queue[i].request_str, request_str, sizeof(request_queue[i].requ
    request_queue[i].request_str[sizeof(request_queue[i].request_str) - 1] = '\0';
    request_queue[i].filesize = filesize;
    queue_size++;
    printf("[DEBUG] Richiesta accodata: %s (size: %ld). Coda attuale: %d\n", reques

    // DEBUG opzionale:
    // printf("Inserita richiesta: %s (size: %ld) in posizione %d\n", request_str,

    pthread_cond_signal(&queue_not_empty);
    pthread_mutex_unlock(&queue_mutex);
}
```

Nella porzione di codice sopra, dopo aver aggiunto l'elemento nella posizione corretta, si segnala con `pthread_cond_signal` che la coda non è più vuota (`queue_not_empty`) e si rilascia il mutex. In caso di coda piena, la richiesta viene scartata (stampando un messaggio di errore).

4.4 Funzione `handle_request`

Infine, riportiamo il codice della funzione chiave `handle_request`, eseguita da ogni thread worker per gestire una singola richiesta. Questo frammento è più esteso, ma nei commenti si possono individuare chiaramente le diverse fasi:

- parsing della stringa di richiesta per separare il percorso del file dal nome della FIFO del client;
- verifica nella cache dei risultati già pronti;
- gestione della possibile concorrenza su uno stesso file tramite la struttura `in_progress_list` (attesa su `pthread_cond_wait` se un altro thread sta già calcolando lo stesso hash);
- calcolo vero e proprio dell'hash tramite `digest_file` (fuori dalla sezione critica);
- inserimento del risultato in cache, aggiornamento dello stato dell'operazione in corso e risveglio di eventuali thread in attesa;
- invio dell'hash calcolato al client attraverso la FIFO di risposta e pulizia finale (liberazione risorse e decremento del contatore di thread attivi).

Riportiamo di seguito l'implementazione integrale della funzione, che ingloba tutti questi aspetti:

```
void* handle_request(void* arg) {
    char* input = (char*)arg;
    // Divide la stringa di input in "filepath" e "fifo_path" usando il separatore
    char* sep = strstr(input, "::");
    if (!sep) {
        fprintf(stderr, "Richiesta malformata: %s\n", input);
        free(input);
        // Decrementa contatore thread attivi e segnala al dispatcher la disponibil
        pthread_mutex_lock(&thread_count_mutex);
        active_threads--;
        pthread_cond_signal(&thread_available);
        pthread_mutex_unlock(&thread_count_mutex);

        return NULL;
    }
    *sep = '\0';                    // termina la stringa file path
    char* filepath = input;
    char* fifo_path = sep + 2;      // inizio del nome FIFO client

    char hash_string[65] = "";      // buffer per hash esadecimale (64 caratteri +
    int need_compute = 0;

    // Sezione critica: verifica se risultato già in cache o se file in elaborazion
    pthread_mutex_lock(&cache_mutex);
    // 1. Controlla la cache per vedere se l'hash è già disponibile
    for (int i = 0; i < cache_size; ++i) {
        if (strcmp(cache[i].filepath, filepath) == 0) {
            strcpy(hash_string, cache[i].hash_string);
            need_compute = 0;
            break;
        }
    }
    if (hash_string[0] == '\0') {
        // Non trovato in cache
        need_compute = 1;
    }

...
```

Questo frammento mostra in azione tutti i meccanismi discussi in precedenza: dall'uso della cache e della lista in-progress con le relative condizioni di attesa, fino alla scrittura del risultato sulla FIFO del client. I commenti nel codice aiutano a seguire la logica e a comprendere come il server gestisca in modo sicuro le situazioni di concorrenza.

5 Testing e Validazione

La fase di collaudo del progetto ha previsto l'esecuzione di test automatici attraverso uno script dedicato (`run_tests.sh`). Di seguito sono sintetizzati i principali casi di test effettuati e i relativi esiti:

- **Test 1 – hash singolo:** verifica il corretto calcolo dell'hash per un singolo file di nome `f_small`. Esito postivo.
- **Test 2 – ordinamento:** verifica che i risultati stampati siano ordinati per dimensione (da `f_small` a `f_1k` a `f_2k`). Esito postivo.
- **Test 3 – file inesistente:** verifica che venga segnalato un errore in caso di file mancante. Esito positivo.
- **Test 4 – cache:** verifica che il secondo calcolo dello stesso file sia più rapido (ad esempio: 3 ms vs 1 ms). Esito postivo.
- **Test 5 – stress concorrente:** verifica la robustezza sotto carico con 8 client concorrenti. Esito postivo.

Tutti i test sono stati superati con successo. Inoltre, al termine dell'esecuzione, lo script `run_tests.sh` provvede anche alla pulizia di eventuali FIFO residue e alla rimozione automatica dei file temporanei, garantendo che il sistema non lasci artefatti nel file system dopo i test.

6 Conclusioni

In conclusione, il sistema realizzato dimostra un utilizzo efficace dei meccanismi offerti dal sistema operativo per ottenere un servizio di calcolo distribuito dell'hash. Dal punto di vista dell'**efficienza**, l'uso di thread concorrenti permette di sfruttare le risorse multicore elaborando più file in parallelo, mentre l'ordinamento delle richieste per dimensione contribuisce a ridurre i tempi di attesa per i task più brevi. La presenza di una cache dei risultati evita ricalcoli ridondanti, con un notevole beneficio in scenari ripetitivi (ad esempio, richieste multiple dello stesso file).

Per quanto riguarda la **scalabilità**, l'architettura a client multipli e un server multi-thread riesce a gestire contemporaneamente molteplici richieste mantenendo buone prestazioni. Il limite imposto sul numero di thread concorrenti (`MAX_THREADS`) previene un uso eccessivo di risorse in caso di carichi molto elevati, fungendo da meccanismo di controllo del flusso. Allo stesso tempo, questo potrebbe costituire un collo di bottiglia se l'hardware a disposizione consentirebbe un grado di parallelismo maggiore; tuttavia, il valore di `MAX_THREADS` può essere adeguato in base alla piattaforma. Anche la dimensione massima della cache e della coda di richieste sono parametri che andrebbero tarati per gestire volumi di richieste crescenti.

In sintesi, il progetto raggiunge l'obiettivo prefissato fornendo un servizio di calcolo di hash SHA-256 in modo concorrente e ottimizzato. Le tecniche implementate (coda prioritaria, thread pool limitato, caching e sincronizzazioni mirate) garantiscono una buona efficienza del sistema e suggeriscono che esso può scalare a un uso

realistico in ambiente monomacchina, compatibilmente con le risorse di calcolo e memoria disponibili. Eventuali estensioni future potrebbero riguardare l'aggiunta di politiche di rimpiazzo per la cache o la distribuzione del carico su più server, ma già nella forma attuale la soluzione si mostra solida ed efficace.