# Servizio Client-Server SHA-256 con Cache e Coda Prioritaria

Michele Pasetto

## 1 Introduzione

Il progetto si prospetta l'implementazione di un servizio client-server per la computazione dell'hash SHA-256 di un file. L'architettura si basa su comunicazione tramite FIFO, gestione concorrente delle richieste mediante thread e cache condivisa, per ottimizzare le prestazioni ed una coda ordinata per dimensione del file.

L'obiettivo principale è dimostrare l'utilizzo efficace di:

- **Comunicazione interprocesso** tramite FIFO
- **Sincronizzazione** con mutex e variabili di condizione
- **Thread** per gestione concorrente
- **Cache** per evitare calcoli ridondanti
- **Schedulazione prioritaria** basata sulla dimensione del file

## 2 Contenuto dei file

La struttura delle cartelle è la seguente:

```
.
+-- CMakeLists.txt
+-- README.md
+-- dist
+-- include
|   +-- cache.h
|   +-- common.h
|   +-- queue.h
|   +-- sha256_utils.h
+-- relazione.md
+-- run_test.sh
+-- script
|   +-- lib.sh
+-- src
|   +-- cache.c
|   +-- client.c
|   +-- queue.c
|   +-- server.c
```

```
|   +-- sha256_utils.c
+-- test
    +-- emptyfile.txt
    +-- password.txt
    +-- testfile.txt
```

| Path | Descrizione |
|---|---|
| `dist` | Viene usata per inserire i compilati |
| `include` | Contiene gli header files (.h) con definizioni di strutture e prototipi |
| `script` | Contiene script bash utili per la compilazione e il testing |
| `src` | Contiene i codici sorgente (.c) del progetto |
| `test` | Contiene i file di test usati per debug e unit tests |

## 2.1 Descrizione dei file principali

### 2.1.1 File sorgente (src/)

- **client.c**: Implementa il client che invia richieste di hash al server tramite FIFO
- **server.c**: Implementa il server multi-thread che gestisce le richieste
- **queue.c**: Implementa la coda prioritaria thread-safe ordinata per dimensione file
- **cache.c**: Implementa la cache condivisa con sincronizzazione per evitare calcoli duplicati
- **sha256_utils.c**: Funzioni di utilità per calcolo SHA-256 usando OpenSSL

### 2.1.2 File header (include/)

- **common.h**: Definizioni comuni (strutture dati, costanti, macro) usate da client e server
- **queue.h**: Interfaccia della coda prioritaria
- **cache.h**: Interfaccia della cache condivisa
- **sha256_utils.h**: Prototipo per calcolo SHA-256

# 3 Requisiti

## 3.1 Librerie necessarie

- **OpenSSL** (libssl-dev): per il calcolo dell'hash SHA-256
- **pthread**: per gestione thread pool e sincronizzazione
- **CMake** ($\geq$ 3.10): per compilazione automatica

### 3.2 Installazione dipendenze

```
# Ubuntu/Debian
sudo apt-get install libssl-dev cmake build-essential
```

## 4 Compilazione

Il progetto utilizza CMake per la compilazione automatica:

```
mkdir -p dist
cd dist
cmake ..
make
```

Questo produce due eseguibili:

- `dist/server`: il server multi-thread
- `dist/client`: il client

## 5 Esecuzione

### 5.1 Avvio del server

Il server deve essere avviato prima dei client:

```
$ cd dist
$ ./server
>
[SERVER] Listening on /tmp/sha256_req_fifo
```

### 5.2 Esecuzione del client

#### 5.2.1 Richiesta di hash per un file

```
$ ./client path/to/file.txt
>
SHA-256:
   a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
```

#### 5.2.2 Query della cache

```
./client CACHE?
```

Il server stamperà su stdout il contenuto corrente della cache.

# 6 Testing

## 6.1 Unit tests automatici

Subito dopo aver iniziato a sviluppare questo progetto, mi sono accorto della lentezza di sviluppo, soprattutto per via della necessità di avviare manualmente il server e poi il client per ogni test. Per questo motivo ho creato uno script bash `run_test.sh` che automatizza l'intero processo di compilazione, avvio del server, esecuzione dei test e verifica dei risultati usando la libreria sha256sum. Questo mi ha permesso di velocizzare notevolmente il ciclo di sviluppo e debugging, permetterndomi anche di usare più file di test in modo semplice.

Lo script `run_test.sh` esegue automaticamente:

1. Compilazione dei file sorgenti
2. Avvio del server in background
3. Esecuzione dei test del client usando i file nella cartella `/test`
4. Verifica correttezza hash tramite confronto con `sha256sum`
5. Terminazione del server

Esecuzione:

```
$ ./run_test.sh test/testfile.txt
>
[*] CMake building
-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to:
   /home/ginkgo/VR495361_Pasetto_Michele_sha256_service/dist
[OK] dist completed
[*] Starting server

[i] Server started with PID 7384
[SERVER] Listening on /tmp/sha256_req_fifo
[*] Running client tests for multiple files

[i] Submitting file to server: test/testfile.txt
    SHA-256:
       a665a45920422f9d417e4867efdc4fb8a04a1f3fff1fa07e998e86f7f7a27ae3
[OK] Digest matches CLI
[OK] All tests completed successfully
```

## 6.2 Test di concorrenza

Per testare richieste concorrenti:

```
# In terminali separati dopo aver avviato il server
./dist/client test/testfile.txt &
./dist/client test/password.txt &
./dist/client test/emptyfile.txt &
```

Il sistema gestisce correttamente:

- **Richieste concorrenti** per file diversi (parallelismo tramite thread pool)
- **Richieste duplicate** per lo stesso file (sincronizzazione tramite cache)

# 7 Dettagli implementativi

## 7.1 Architettura del sistema

### 7.1.1 Thread Pool (server.c)

Un pool di **4 thread worker** sempre attivi permette ad ogni thread di estrarre la richieste dalla coda prioritaria.

### 7.1.2 Coda prioritaria (queue.c)

La coda è una lista ordinata per dimensione del file, dando priorità ai lavori "leggeri", ovvero i file di dimensioni ridotte. Un mutex e una condition variable gestiscono i thread. `queue_push()` inserisce il job al posto giusto, `queue_pop()` aspetta finché c'è qualcosa e poi lo consegna al primo thread libero.

### 7.1.3 Cache condivisa (cache.c)

- Lista concatenata di entry con **mutex e condition variable per entry**
- `cache_lookup_or_insert()`: cerca o crea entry atomicamente
- `cache_set_digest()`: aggiorna risultato e notifica thread in attesa
- Permette a thread multipli di attendere lo stesso calcolo

### 7.1.4 Comunicazione FIFO

- **FIFO request**: `/tmp/sha256_req_fifo` (globale, write-only per client)
- **FIFO response**: `/tmp/sha256_resp_<PID>_fifo` (per client, create dal client stesso)

### 7.1.5 Calcolo SHA-256 (sha256_utils.c)

- Usa API OpenSSL: `SHA256_Init()`, `SHA256_Update()`, `SHA256_Final()`
- Lettura file a blocchi di 4096 byte
- Conversione digest binario (32 byte) in stringa hex (64 caratteri)

### 7.2 Gestione errori

- Controllo apertura file e FIFO
- Verifica lettura/scrittura completa
- Log errori
- Status code nelle risposte: 0=success, 1=error, 3=cache hit

## 8 Note

### 8.1 Scelte progettuali

- **FIFO in /tmp**: facilita l'eliminazione di file temporanei generati durante l'esecuzione
- **Client rimuove FIFO risposta**: evita accumulo di file temporanei
- **Lettura file a blocchi**: efficiente per file grandi
- **Thread pool fisso**: evita overhead creazione/distruzione thread
- **Cache con mutex per ogni entry**: massimizza parallelismo tra calcoli diversi

### 8.2 Limitazioni note

- Cache illimitata: potrebbe crescere indefinitamente, non vi è un sistema per rimuovere quelle già inserite (potrebbe utilizzare un concetto di "data di scadenza")
- Nessuna autenticazione client
- FIFO può saturarsi con molti client simultanei

### 8.3 Possibili estensioni per un utilizzo intensivo

- Implementare cache LRU con dimensione massima
- Supporto per hash multipli (MD5, SHA-512, ecc.)
- Statistiche server (hit rate, latenza media, throughput)
- Rate limiting per client

## 9 Conclusioni

Il progetto dimostra l'implementazione di un servizio concorrente per il calcolo di hash SHA-256. L'architettura basata su **thread pool**, **coda prioritaria** e **cache condivisa** garantisce:

- **Efficienza**: riutilizzo di hash già calcolati e velocità di processazione grazie ai thread

- **Correttezza**: sincronizzazione e nessuna race condition
- **Scalabilità**: gestione concorrente di client multipli

La difficoltà principale è stata l'implementazione della cache condivisa con attesa di thread multipli, nonostante il mutex e le condition variable mi son dovuto concentrare sulla gestione corretta della sincronizzazione per evitare race condition e crash inattesi.

# 10 GitHub

Il codice sorgente è stato anche caricato su GitHub, con GitHub action che eseguono unit test ogni nuova versione per garantire la correttezza del codice.

Link alla repository