# Progetto di Laboratorio di Sistemi Operativi - Filippo Costa, 596533

**Introduzione.**
Il progetto è strutturato in un modulo *client* e un modulo *server*, come da specifica. I codici sorgenti C99 dei due moduli si trovano rispettivamente ai path *src/client* e *src/server*, mentre i file top-level di *src/* e *include/* contengono file di supporto, condivisi dai due moduli. L'intero progetto è reso disponibile su licenza MIT e fa uso di tre librerie di supporto, i cui codici sorgenti sono disponibili come *git submodules* in *lib/*:

- *log.c*: https://github.com/rxi/log.c
  Autore: *user rxi*
  Licenza: MIT (https://github.com/rxi/log.c/blob/master/LICENSE)
  Utilizzo: Adotto *log.c* per logging da lato sia server che client. Ho deciso di utilizzare questa libreria perché è piuttosto conosciuta, stabile, ben mantenuta, produce output di ottima qualità e con un formato facilmente leggibile ma anche interpretabile a macchina.
- *tomlc99*: https://github.com/cktan/tomlc99
  Autore: C.K. Tan
  Licenza: MIT (https://github.com/cktan/tomlc99/blob/master/LICENSE)
  Utilizzo: Adotto *tomlc99* per effettuare parsing corretto e performante del formato TOML per i file di configurazione del server. Ritengo che affidarsi a file formats conosciuti e noti sia, nella maggior parte delle situazioni, la scelta migliore. TOML è un formato molto semplice -piuttosto simile a INI- e *tomlc99* offre ottimo controllo sul parsing dei file dati in input.
- *xxHash*: https://github.com/Cyan4973/xxHash
  Autore: Yann Collet
  Licenza: BSD 2-Clause (https://github.com/Cyan4973/xxHash/blob/dev/LICENSE, sottolineo la compatibilità di BSD 2-Clause con MIT)
  Utilizzo: Adotto xxHash come funzione di hashing nella tabella hash. xxHash è noto per avere ottime prestazioni, in particolare su stringhe di media lunghezza come i filepaths che fungono da chiave. Viene offerto sia un hash a 32-bit e da 64-bit; ho scelto di affidarmi esclusivamente all'hash a 64-bit per una questione di semplicità e prestazioni (i benchmark su GitHub mostrano che su un'architettura a 64-bit l'hash ha una velocità nettamente superiore). È un hash non crittografico, che nel nostro caso non comporta problemi; notiamo che questo ci esporrebbe ad attacchi di tipo HashDOS nel caso in cui il file storage server ricevesse "untrusted input". Questo tipo di attacco è prevalentemente accademico ma è bene esserne consapevoli.

I codici sorgenti del progetto sono disponibili su GitHub al link https://github.com/neysofu/uni-file-storage e la *Git history* è molto completa. Nella sezione *releases*, invece, è possibile scaricare l'archivio che include sia le sorgenti che i file di test: https://github.com/neysofu/uni-file-storage/releases. In questo documento si fa riferimento all'ultima versione.

La compilazione di queste librerie è gestita interamente dal Makefile che si trova nella top directory del progetto. I target del Makefile sono, come da specifica, I seguenti:

- *server*
- *client*
- *clean* (a.k.a. *cleanall*)
- *test1*
- *test2*
- *test3*

**Architettura.**
La prima scelta effettuata a livello di design di sistema è stata quella del protocollo di comunicazione tra client e server. Il protocollo di comunicazione è *transport-agnostic*; questo significa che l'utilizzo di Unix sockets di tipo SOCK_STREAM non è vincolante, e potrebbe essere sostituito da un protocollo semanticamente simile quale TCP senza alterare il design del sistema. Ogni messaggio inviato dal client al server possiede un header di 8 bytes, che indica la lunghezza del contenuto del messaggio con un intero *big-endian* senza segno. Il contenuto effettivo del messaggio ha come primo byte un tag che individua univocamente il tipo di operazione richiesta e a seguire operandi di tipo, lunghezza ed encoding variabile a

seconda dell'operazione stessa. I tag rispettano in modo piuttosto fedele l'API offerta dall'*header file* della specifica.

Il codice sorgente del server è distribuito su un numero non indifferente di files; indico in seguito le principali componenti del server, i relativi file sorgenti e come questi interagiscono fra di loro. A ogni file sorgente *.c* è associato un *header file* con le rispettive APIs.

- *config.c* effettua il parsing dei file di configurazione in un *struct Config*.
- *htable.c* implementa lo storage, offerro la struttura dati *thread-safe* che mantiene in memoria principale i contenuti dei file salvati nel server. All'interno del file viene anche implementato il *visitor pattern* (fondamentale per *readNFiles*) e le politiche di rimpiazzamento della cache.
- *deserializer.c* implementa una macchina a stati finiti che fa il decoding di un messaggio con contenuti arbitrari e con un *length prefix* di 64-bit in *big-endian*.
- *receiver.c* è un *polling loop* che mantiene un *struct Deserializer* per ogni *file descriptor* da cui *select()* legge dati in ingresso. Quando il *deserializer* ottiene un messaggio completo, viene scelto un worker thread a caso e il messaggio viene aggiunto alla relativa *queue*.
- *worker.c* effettua ulteriore decoding dei messaggi estratti dalla *queue* del *worker* e si interfaccia con la *struct Htable* che contiene i dati per produrre una risposta alla *query*.
- *workload_queue.c* implementa la *queue* sopracitata da cui ogni *worker* estrae i messaggi in ingresso.
- *global_state.c* contiene varie variabili globali e il codice di gestione di stato condiviso. Ho evitato lo stato globale quanto più possibile e per motivi di mantenibilità ho deciso di dichiarare le rimanenti variabili di stato globale in un unico file.
- serverapi.c implementa il protocollo di comunicazione su socket tra clients e server.

**Worker threads.**
All'avvio del server, viene inizializzato un numero di *worker threads* indicato dal file di configurazione. Ognuno di questi worker threads ha accesso alla sua queue di messaggi in arrivo, salvati in memoria e in attesa di essere processati dal worker thread. Per immettere I messaggi in arrivo nelle queues dei worker threads, il server deve effettuare due operazioni fondamentali: (1) decoding dei messaggi in arrivo e (2) una volta completato il decoding di un messaggio, scegliere a quale worker thread affidare il messaggio. In alternativa, sarebbe possibile disporre di un'unica *queue* condivisa tra tutti I worker threads: si tratta di un'architettura più semplice ma che potrebbe sottoporre il server a maggior thread starvation in momenti di stress. D'altra parte, un load balancer implementato male può facilmente annullare qualsiasi beneficio ottenuto con questa scelta. Ho scelto un semplice algoritmo di scelta casuale, che nei momenti di stress per la legge dei grandi numeri si comporta in modo piuttosto prevedibile e performante ed è molto utilizzato anche in ambito web. Si ottiene così un modello totalmente asincrono tra i messaggi ricevuti dal server, in cui l'ordinamento cronologico è parziale e ristretto all'interno di un unico worker thread: non si ha modo di sapere quale fra due messaggi appartenenti a queues diverse sia arrivato prima. Con scelte opportune di strutture dati interne, questo non è un problema.

**Storage.**
La struttura dati che funge da *storage* interno in memoria principale dei file memorizzati dal server è senza dubbio la parte più complessa del progetto. Intuitivamente, c'è bisogno di un meccanismo sincronizzato tra thread di associazione tra *filepaths* e dati binari (la funzionalità di *locking* complica un po' questa definizione, ma non cambia l'idea di fondo). Una possibilità sarebbe quella di usare un *mutex* di guardia che regoli l'accesso a una tabella hash: le operazioni di modifica, inserimento e (un)locking di file memorizzati nella tabella hash, tuttavia, sono piuttosto costose: richiedono l'utilizzo di ingenti quantità di RAM, il che ha un costo non indiffente su architetture moderne con cache gerarchiche di memoria. Sincronizzare queste costose operazioni con un unico mutex risulterebbe in prestazioni ridotte.
Ho perciò optato per una tabella hash con liste di trabocco "a sincronizzazione locale": l'accesso a ogni lista di trabocco è protetto da un mutex unico a tale lista, e quindi operazioni di lettura e scrittura sulla tabella hash possono procedere in parallelo (a meno che le chiavi non risultino in una *hash collision* sulla stessa lista di trabocco, in tal caso non è possibile nessun parallelismo). Per alcune operazioni rimane necessario adottare un meccanismo di sincronizzazione globale, ma si tratta di operazioni semplici e veloci mirate quasi esclusivamente a raccogliere statistiche interne sull'utilizzo della tabella hash al fine di produrre un report finale d'utilizzo.

# Progetto di Laboratorio di Sistemi Operativi - Filippo Costa, 596533

L'utilizzo di liste di trabocco a sincronizzazione locale permette pattern di accesso altamente parallelizzabili. Prendiamo come esempio l'implementazione di *readNFiles*: questa si appoggia all'API *visitor* esposta in *htable.h*. In seguito alla creazione di un *visitor* con *htable_visit*, le successive chiamate a *htable_visitor_next* mantengono il locking sul mutex sulla listra di trabocco dell'ultimo elemento visitato. Questo permette di effettuare chiamate a *readNFiles* che raccolgono ingenti quantità di dati (quindi costose) senza bloccare l'accesso in lettura e/o scrittura alla maggior parte degli altri file.

**Logging**.
Come accennato in precedenza, il logging su lato server è effettuato con la libreria *log.c*. Nel file *src/server/global_state.h* ho reso disponibili dei wrappers thread-safe per effettuare logging sincronizzato tra worker threads. Notiamo che fare locking e successivamente unlocking di un mutex globale per ogni operazione di logging è molto costoso e ho fatto questa scelta per semplicità. In un sistema *production-ready*, sarebbe opportuno riservare un thread unicamente al logging e usare un meccanismo ottimizzato di message-passing.

**Test.**
Tutti i test usano come file di prova I contenuti della cartella *test/data/Imgur*, che contiene un gran numero di immagini scaricate dal sito di condivisione immagini Imgur. Tutti I test stampano a schermo il resoconto prodotto da *statistiche.sh*, per convenienza di debugging e completezza, nonostante non richiesto espressamente dalla specifica.

1. *test/test1.sh* è un test generale delle funzionalità del sistema e memorizza sul server molti file. Il test può richiedere un po' di tempo e al termine dell'esecuzione I file letti dal server si trovano in *test/data/Imgur/target* e quelli rimossi dalla cache nella sottocartella *evicted*.
2. *test/test2.sh* è molto più semplice e l'esecuzione è pressoché istantanea. Il codice del test effettua dei test altamente specifici, scrivendo e leggendo file di misure molto precise sul server e aspettandosi la rimozione di un numero esatto di files a ogni passaggio.
3. *test/test3.sh* ha una durata esatta di 30 secondi e, più che qualche funzionalità in particolare, mira a testare la capacità del server di rimanere attivo e funzionante anche quando sotto una mole di carico notevole.

Nota bene: I file di configurazione dei test, contenuti nella cartella *config/*, specificano l'uso di un file socket in */tmp/* il cui accesso potrebbe richiedere privilegi *root*, e.g. *sudo make test2*.

**Sviluppi opzionali.**
Oltre alla *cache eviction policy* FIFO, ho implementato una *policy* alternativa che sfrutta il design della tabella hash ad accesso parallelo: le liste di trabocco vengono perciò usate come *queue* FIFO a tutti gli effetti, e la cache assume un grado di *associativity* pari al numero di *buckets*. Questa tecnica non richiede alun overhead di memoria. Nel codice, do a questa politica di rimpiazzamento il nome di *segmented FIFO*. Non è utilizzata nei test per evitare confusione, dato che può portare a comportamenti ed rimozioni di file inaspettate rispetto a un algoritmo più prevedibile come FIFO.

Ho valutato la possibilità di comprimere i contenuti dei file in memoria con LZ4 (https://github.com/lz4/lz4), ma ho successivamente deciso di evitare per ragioni di tempistiche.