## Architettura del server

Il server utilizza in totale 3 threads + tanti worker threads quanti specificato nel file di configurazione (almeno uno). I threads sono specializzati in questo modo:

* Master thread: ascolta sul server socket specificato nel file di configurazione, gestisce le nuove connessioni in entrata e avverte i worker threads se i client hanno inviato una nuova richiesta. Si occupa di mantenere una lista di client connessi e fa da fornitore per la coda di client da servire.
* Signal handler thread: gestisce i segnali SIGINT, SIGQUIT, e SIGHUP come da specifica. Se viene ricevuto uno di questi segnali, il thread invia un messaggio al master server, il quale provvede a svegliare i worker e a chiudere i file descriptor dei client connessi, dunque a terminare stampando le statistiche.
* Logging thread: si occupa di mantenere il log del server. I vari thread inviano messaggi tramite un pipe al thread di logging, e questo stampa su stdout e scrive sul file di log.
* Worker threads: si occupano di servire le richieste dei client. Quando il master thread riconosce che il file descriptor di un client è pronto ad una lettura, mette il file descriptor in una coda di client pronti e sveglia un worker. Questo quindi provvede a rimuovere il client dalla coda e a processarne la richiesta.

## Protocollo di comunicazione client-server

Il protocollo di comunicazione tra client e server è stato progettato specificamente per fare in modo che il server non si blocchi in alcun caso ad attendere una risposta di un client. Per ottenere ciò, di ogni client connesso si mantiene uno stato, che serve al server per sapere che messaggio deve aspettarsi dal client. Ad esempio, in caso il client esegua un’operazione di scrittura, come prima cosa invia al server un pacchetto contenente il tipo di operazione che vuole eseguire (FCP\_WRITE), la quantità di bytes che invierà al server, ed il nome del file su cui scrivere. A questo punto, il server non aspetta attivamente che il client invii i dati, ma aggiorna lo stato del client e considera la richiesta soddisfatta. Il client poi invierà i dati con una seconda richiesta, ed il server saprà quanti bytes leggere, essendo questi stati salvati nello stato del client. In questo modo, il server non rimane mai in attesa del client, ma succede solo l’opposto, le uniche read da parte del server sono eseguite quando la select segnala che è possibile eseguire una read senza bloccarsi.

Lo stato dei client contiene inoltre una lista di files aperti, utilizzata per implementare i meccanismi di apertura e di chiusura dei files del protocollo.

Lo stato dei client è anche usato per implementare il meccanismo di lock sui files: quando un client richiede un lock su un file:

* se nessun altro client possiede un lock su quel file, allora il client ottiene subito il lock;
* se un altro client possiede un lock su quel file, lo stato del client che lo ha richiesto viene settato a WaitingForLock, col nome del file salvato nello stato, e quando il client che possedeva il lock originariamente rilascia il lock (o si disconnette, in quanto la disconnessione comporta la chiusura automatica di tutti i files aperti e il rilascio di tutti i lock posseduti), il server controlla nella lista dei client (in ordine di connessione) se qualche client stava aspettando il lock su quel file. In caso positivo, il lock viene passato a quel client.

Se un client sta aspettando un lock su un file, ma il client che possedeva il lock in precedenza effettua un’operazione di cancellazione del file, il server invia al client in attesa del lock un messaggio di errore.

## Client

Il programma client è stato implementato come da specifica, con l’aggiunta di alcune opzioni da riga di comando:

* -v – Alias per -p
* -a file1,file2 – Invia al server una richiesta di append, apponendo al file file1 il contenuto di file2

## Client API

Le API lato client sono state implementate come da specifica, con l’aggiunta di una nuova funzione:  
che permette di specificare una cartella in cui salvare i file espulsi dal server, come fa la funzione writeFile. La funzione openFile2 si comporta come una chiamata a openFile2 con il parametro dirname settato a NULL, ovvero ignora i files inviati dal server.

## File di configurazione del server

Il file di configurazione del server è in semplice formato key=value, con value che può essere sia un valore numerico, sia una stringa. In caso sia valore numerico, viene riportato così com’è, in caso sia una stringa, dev’essere delimitato da "". Sono presenti alcuni parametri senza i quali il server non si avvia, ed altri opzionali. Per i parametri opzionali il server assume dei valori di default.

I parametri di configurazione sono:

|  |  |  |  |
| --- | --- | --- | --- |
| Parametro | Tipo | Required | Descrizione |
| cacheAlgorithm | String | No | Determina l’algoritmo di espulsione dalla cache utililzzato dal server. Se settato a "LRU", il server utilizzerà un algoritmo LRU. Se settato a "FIFO", ad una stringa invalida, o non settato, il server utilizzerà un algoritmo FIFO. |
| compression | String | No | Determina che algoritmo di compressione viene utilizzato dal server per salvare i file in memoria. Se settato a "none", allora i files vengono salvati non compressi, se settato a qualsiasi altro valore, o non presente nel file di configurazione, i files vengono compressi usando un algoritmo basato su zlib. |
| logFile | String | Sì | Determina il percorso in cui salvare il file di log. |
| logMode | String | No | Determina la modalità di scrittura del file di log. Se settato a "trunc", il file di log verrà sovrascritto ad ogni esecuzione del server. Se settato a qualsiasi altro valore, o se non settato, il file verrà aperto in modalità append. |
| logTimeFormat | String | No | Determina il formato del timestamp nel log e nell’output del server. Valori possibili:   * "none" – Il tempo non viene riportato nel log e nell’output; * "timestamp" – Il tempo viene riportato come timestamp unix. * "formatted" – Il tempo viene riportato come stringa formattata secondo il formato “yy/mm/dd HH:MM:SS”.   Se non indicato, o se la stringa non è un valore valido, il valore di fallback è timestamp. |
| maxFiles | Long | Sì | Determina il massimo numero di files che il server può contenere. |
| nWorkers | Long | Sì | Determina il numero di worker che vengono avviati. Minimo 1. |
| socketPath | String | Sì | Determina il percorso del socket su cui il server ascolta. |
| storageSize | String | Sì | Determina la dimensione massima che può raggiungere il server. Il formato è "numero[unità]", dove numero è un intero positivo e unità è un valore opzionale tra [b/B/k/K/m/M/g/G] e indica se il numero è da interpretarsi come numero di bytes, kilobytes, etc.  Esempio: "128M" = 128 megabytes, "1g" = 1 gigabyte. |

## Compressione

È stata implementata la compressione dei files salvati in memoria principale. Il server utilizza una libreria con licenza MIT, chiamata miniz (<https://github.com/richgel999/miniz>), scelta perché leggera, semplice da usare, e non block-based. Questa libreria implementa lo standard zlib.

Al momento del salvataggio di un file ricevuto dal client, il server comprime il file e controlla che il nuovo file ottenuto sia di dimensione minore del file di input. Se il file risulta più piccolo, allora viene salvato in formato compresso, altrimenti viene salvato non compresso, così da non doverlo decomprimere al momento della lettura. La compressione del file è del tutto invisibile al client, che invia e riceve files non compressi. È stata presa in considerazione l’idea di usare una cache per i file non compressi, in modo da non dover decomprimere più volte i files letti più frequentemente, ma per motivi di tempo non è stato possibile implementare quest’idea.

## Gestione della cache

La cache di file è implementata come una linked list di files. Questa lista è sincronizzata tra i vari thread con un rwlock, in modo da permettere più letture contemporaneamente, ma una sola scrittura. Questo per motivi di performance, poiché potenzialmente un gran numero di thread deve poter accedere contemporaneamente alla struttura, e parallelizzare le letture dovrebbe apportare un gran beneficio alle performance. Il rwlock è usato solo per la struttura della cache, ma ogni file ha anche un suo lock usato per l’accesso concorrente al contenuto ed ai metadati del file in questione.

Ogni file è rappresentato in memoria con una struttura contenente il nome del file, un buffer contenente il suo contenuto, informazioni sulla dimensione del file, sul client che ha effettuato un’operazione di lock su di esso (se presente), il lock del file, un timestamp che indica il momento dell’ultimo accesso, e informazioni sull’algoritmo di compressione usato per memorizzare il file.

Sono stati implementati due algoritmi di rimpiazzamento della cache:

* Come primo è stato implementato un algoritmo FIFO, sfruttando il fatto che la cache in sé è implementata come una lista: al momento della creazione del file, questo è inserito in testa alla lista. Quando l’algoritmo per determinare il file da eliminare per liberare spazio viene invocato, prende il file più in fondo alla lista possibile su cui nessun client abbia fatto una lock (e che sia diverso da un file passato opzionalmente come parametro).
* Secondo algoritmo implementato è un algoritmo LRU, che usando un timestamp relativo all’ultimo accesso al file, restituisce il file col timestamp più piccolo, sempre escludendo dalla ricerca i file su cui un client mantiene un lock ed un file passato opzionalmente come parametro.

Il file da escludere passato come parametro all’algoritmo di espulsione è utile nel caso in cui si effettui una una scrittura su file, per evitare che l’algoritmo decida di espellere il file su cui si sta scrivendo. Questo è utile in realtà solo nel caso in cui la scrittura si effettui su un file su cui non si ha un lock, cosa non possibile secondo la specifica delle API, ma si è deciso di implementarlo lo stesso in questo modo per renderlo più flessibile.

Per come sono specificate le API, la creazione e la scrittura del file sono eseguite in due momenti separati. Questo presenta un problema per la gestione della capacità del server, poiché è possibile che un file venga creato con successo, ma una scrittura non sia possibile perché il file è troppo grande e l’algoritmo di gestione della cache non riesce a determinare un file da espellere, o perché la dimensione del singolo file supera la capacità massima del server. In questo caso, il server rimarrebbe con un file al suo interno che è stato creato, ma che non ha contenuto. Per questo motivo, in caso di write/append fallita per motivi di capacità della cache su un file di dimensione 0, il file viene eliminato dal server per non lasciare un file vuoto in memoria.

## Sviluppo del progetto

Il progetto è stato sviluppato su piattaforma Windows 10 + WSL2 (Ubuntu 18.04) con l’ausilio dell’IDE CLlion (v. 2021.2.1).

Durante lo sviluppo del progetto è stato fatto uso estensivo di valgrind per assicurarsi di non incorrere in memory leaks ed accessi a locazioni invalide di memoria.

Come build system si è usato make, come richiesto dal progetto. È stato fatto uso di regole implicite per la maggior parte dei file di output, eccetto per i due eseguibili, client e server, che hanno una regola esplicita poiché richiedono un comando più complesso per essere compilati correttamente. Sono inoltre stati creati dei target phony che fungono sostanzialmente da shortcut per alcune operazioni come l’invio di segnali al server. Il target clean rimuove tutti i file oggetto creati come prodotto intermedio della compilazione dei due eseguibili, mentre il target cleanall, oltre a questi ultimi, elimina anche gli eseguibili, i files scritti dai test, il file socket ed il file di logging (se locati nei percorsi predefiniti /tmp/LSOfilestorage.sk e /tmp/LSOfilestorage.log)­­­­­­­­­.

Sono stati realizzati i tre test richiesti dalla consegna, più altri due test per testare il funzionamento del meccanismo di lock dei file e il comportamento del server in caso di ricezione del segnale SIGHUP. Per ogni test è anche presente nel makefile un target per eliminare tutti i file creati dai test.

Lo script statistiche.sh esegue il parsing del file di log del server. Nel caso non sia passato alcun file come parametro, lo script cerca il file di log nel percorso di default (/tmp/LSOfileserver.log), altrimenti lo cerca nel primo parametro passato. Oltre alle informazioni richieste, lo script riporta informazioni relative alla compressione dei file, come la quantità di bytes risparmiati nella memorizzazione e il rapporto di compressione.

Vista la possibilità di configurare il server per scrivere sul file di log in modalità append, lo script di statistiche eseguirà il parsing delle informazioni relative solo all’ultima esecuzione, ignorando quelle relative a precedenti esecuzioni.

È stato inoltre usato git come VCS, usando una repository pubblica su github, situata al seguente indirizzo: <https://github.com/Alex23087/SOL-Project>