Relazione finale del progetto di Sistemi Operativi (6 CFU)

Anno 2021/2022

Giuseppe Marseglia, matricola n. 0252379

Indice

Risorse

Dizionario

Operazione, operation

Unità di scambio di informazioni tra server e client. Invitata tramite socket.
Un'operazione è composta da:

- **UID, User IDentifier** : codice identificativo del mittente
- **Codice, Code** : Codice dell'operazione
- **Testo, Text** : Stringa che compone il corpo dell'operazione, non obbligatorio

Messaggio, message

Unità di informazioni presenti nella bacheca. Inviati dai client.
Un messaggio è composto da:

- **MID, Message IDentifier** : codice identificativo del messaggio
- **UID mittente, sender UID** : UID del mittente
- **Oggetto, subject** : Intestazione, titolo del messaggio
- **Testo, body** : Corpo, testo del messaggio

Azione, action

Funzione fornita all'utente dal sistema, definite nella specifica data.
Le azioni possibili sono:

- **Leggere, read all** : Il client richiede al server di leggere tutti i messaggi.
- **Inviare, post** : Il client invia un nuovo messaggio.
- **Rimuovere, delete** : Il client richiede di eliminare un messaggio da lui inviato.

Credenziali dell'utente, user info

Informazioni necessarie per il login e la registrazione di un nuovo utente, salvate dal server su un apposito file.

Le credenziali dell'utente sono composte così:

- **UID, User IDentifier** : Identificativo dell'utente.
  Viene calcolato incrementalmente per ogni nuovo utente. 0 e 1 sono rispettivamente usati per contrassegnare utente anonimo e Server.
- **Username** : Stringa di massimo 32 caratteri.
  - Usato dal Client per il Login.
- **Password** : Stringa di massimo 32 caratteri.
  Usato dal Client per il Login. La password viene scritta in chiaro dall'utente che usa il Client, criptata, inviata al Server, criptata di nuovo, e infine salvata su file.

File

File generati dal Server per la persistenza dei dati.

- **File degli utenti, Users file** : il file contiene l'UID, l'username e la password criptata per ogni utente. Il file è leggibile con '*cat*'.

- **File della bacheca** :
  - **File dei messaggi, Messages file** : questo file contiene gli oggetti ed i testi di tutti i messaggi.
    Il file è leggibile con *'cat'*.
  - **File degli indici, Index file** : questo file contiene l'UID del mittente, l'offset all'interno del file dei messaggi e la lunghezza di ogni messaggio.
    Il MID è uguale al numero di riga. Il file va letto con '*od -t d4'* in quanto è salvato in binario.
  - **File delle aree libere, Free Areas file** : questo file contiene l'offset e la lunghezza di tutte le aree libere all'interno del file dei messaggi.
    Le aree libere vengono generate dall'eliminazione di un messaggio. Il file va letto con '*od -t d4'* in quanto il messaggio è salvato in binario.

Diagrammi

Nella cartella *'Diagrammi/'* ci sono dei diagrammi che mostrano l'architettura generale del Server e del Client, lo svolgimento del login, della registrazione e delle varie azioni, sia per il Server sia per il Client.

Introduzione contenente le specifiche del progetto

Specifica da implementare: Bacheca elettronica

Realizzazione di un servizio "bacheca elettronica" il quale permetta ad ogni utente autorizzato di inviare messaggi che possono essere letti da ogni altro utente interessato a consultare la bacheca stessa.

Il servizio di accesso alla bacheca elettronica deve essere offerto da un server che accetta e processa sequenzialmente o in concorrenza (a scelta) le richieste dei client (residenti, in generale, su macchine diverse dal server).

Un client deve fornire ad un utente le seguenti funzioni:

1. Leggere tutti i messaggi presenti sulla bacheca elettronica.
2. Spedire un nuovo messaggio sulla bacheca elettronica.
3. Rimuovere un messaggio dalla bacheca elettronica, se inserito dallo stesso utente interessato a cancellarlo (verifica da effettuare tramite un meccanismo di autenticazione a scelta).

Un messaggio deve contenere almeno i campi Mittente, Oggetto e Testo.

Si precisa che la specifica richiede la realizzazione del software sia per l'applicazione client che per l'applicazione server.

Per progetti misti Unix/Windows e' a scelta quale delle due applicazioni sviluppate per uno dei due sistemi.

Specifica adottata

Il servizio di accesso alla bacheca elettronica è offerto da un Server multi thread, che accetta e processa in concorrenza le richieste dei Client. (Sono stati testati sia Clients che risiedono sulla stessa macchina, sia Client che risiedono su macchine diverse ma collegate alla stessa rete).

È presente un meccanismo di autenticazione tramite username e password, preventivo allo svolgimento di qualsiasi azione.

I messaggi contengono i campi: ID del mittente, Oggetto e Testo.

L’applicazione è stata realizzata per essere eseguita in ambiente UNIX.

Discussione delle scelte di progetto e realizzative, e delle tecniche e metodologie generali usate

Scelta del protocollo di comunicazione e gestione della connessione

Il protocollo di comunicazione scelto è **TCP**. In questo modo si ha uno scambio di messaggi affidabile.

Seguendo il concetto di connessione di TCP, il Client si connette al Server durante la fase di login o di registrazione, e rimane connesso fino alla esplicita richiesta di disconnessione o alla terminazione del Client o del Server.

Questo approccio **semplifica la scrittura del codice** per lo scambio di operazioni tra Server e Client.

Lo **svantaggio** principale è però che il Server ha a disposizione un numero limitato di thread adibiti alla comunicazione con i Clients, quindi **il numero di utenti contemporaneamente attivi è limitato**.

Un utente è considerato attivo anche se in stato di *idling*, ovvero lo stato nel quale il Server attende una richiesta dal Client.

Si potrebbe implementare un meccanismo di disconnessione forzata di Client, con relativo re-login automatico. La scelta della “vittima” potrebbe essere tramite timer o scegliendo fra i Client che non fanno richieste da più tempo.

Un’altra ipotesi che era stata presa in considerazione era quella di un approccio dove per ogni operazione il Client si connette al Server per poi disconnettersi al completamento della richiesta. In questo modo si eviterebbe il problema degli utenti in idling, complicando però il meccanismo di autenticazione dei Client. Per quanto riguardo il login si potrebbe utilizzare un meccanismo di token randomici, che vengono assegnati al Client nel momento del login e che vanno poi ripetuti in ogni operazione.

Implementazione dello scambio dei dati tra Server e Client

Lo scambio di dati tra Server e Client e viceversa avviene attraverso lo scambio di unità chiamate "**operazioni**". Come indicato nel dizionario, ogni operazione contiene l'ID del mittente, il codice che identifica il tipo di operazione, e eventualmente un testo che contiene i dati.
I codici sono condivisi fra Server e Client e sono in *common.h*, un header condiviso.
Alcuni codici come OK e NOT ACCEPTED, vengono usati per notificare il successo od il rifiuto di un'azione. Altri come MSG, vengono usati per indicare al Server o al Client che tipo di richiesta si sta effettuando, e contemporaneamente inviare i dati per eseguirla.

Lo **scambio di operazioni** avviene **attraverso due funzioni**, che presentano entrambe una variante.

Le due funzioni **ritornano** 0 se lo scambio è avvenuto, e -1 altrimenti. In questo modo il mittente può facilmente controllare il risultato della funzione.

Per migliorare la portabilità, le due funzioni utilizzano *uint32_t*, di taglia fissa 4 byte, piuttosto che *int*, che solitamente ha taglia 4 byte ma può variare.

send_operation_to() e send_operation_to_2()

Questa funzione e la sua variante vengono usate per **inviare operazioni**.

La funzione invia l'operazione in due step:
Nel primo step la funzione usata è una ***'writev()'*** : In questo passaggio vengono inviati, l'ID del mittente, il codice e la lunghezza del testo che sarà inviato. Viene usato *'writev()'* perché permette l'invio atomico di più buffer non contigui.
Nel secondo step la funzione usata è una ***'send() con flag MSG_NOSIGNAL'*** : In questo passaggio viene inviato il testo dell'operazione. La flag *MSG_NOSIGNAL* è usata per poter gestire l'errore *EPIPE*, quando il SO riconosce che il canale di comunicazione è stato interrotto.

Il puntatore all'area di memoria che contiene il testo può essere NULL : in questo caso la 'send()' non viene chiamata e la lunghezza del testo è impostata a 0.

receive_operation_from() e receive_operation_from_2()

Questa funzione e la sua variante vengono usate per **ricevere operazioni**.

La funzione, similmente alla '*send_operation_to()*' riceve l'operazione in due step:
Nel primo step la funzione usata è una ***'readv()'*** : In questo passaggio vengono letti l'ID del mittente, il codice e la lunghezza del testo che sarà ricevuto. Viene usata una *'readv()'* specularmente alla *'writev()'*.
Viene quindi allocata una zona di memoria per contenere il testo. La scelta di inviare le operazioni in due momenti è stata presa nelle prime parti dello sviluppo per facilitare

l'allocazione dinamica dei testi delle operazioni. L'area di memoria allocata **dovrebbe essere liberata** con una *'free()'* dopo essere stata utilizzata.

Nel secondo step la funzione è una ***'read()'*** : In questo passaggio viene letto il testo dell'operazione. Non è stato necessaria una *'recv()'* in quanto non era necessaria nessuna FLAG.

Se la lunghezza del testo letta è 0, allora la seconda funzione non viene invocata.

Scelte riguardanti il Server

Persistenza

Il Server per la persistenza si avvale di **4 file**:

- 1 viene usato per l'autenticazione degli utenti.
- 3 vengono usati per i messaggi della bacheca.

I file vengono aperti, o creati se non esistono, a partire dalla *Current working directory* da cui viene lanciato il Server. È importante quindi lanciare il Server dalla precedente *cwd* per mantenere lo stato di persistenza precedente.
I file possono essere visualizzati, ma **non** dovrebbero mai essere modificati manualmente.

Multi threading

Per avere la possibilità di gestire in concorrenza le richieste dei Client, il Server è stato strutturato come applicazione multi thread.

Sono presenti due tipi di thread:

1. Il **Main Thread**, che viene lanciato per primo, si occupa del *setup*, accetta le connessioni con i Client e crea il secondo tipo di thread. Ce ne può essere uno solo.
2. I **Communication Thread**, che si occupano dell'esecuzione delle richieste dei Client. Ce ne possono essere diversi.

Il Main Thread crea i Communication Thread solo quando accetta una nuova connessione.
I Comm. Thread possono terminare o dopo aver ricevuto la richiesta di disconnessione dal Client, o dopo aver riscontrato una terminazione del Client, o tramite segnale inviato dal Main Thread (spiegato successivamente).
Sia perché **il numero di Comm. Thread deve essere limitato**, sia perché è necessario salvare delle informazioni dei Comm. Thread per permettere al Main Thread di inviare loro segnali, il Server mantiene una **bitmask degli "slot" dei Comm. Thread**.
Il numero massimo di thread attivi contestualmente è una costante definita in *server.h*.

La **bitmask** è gestita internamente da un modello di ***lock* e *unlock***, e può quindi essere modificata sia dal Main Thread sia dai Comm. Thread in sicurezza.
Il Main Thread indica uno slot come occupato quando crea un Comm. Thread. I Comm. Thread indicano il proprio slot come libero quando terminano.
La bitmask è stata realizzata insieme al progetto e non è quindi una libreria esterna.

Naturalmente la scelta di un'applicazione **multi thread comporta problemi aggiuntivi** di sincronizzazione e di consistenza delle informazioni.

Sincronizzazione fra i thread del Server

La sincronizzazione fra i thread del Server è garantita da due coppie di semafori, più un quinto semaforo.

Le **due coppie** di semafori servono per la **sincronizzazione sulle letture e le scritture sui file**. Viene usata una coppia di semafori per il file degli utenti, ed un'altra coppia per i file della bacheca.

Vengono usate delle coppie di semafori, piuttosto che un semaforo singolo, per permettere situazioni di **single writer multiple reader**. Ovvero più thread possono leggere contemporaneamente lo stesso file, mentre la scrittura può essere eseguita contestualmente da un solo thread alla volta.

Il **quinto semaforo**, viene utilizzato per notificare al Main Thread **quando almeno uno slot è libero.** Il Server non accetta nuove connessioni con i Client, fino a quando almeno uno slot è libero.

Dopo aver riscontrato che c'è almeno uno slot libero, il Main Thread usa la bitmask per determinare qual'è.

Segnali

I segnali di cui è stata modificata la gestione sono **SIGINT** e **SIGUSR1**.

Il segnale **SIGINT**, viene usato per indicare al Server di cominciare la routine di terminazione. SIGINT viene sempre bloccato sui Comm. Thread, mentre è permesso in alcuni momenti dell'esecuzione del Main Thread, sul quale comincia il gestore del segnale (sono indicati nel diagramma Architecture e verranno poi indicati più nello specifico nelle sezioni successive).

Il **gestore del segnale SIGINT** che parte sul Main Thread esegue queste operazioni:

1. Chiude il socket adibito ad accettare le connessioni.
2. Manda a tutti i Communication Thread il segnale SIGUSR1. Il segnale SIGUSR1 causerà la terminazione dei Communication Thread, appena questa sia per loro possibile.
3. Fa la *join()* di tutti i Communication Thread.
4. Chiude i semafori aperti.
5. Termina il Main Thread e quindi l'applicazione.

Il segnale **SIGUSR1** viene inviato dal Main Thread ai Comm. Thread per indicare loro di cominciare la routine di terminazione. L'invio a thread specifici è permesso dalla funzione '*int pthread_kill(pthread_t thread, int sig);*'.

SIGUSR1 è sempre bloccato sul Main Thread, mentre è permesso in alcuni momenti dell'esecuzione dei Comm. Thread, sui quali cominciano i gestori del segnale (sono indicati nel diagramma Architecture e verranno poi indicati più nello specifico nelle sezioni successive).

Il **gestore del segnale SIGUSR1** che parte sui Communication Thread esegue queste operazioni:

1. Esegue la routine di connessione con il Client (indicata nella sezione dei Communication Thread).
2. Termina il thread.

Main Thread

Il Main Thread fa il setup dell’ambiente, che comprende i semafori, la bitmask e la gestione dei segnali, crea il socket TCP che accetterà le connessioni, fa il bind del socket ad un indirizzo passato per parametro all'applicazione e si mette in stato Listen per ricevere connessioni.

Comincia quindi il Main Cycle del Main Thread del Server.

Main Cycle del Main Thread

Durante il suo Main Cycle, il Main Thread esegue queste operazioni:

1. Permette la ricezione del segnale SIGINT.
2. Aspetta tramite il semaforo *sem_free_threads* che ci sia almeno un slot libero. Inizialmente tutti gli slot sono considerati come liberi.
3. Usa la bitmask per trovare il primo slot libero.
4. Chiama *accept()* e si mette in attesa che ci sia una connessione con un Client disponibile.
5. Accettata una connessione, blocca il segnale SIGINT.
6. Segnala lo slot prima trovato come occupato.
7. Crea un Communication Thread che si occuperà di gestire le azioni che il Client connesso richiederà.
8. Ritorna allo step 1.

Il Main Cycle può essere interrotto solo dall’arrivo del segnale SIGINT; alla ricezione del segnale verrà avviato il gestore del segnale.

Il segnale **SIGINT viene sbloccato** solo durante la fase di attesa di uno slot libero o di una connessione di un Client. In questo modo la creazione di uno nuovo thread non può essere interrotta.

Communication Thread

I Communication Threads si occupano dell'esecuzione delle azioni richieste dal Client.

Quando viene avviato, un Communication Thread esegue queste operazioni:

1. Riceve l'operazione che il Client invia per cominciare l'handshake prima dell'invio delle credenziali.
2. Invia l'operazione per terminare l'handshake.
3. Riceve le credenziali che il Client invia per fare l'accesso e la modalità di accesso: login o registrazione.
4. Esegue il login o la registrazione e risponde al Client con l'User ID in caso di successo o notifica il fallimento.
5. In caso di fallimento, esegue la routine di chiusura della connessione.
6. In caso di successo, comincia il suo Main Cycle.

Main Cycle dei Communication Threads

Durante il suo Main Cycle, il Comm. Thread esegue queste operazioni:

1. Permette la ricezione del segnale SIGUSR1.
2. Si mette in attesa di ricevere un operazione dal Client.
3. All'arrivo dell'operazione, blocca la ricezione del segnale SIGUSR1.
4. Esegue l'azione associata alla richiesta del Client, inviando eventualmente operazioni al Client.
5. Torna allo step 1.

Alla ricezione del segnale SIGUSR1 verrà avviato il gestore del segnale.
Il segnale **SIGUSR1 viene sbloccato** solo durante la fase di attesa di una operazione da parte di un Client. In questo modo **l'esecuzione delle richieste già in corso non può essere interrotta**, ma allo stesso tempo il Comm. Thread può terminare quando sta aspettando il Client.

Chiusura della connessione con il Client

Alla ricezione del segnale **SIGUSR1, o** quando il Comm. Thread trova il **socket del Client chiuso, o** quando riceve l'**apposita operazione** di terminazione della connessione **da parte del Client**, il Communication Thread chiama la routine di chiusura della connessione con il Client.

Durante la routine della chiusura della connessione con il Client, il Comm. Thread esegue queste operazioni:

1. Segnala il suo slot come libero sulla bitmask.
2. Segnala un nuovo slot libero sul semaforo degli slot.
3. Chiude la connessione con il Client.
4. Termina (solo il Comm. Thread).

Esecuzione delle azioni

Le azioni implementate dal Server sono: leggere tutti i messaggi inviati, inviare un messaggio, eliminare un messaggio.
I Comm. Threads eseguono le azioni dopo aver ricevuto una richiesta da parte del Client tramite un'operazione.

È importante sottolineare la **differenza tra richiesta rifiutata e routine che termina NON correttamente**.
Una richiesta del Client può essere rifiutata quando non rispetta determinati parametri, come l'eliminazione di un messaggio postato da un altro utente. Il rifiuto però non comporta la disconnessione dal Client e la terminazione del Comm. Thread, ma il Comm. Thread torna al suo Main Cycle.
Una richiesta, o una routine, che termina NON correttamente, invece indica che non è possibile eseguire altre richieste, ad esempio perché il Client è irraggiungibile. Allora il Comm. Thread esegue la routine di chiusura della connessione.

Gli step che prevedono l'invio di operazioni o la loro ricezione possono fallire. Questo accade **se il canale di comunicazione è stato interrotto**, o per la terminazione del Client o per un problema alla rete.
Quando gli invii e le ricezioni falliscono la routine che il Comm. Thread stava eseguendo viene considerata come terminata **NON correttamente**.

Registrazione di uno nuovo utente

La routine di registrazione di un nuovo utente viene avviata quando un Comm. Thread riceve le credenziali dell'utente e la richiesta di registrazione:

Durante la registrazione, il Comm. Thread esegue questi step:

1. Blocca letture e scritture, tramite semafori, sul file in cui verranno salvate le credenziali dell'utente.
2. Cerca fra le credenziali salvate se è presente un'altro utente con lo stesso username dell'utente che cerca di registrarsi.
3. Se un utente con lo stesso username viene trovato, sblocca letture e scritture sul file degli utenti, ed invia al Client un'operazione che indica che la registrazione NON è andata a buon fine. Il Comm. Thread esegue la routine di chiusura della connessione.
4. Se non viene trovato nessun utente con lo stesso username, calcola l'UID per il nuovo utente. Il nuovo UID viene calcolato incrementando di 1 il precedente UID più alto, e se è il primo utente a registrarsi viene assegnato un UID definito come *INITIAL_UID* in *server.h*.
5. Salva le credenziali dell'utente sul file degli utenti. La password viene criptata.
6. Sblocca le letture e le scritture sul file degli utenti.
7. Invia un'operazione al Client indicando il successo della registrazione.
8. Comincia quindi il suo Main Cycle.

Login di un utente

La routine di registrazione di un nuovo utente viene avviata quando un Comm. Thread riceve le credenziali dell'utente e la richiesta di login:

Durante il login, il Comm. Thread esegue questi step:

1. Blocca le scritture sul file degli utenti.
2. Cerca fra le credenziali salvate, se è presente un'altro utente con lo stesso username dell'utente che cerca di registrarsi.
3. Sblocca le scritture sul file degli utenti.
4. Se nessun utente è presente con l'username richiesto dal login, il Comm. Thread invia al Client un'operazione che indica che il login NON è andato a buon fine. Il Comm. Thread esegue la routine di chiusura della connessione.
5. Se viene trovato un utente con l'username richiesto, il Comm. Thread decripta la password salvata e la confronta con quella inviata dal Client.
6. Se le password non coincidono, il Comm. Thread invia al Client un'operazione che indica che il login NON è andato a buon fine. Il Comm. Thread esegue la routine di chiusura della connessione.
7. Se le password coincidono, Il Comm. Thread invia un'operazione al Client indicando il successo della registrazione.
8. Comincia quindi il suo Main Cycle.

Salvare un messaggio inviato

La routine di salvataggio di un messaggio viene avviata quando un Comm. Thread, durante il suo Main Cycle, riceve da un Client un'operazione di invio messaggio. Il nome della funzione nel file *server-actions.c* è *post_message()*.

Durante la routine di salvataggio di un messaggio, il Comm. Thread esegue questi step:

1. Estrae dal testo dell'operazione: oggetto ed testo del messaggio.
2. Blocca letture e scritture sui file della bacheca.
3. Calcola il Message ID, che equivale al numero della riga del file degli indici in cui verranne scritte le informazioni per il salvataggio del messaggio.
4. Cerca aree di memoria libere all'interno del file dei messaggi abbastanza grandi da ospitare il messaggio appena ricevuto. Se viene trovata un'area di memoria libera abbastanza grande, allora il messaggio verrà salvato in quell'area, e le informazioni dell'area di memoria libera verranno aggiornate. Nel File delle Aree Libere viene mantenuta per ogni area di memoria libera: l'offset rispetto al File dei Messaggi e la lunghezza.
5. Scrive sul File degli Indici le informazioni del messaggio ricevuto: offset rispetto al File dei Messaggi, la lunghezza e il UID del mittente.
6. Scrive sul File dei messaggi l'oggetto ed il testo all'offset prima indicato.
7. Sblocca letture e scritture sui files della bacheca.
8. Manda un'operazione al Client che indica il successo dell'invio del messaggio.

Eliminare un messaggio ricevuto

La routine di eliminazione di un messaggio viene avviata quando un Comm. Thread, durante il suo Main Cycle, riceve un'operazione di richiesta di eliminazione di un messaggio. Il nome della funzione nel file *server-actions.c* è *delete_message()*.

Durante la routine di eliminazione di un messaggio, il Comm. Thread esegue questi step:

1. Blocca le letture e le scritture sui file della bacheca.
2. Controlla se il messaggio richiesto esista.
3. Se il messaggio non esiste, il Comm. Thread invia un'operazione che indica il rifiuto della richiesta. Passa allo step 12.
4. Legge dal File degli Indici le informazioni del messaggio richiesto: l'offset rispetto al File dei Messaggi, la lunghezza e l'UID del mittente.
5. Se l'utente che ha fatto la richiesta di eliminazione è diverso dal mittente del messaggio, allora il Comm. Thread invia un'operazione che indica il rifiuto della richiesta. Passa allo step 12.
6. Se il messaggio richiesto è già stato eliminato, allora il Comm. Thread invia un'operazione che indica il rifiuto della richiesta. Passa allo step 12.
7. Il Comm. Thread aggiorna l'offset del messaggio richiesto, sostituendolo con un offset che indica che il messaggio è stato eliminato. Il *DELETED_OFFSET* utilizzato è *0xffffffffffffffff*.
8. Cerca tra le aree di memoria libera se esistono aree di memoria adiacenti al messaggio che deve essere eliminato.
9. Se esistono tali aree di memoria libera, allora il Comm. Thread aggiorna l'offset dell'area di memoria libera già esistente, effettuando un'operazione di *merging*.
10. Se NON esistono tali aree di memoria libera, allora il Comm. Thread aggiunge una nuova area libera, che viene ottenuta eliminando il messaggio richiesto.
11. Il Comm. Thread NON elimina il contenuto del messaggio dal file dei contenuti.
12. Sblocca le letture e le scritture sui file della bacheca.
13. Se nessuna operazione che indicava il rifiuto della richiesta è stata inviata, allora invia un'operazione che indica il successo della richiesta.

Inviare tutti i messaggi

La routine di invio di tutti i messaggi viene avviata quando un Comm. Thread, durante il suo Main Cycle, riceve un'operazione di richiesta di lettura di tutti i messaggi. Il nome della funzione nel file *server-actions.c read_all_messages()*.

Durante la routine di invio di tutti i messaggi, il Comm. Thread esegue questi step:

1. Blocca le scritture sui file della bacheca.
2. Esegue un ciclo attraverso tutti i messaggi salvati.
   Per ogni messaggio:
   a. Legge dal File degli Indici alla riga che corrisponde al MID del messaggio: l'offset rispetto al File dei Messaggi, la lunghezza del messaggio e l'UID del mittente.

    b. Legge dal File dei Messaggi, all’offset ottenuto prima, l’oggetto ed il testo del messaggio.
    c. Converte l’UID del mittente con l’username.
    d. Invia quindi un'operazione che indica l’invio di un messaggio dal Server al Client, contenente username del mittente, oggetto e testo.
3. Terminato il ciclo attraverso i messaggi, Il Comm. Thread sblocca le scritture sui file della bacheca.
4. Invia un’operazione che indica il successo della richiesta.

File in server/

server.h

Il file è un header condiviso fra tutti i file .c del Server. Include anche *common.h* e *pthread.h*.
Contiene:

1. I nomi dei file adibiti a contenere gli utenti ed i messaggi
2. Le Macro per aprire i file
3. Costanti
4. Dichiarazione dei semafori condivisi fra i thread
    a. *UW* e *UR*, servono per scrivere e leggere sul file degli utenti, secondo la regola di single writer multiple reader
    b. *MW* e *MR*, servono per scrivere e leggere sui file dei messaggi, secondo la regola di single write multiple reader
    c. *sem_free_threads*, serve per tenere il conto di quanti thread sono liberi e quindi a cui può essere assegnata una comunicazione con un client.
5. Dichiarazione della bitmask adibita a tenere traccia di quali thread sono liberi e quali sono occupati.
6. Dichiarazioni delle maschere di segnali.
7. Prototipi delle funzioni che implementano le azioni del Server.
8. Prototipi di funzioni utili condivise fra i file .c del Server.

server-main.c, server-thread.c, server-log-reg.c, server-actions.c

I file contengono l'implementazione delle varie funzioni del Server.
Nello specifico:

- *server-main.c* : implementazione del Main Thread e della gestione dei segnali.
- *server-thread.c* : implementazione dei Communication threads.
- *server-log-reg.c* : implementazione del Login e della registrazione.
- *server-actions.c* : implementazione delle azioni.

Scelte riguardanti il Client

Single thread

Il Client è un'applicazione single thread, e quindi decisamente più semplice rispetto al Server.

La scelta è stata presa proprio perché il Client non necessita di più thread per la maggior parte delle funzioni, anche se alcune *features quality of life* avrebbero giovato di un secondo thread.

Segnali

Il segnale di cui è stata modificata la gestione è il **SIGINT**.

Il segnale SIGINT, viene usato per indicare al Client di interrompere ciò che stava facendo, disconnettersi dal Server e terminare. Il SIGINT viene **bloccato solo durante l'invio di operazioni**, così da garantire il corretto e completo invio.

Il **gestore del segnale** è molto semplice:

1. Chiude la connessione con il Server.
2. Termina l'applicazione.

Main Cycle del Client

l comportamento del Client è il seguente:
Durante l'esecuzione, il Client esegue questi step:

1. Si connette al Server.
2. Esegue la routine di Login o Registrazione.
3. Se l'accesso fallisce, ad esempio per credenziali non corrette, il Client chiude l'attuale connessione e torna allo Step 1.
4. Comincia il suo Main Cycle:
    a. Chiede all'utente quale azione vuole eseguire ed i parametri necessari per eseguirla.
    b. Invia un'operazione al Server e attende eventuali operazioni di risposta.
    c. Se l'azione termina correttamente, allora torna allo step 4. Altrimenti termina.

Il Client termina se l'utente sceglie l'azione che indica la disconnessione dal Server, se riceve un segnale *SIGINT*, se riscontra una terminazione del Server o se un'azione riscontra un errore critico per cui non è possibile proseguire.

Esecuzione delle azioni da parte del Client

Login e Registrazione

A differenza del Server, le procedure di richiesta di Login e Registrazione da parte del Client sono praticamente identiche. La discriminante fra i due tipi di richiesta è il codice dell'operazione che viene inviata al Server.

Durante la procedura di Login o Registrazione, il Client esegue questi step:

1. Manda l'operazione per cominciare l'handshake e attende la risposta del Server. **L'handshake** è necessaria per assicurare che l'utente inserisca le proprie credenziali solo nel momento in cui c'è uno slot libero pronto ad accogliere la richiesta.
2. Chiede all'utente le credenziali per l'accesso e la modalità: registrazione o login.
3. Invia un'operazione contenente la modalità di accesso e le credenziali.
4. Attende la risposta. Il codice dell'operazione ricevuta indica il successo o il fallimento dell'accesso. Se il codice è NOT ACCEPTED, è annesso anche il testo che indica il motivo del fallimento.
5. L'esecuzione continua dal punto 3. del Main Thread.

Inviare un messaggio

La routine per inviare un messaggio viene avviata quando l'utente seleziona l'apposita opzione. Il nome della funzione in *client-actions.c* è *post_message()*.

Durante la routine di invio di un messaggio, il Client esegue questi step:

1. Chiede all'utente di inserire l'oggetto ed il testo del messaggio.
2. Invia un'operazione con il codice MSG, contenente l'oggetto ed il testo del messaggio.
3. Attende la risposta del server. Il codice dell'operazione ricevuta indica il successo, codice OK, o il rifiuto, codice NOT ACCEPTED dell'azione. In entrambi i casi l'azione termina **correttamente**.
4. Se il codice ricevuto è diverso da OK e NOT ACCEPTED, allora l'azione termina **NON correttamente**.

Eliminare un messaggio

La routine per inviare un messaggio viene avviata quando l'utente seleziona l'apposita opzione. Il nome della funzione in *client-actions.c* è *delete_message()*.

Durante la routine di eliminazione di un messaggio, il Client esegue questi step:

1. Chiede all'utente quale messaggio vuole eliminare.
2. Invia un'operazione al server con il codice DELETE REQUEST e contente il MID del messaggio da eliminare.
3. Attende la risposta del server. Il codice dell'operazione ricevuta indica il successo, codice OK, o il rifiuto, codice NOT ACCEPTED dell'azione. In entrambi i casi l'azione termina **correttamente**.

4. Se il codice ricevuto è diverso da OK e NOT ACCEPTED, allora l'azione termina **NON correttamente**.

Leggere tutti i messaggi

La routine per leggere tutti i messaggi viene avviata quando l'utente seleziona l'apposita opzione. Il nome della funzione in *client-actions.c* è *read_all_messages()*.

Durante la routine di lettura dei messaggi, il Client esegue questi step:

1. Invia un'operazione con il codice READ REQUEST.
2. Riceve delle risposte dal server. Il codice della risposta indica se l'operazione inviata contiene un messaggio da stampare, indica la fine dei messaggi, o indica qualche tipo di errore.
3. Il Client continua a stampare i messaggi finché non riceve un'operazione con codice OK, che indica la fine dei messaggi.
4. Se l'ultima operazione ricevuta ha codice OK o NOT ACCEPTED allora l'azione termina **correttamente**. Altrimenti l'azione termina **NON correttamente**.

File in client/

client.h

Il file è un header condiviso fra tutti i file .c del Client. Include anche *common.h*.
Contiene:

1. Dichiarazioni delle maschere di segnali.
2. Prototipi delle funzioni che implementano le azioni del Client.
3. Prototipi di funzioni utili condivise fra i file .c del Client.

client-main.c, client-log-reg.c, client-actions.c

I files contengono l'implementazione delle varie funzioni del Client.
Nello specifico:

- *client-main.c* : implementazione del Main Thread e della gestione dei segnali.
- *client-log-reg.c* : implementazione del Login e della registrazione.
- *client-actions.c* : implementazione delle azioni.

Breve manuale d'uso dei programmi

Git Repository

Il progetto è interamente presente su GitHub. Il nome della repository è **SO-message-board** e l'autore è **gmarseglia**.
Il link è : https://github.com/gmarseglia/SO-message-board

Per scaricarlo è sufficiente usare il comando *git clone* seguito dall'URL apposito.
git clone https://github.com/gmarseglia/SO-message-board.git

Come compilare

All'interno della cartella *source/* è presente un Makefile.
Per compilare basta usare *'make'*. Verrà generata una cartella *'runnable'* al cui interno ci saranno gli eseguibili del Server e del Client.
Altri target del makefile sono *'debug'* e *'fine'*, utilizzati in fase di sviluppo per stampare maggiori informazioni a schermo e connettere velocemente Server e Client.
Infine *'make clear'* permette di eliminare tutti gli eseguibili e file creati.

Come usare

Informazioni riguardo ai parametri del Server e del Client possono essere ottenute con *'xserver --help'* e *'xclient --help'*.
CTRL + C per terminare il Server o il Client.
Assicurarsi di eseguire il Server sempre dalla stessa *Current working directory*.