## 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.

## 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 "".

I parametri di configurazione sono:

|  |  |  |
| --- | --- | --- |
| Parametro | Tipo | Descrizione |
| compression | String | 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 | Determina il percorso in cui salvare il file di log. |
| logMode | String | 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 | 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 | Determina il massimo numero di files che il server può contenere. |
| nWorkers | Long | Determina il numero di worker che vengono avviati. Minimo 1. |
| socketPath | String | Determina il percorso del socket su cui il server ascolta. |
| storageSize | String | Determina la dimensione massima che può raggiungere il server. |

## 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

implementata