Relazione

Progetto di Programmazione di Sistema A.A. 2020-2021

Note generali (sia Linux che Windows)

- Per quanto riguarda gli import abbiamo utilizzato la direttiva #ifndef seguita dal simbolo riconoscitivo della libreria importata per assicurarci che non avvengano più volte gli stessi import.
- Sia per il server che per il client abbiamo effettuato la validazione degli argomenti passati da tastiera al momento dell'esecuzione.
- Abbiamo realizzato i wrapper per le funzioni utilizzate più volte.
- Per quanto riguarda i comandi UPLOAD e DOWNLOAD abbiamo ritenuto necessario creare un sistema di ACK al fine di evitare che il server chiuda la socket del client prima che terminasse la ricezione di tutti i bytes inviati. Questo sistema consiste nell'invio da parte del client del comando 200 "OK" e nella ricezione da parte del server di quest'ultimo (per il comando UPLOAD), viceversa per il comando DOWNLOAD.
- Un file di esempio contenente le configurazioni (-c) (utile per osservare il formato) sarà situato nella directory Linux/server e Windows/Server con il nome di config

Versione Linux

Note utili:

- Nella directory Linux sarà presente un makefile che permetterà attraverso il comando make di effettuare la produzione degli eseguibili (client e server) e con il comando make clean sarà possibile eliminarli.
- Nella directory Linux ci sono due file (common.c e common.h) che contengono diverse funzioni e wrapper utilizzati sia dal client che dal server.
- In fase di sviluppo abbiamo eseguito la compilazione con diversi flags tra cui: -Wall -Werror -Wextra -fsanitize=address al fine di generare un codice migliore e evitare errori di accesso alla memoria.

Server:

- Anche se non richiesto, al fine di permettere una corretta terminazione del server, abbiamo inserito una macro (MAX_ACCEPT) indicante il numero di connessioni dopo le quali il server dovrà terminare la sua esecuzione attualmente impostata a 10 (presente nel file source_server.c)
- Per permettere successive esecuzioni del server, anche nel caso in cui non dovesse essere correttamente terminato e quindi evitare errori sulla Bind, abbiamo impostato l'opzione della socket SO_REUSEADDR a 1
- Al fine di facilitare la comprensione dello svolgimento dell'esecuzione del server in modalità daemon, il flusso dello stdout e stderr sarà ridirezionato nel file chiamato "syslogd" situato nella cartella Linux/server, il server produrrà diversi messaggi informativi sullo stato del server e dello svolgimento delle operazioni

** Motivazioni di questa scelta:

Solitamente i processi daemon utilizzano il system logger per salvare messaggi, ma per permettere una migliore accessibilità ho optato per la creazione di un file apposito

Thread

- All'avvio del server o alla rilettura delle configurazioni (in seguito al segnale SIG_CHILD) viene allocato dinamicamente un array di struct di tipo my_thread della dimensione passata in input o con il valore di default contenente i threads che saranno usati per esaudire le richieste provenienti dai client.

  L'utilizzo dei threads sarà regolamentato attraverso l'uso di un semaforo, il cui valore sarà inizializzato in base al numero di threads scelti. Tale semaforo ci permetterà di non far accettare connessioni al server se non sono disponibili threads.

  Per identificare se un thread è occupato o meno la struct my_thread conterrà un campo chiamato state che potrà valere 0 o 1.

Ciascun thread sarà utilizzato per un solo client alla volta e si occuperà di autenticare il client e se autenticato con successo eseguirà il comando richiesto dal client. Inoltre, ciascun thread loggherà sul file log specificato o se non specificato sul file presente nella directory tmp (chiamato server.log) le informazioni indicate sulla traccia del progetto.

Come meccanismo di lock utilizzato per permettere l'accesso in scrittura al file log abbiamo scelto di utilizzare un mutex inizializzato senza attributi, che permette a un solo thread alla volta di scrivere sul file.

Comandi utili per osservare lo stato / l'esecuzione del server:

// ottieni pid del server

- ps -e | grep "server"

// il server si riavvierà con i parametri presenti nel file config (se avviato con l'opzione -c <path>)

- kill -s 1 <server_pid>

// visualizza la porta utilizzata dal server

- netstat -ae | grep "server"

// comando da usare dopo che il server è entrato nella modalità daemon, ci consente di sapere se è stato correttamente daemonizzato

- ps -xj | grep "server"

// visualizza i messaggi informativi riguardanti l'esecuzione e lo stato del server

- cat syslogd

Versione Windows

Note utili:

- Nella directory Windows sono presenti due file (functions.c e functions.h) che contengono diverse funzioni e wrapper utilizzati sia dal client che dal server.
- Abbiamo realizzato i wrapper per le funzioni utilizzate più volte.
  Alcuni di questi hanno una modalità (Flag) che indica se notificare gli errori localmente o no. Infatti per errore locale, intendiamo un errore che blocca l'esecuzione, quindi un messaggio di errore + exit(1); altrimenti avremmo un messaggio di errore + ExitThread(1).
- Per controllare possibili problemi di allocazioni e memory leaks abbiamo utilizzato un software "Memory Validator", in quanto stumenti come -fsanitize=address non sono supportati da MingW.
- Inoltre è importante ricordare che in fase di compilazione è necessario linkare la dll di Win 32 utillizzando il flag -lws2_32 :

  ```
  gcc -Wall -Werror -Wextra -o server.exe ./../functions.c server.c -lws2_32
  gcc -Wall -Werror -Wextra -o client.exe ./../functions.c client.c -lws2_32
  ```

Server:

- Anche se non richiesto abbiamo inserito un numero massimo di connessioni dopo il quale il server temina = numero di threads scelti * 3.

Thread

- All'avvio del server viene creato un array di struct tread, contenente informazioni quali lo stato, il file di log utilizzato, la socket sul quale comunicare, indirizzo IP, porta e timestamp di avvio thread. Ogni thread avrà tutte le informazioni necessarie per esaudire la richiesta del client.

Si tiene conto del numero di thread pronti tramite la variabile readyThreads, che insieme a un semaforo permette di andare a controllare e bloccare l'accettazione di nuove connessioni. Così facendo possiamo distinguere due casistiche, se readyThreads == 0 allora effettueremo una WaitForSingleObjects con il flag waitAll = FALSE, in modo da poter rilasciare le risorse e il semaforo per poter accettare nuove connessioni (readyThreads++); altrimenti se readyThreads > 0 allora periodicamente verranno controllati gli stati dei threads così da rilascare il semaforo e le risorse lì dove possibile (readyThreads++).

Una volta aver accettato una connessione, viene incaricato un thread (se libero) di inziare a comunicare con il client, così dopo aver effettuato l'autenticazione si potrà esaudire la richiesta del client. Ovviamente, il thread avrà a disposizione come argomenti alla funzione la struct thread ben inizializzata per portare a termine la richiesta.

Per non rallentare l'architettura si è deciso che la scrittura sul file di log, presente nella dir Windows/Server/tmp chiamato server.log, non avvenga tramite mutex, ma sfruttando un flag FILE_APPEND_DATA, che permette di effetturare append atomiche al file.

Test:

- Abbiamo testato sia l'approccio ibrido che non e non abbiamo rilevato malfunzionamenti, inoltre, abbiamo testato l'esecuzione su macchine remote nella stessa rete, per fare questo abbiamo dovuto disabilitare i firewall interni alle macchine ed effettuato un port forwarding.