Selmi Mohamed Aziz 0322027.

Introduzione al progetto.

Il progetto realizzato consiste in una bacheca elettronica, che permette agli utenti in seguito all'autenticazione di: visualizzare i messaggi presenti nella bacheca, sia quelli propri ma anche quelli emessi da altri utenti; inserire un nuovo messaggio all'interno della bacheca; eliminare un proprio messaggio presene all'interno della bacheca. Informazioni più accurate sulla realizzazione delle varie parti saranno fornite in seguito.

Indice:

Sezione 1: Organizzazione del progetto.

All'interno della directory electronic-bulletin-board sono presenti i file del progetto divisi in tre sottodirectory: src, include e data.

In src sono presenti i codici sorgenti; in include i vari header file; in data invece sono presenti file csv utilizzati per la permanenza dei dati dal server una volta che si va in shutdown.

Sezione 2: Struttura del codice.

Il codice applicativo della parte server si trova in server.c e functionServer.c .

Il codice applicativo della parte client si trova in client.c e functionClient.c .

L'implementazione del progetto è realizzata attraverso una divisone delle responsabilità. Sono state create diverse librerie con compiti specifici, informazioni a riguardo in seguito.

fileMessageLib.h: Libreria per la gestione dei file per la permanenza dei dati.

messageLib.h: Libreria per l'utilizzo della struttura dati bacheca in memoria centrale.

mutexLib.h: U: Libreria per la gestione dei mutex.

helper.h: Libreria per operazioni comuni come presa di dati da stdin o altro.

protocolUtilis.h: Libreria per le funzioni di sincronizzazioni tra server e client.

Sezione 3: Istruzioni d'utilizzo.

Server:

Per compilare il server, utilizzare il seguente comando: make server.

Per avviare il server, utilizzare il comando: make run-server.

Client:

Per compilare il client, utilizzare il seguente comando: make client.

Per avviare il client, utilizzare il comando: make run-client.

Utilizzo:

Una volta che il client ha stabilito la connessione con il server, viene invitato a scegliere tra la registrazione o l'accesso ad un account già esistente attraverso username e password (per maggiori informazioni a riguardo guardare sezione 4: Autenticazione). Eseguito l'accesso correttamente è possibile scegliere tra le seguenti cinque attività:

1. Postare un messaggio all'interno della bacheca.
2. Visualizzare i propri messaggi.
3. Visualizzare tutti i messaggi presenti in bacheca.
4. Eliminare un proprio messaggio.
5. Terminare la connessione con il server.

Sezione 3: Protocollo e sincronizzazione.

Il server e il client comunicano tramite il protocollo TCP e, per la sincronizzazione, si scambiano comandi. Ogni comando è un valore esadecimale che rientra in un singolo byte e viene interpretato come char all'interno dell'applicazioone. Sono stati definiti esattamente 15 comandi, dichiarati come macro nel file commServerClientConfig.h, utilizzato sia dal server che dal client.

Comandi:

I comandi svolgono funzioni specifiche nel protocollo di comunicazione: COMMAND_AUTH (0x01) viene mandato dal server al client per avviare il processo di autenticazione. COMMAND_LOG (0x02) e COMMAND_SUB (0x03) sono mandati dal client al server per richiedere rispettivamente l'accesso o la registrazione di un nuovo utente. Gli errori durante l'autenticazione vengono segnalati dal server al client con COMMAND_ERR_USER_NOT_FOUND (0x04), che indica che l'utente non esiste nel sistema, e con COMMAND_ERR_NOT_MATCH_CREDENTIALS (0x05), che segnala credenziali non valide. COMMAND_ERR_USER_ALREADY_EXISTS (0x06) viene mandato dal server al client per indicare che il nome utente scelto per la registrazione è già in uso.

Gli esiti delle operazioni sono comunicati dal server al client con COMMAND_SUCCESS (0x07), per indicare un'operazione riuscita, e con COMMAND_FAILURE (0x08), per segnalare un fallimento. COMMAND_CLOSE (0x09) viene mandato dal server al client quando il server riceve dati o comandi non conformi alle aspettative e decide di chiudere la connessione.

Per la gestione dei messaggi, il client manda al server COMMAND_POST_MSG (0x0A) per pubblicare un nuovo messaggio, COMMAND_VIEW_MSG (0x0B) per visualizzare i propri messaggi, e COMMAND_VIEW_ALL_MSG (0x0C) per recuperare tutti i messaggi presenti nella bacheca. COMMAND_DELETE_MSG (0x0D) viene mandato dal client al server per richiedere l'eliminazione di un messaggio specifico.

COMMAND_QUIT (0x0E) viene mandato dal client al server per terminare la connessione in modo controllato. Infine, COMMAND_FINISH_ATTEMPTS (0x0F) viene mandato dal server al client per indicare che si è superato il numero di tentativi di autentificazione.

Funzioni:

Le funzioni readCom e writeCom sono fornite dal file di intestazione protocolUtils.h e gestiscono la comunicazione tra client e server attraverso la socket.

La funzione int readCom(int socket, char *command); ha la responsabilità di leggere un singolo byte dalla socket e memorizzarlo nella variabile puntata da command. Questa funzione viene utilizzata per ricevere i comandi inviati dall'altra parte della connessione.

La funzione int writeCom(int socket, char command); si occupa invece di inviare un singolo byte attraverso la socket. Viene utilizzata per trasmettere un comando al destinatario.

Nel file protocolUtils.h sono presenti le funzioni writeBuffSocket e readBuffSocket, che permettono rispettivamente di scrivere e leggere un numero specifico di byte tramite la socket. Per garantire che server e client siano allineati riguardo al numero di byte da attendere ed emettere, ho deciso di inviare sempre il numero massimo di byte per ogni campo, utilizzando il padding nei casi in cui il campo non raggiunga la lunghezza effettiva. Una volta ricevuto lo stream di byte, il destinatario dovrà effettuare l'operazione di rimozione del padding.

Il server dispone anche di una funzione di lettura, readTimeout, che esegue una lettura con un timeout specificato, verificando che errno sia impostato su EAGAIN quando la funzione read restituisce –1, in modo da evitare che il server rimanga in attesa indefinitamente che il client invii dei dati.

Il timer per la socket è configurato tramite la funzione receiveTimeout, presente nel file server.c.

Sezione 4: Autenticazione.

Gli utenti possono accedere alla bacheca elettronica solamente se sono registrati all'interno di essa, ogni utente è identificato all'interno del sistema attraverso uno username e una password. Il campo username deve essere almeno superiore di quattro caratteri fino ad un massimo definito nella macro SIZE_USERNAME, attualmente configurata a sessantaquattro. La password invece deve contenere un numero di caratteri da otto fino ad un massimo definito nella macro SIZE_PASSWORD, attualmente configurata a sessantaquattro, la password deve contenere almeno una lettera maiuscola, un carattere speciale e un numero.

Nota: Le macro SIZE_USERNAME e SIZE_PASSWORD, sono definite nell ’header file commFieldsConfig.h, utilizzato dal server e dal client.

Sezione 5: Bacheca in memoria centrale.

La bacheca è organizzata in memoria centrale a partire dalla struttura BulletinBoard, che rappresenta l'entità principale del sistema. La struttura BulletinBoard contiene un puntatore a una lista di utenti, che consente di gestire e navigare tra tutti gli utenti registrati sulla bacheca. Inoltre, include due contatori: msgCount, che tiene traccia del numero totale di messaggi presenti sulla bacheca, e idCount, che gestisce l'assegnazione di identificativi univoci ai nuovi messaggi.

Ogni utente è rappresentato dalla struttura User, che contiene informazioni cruciali come l'username e la password per l'autenticazione, insieme a un contatore count che tiene traccia del numero di messaggi dell'utente presenti nella bacheca. La struttura User ha anche un campo messages, che è un array dinamico di puntatori a strutture Message, permettendo a ciascun utente di avere più messaggi associati a sé. Infine, il campo next della struttura User funge da collegamento a un altro utente, formando una lista collegata di utenti nella bacheca.

La struttura Message descrive un singolo messaggio pubblicato sulla bacheca e include vari campi: object, che memorizza l'oggetto del messaggio; text, che contiene il corpo del messaggio; idMessage, che è un identificatore univoco per ogni messaggio.

Ogni volta che il server viene eseguito, la bacheca viene ricostruita dai file data_messages.csv e data_users.csv. (Per maggiori informazioni a riguardo si veda Sezione 7: Gestione e permanenza dei dati.)

La dichiarazione dellaestrutture e delle funzioni operanti su esse sono presentii nel header file messageLib.h, l'implementazione in messageLib.c.

Le dimensioni dei vari campi sono dichiarate nel header file messageLib.h.

Sezione 7: Permanenza dei dati.

In modo da garantire la permanenza dei dati in seguito allo shutdown del server si è fatto utilizzo di due file csv: data_users.csv, contenente gli username e le password degli utenti che si sono segnati alla bacheca; data_messages.csv, contenente i messaggi degli utenti, nell'ultimo campo di ogni messaggio vi è un valore numerico utilizzato per indicare se il messaggio è stato cancellato o meno. In pratica ogni qualvolta che un utente chiede la cancellazione di un messaggio, quest'ultimo non viene cancellato direttamente ma viene cambiato l'ultimo campo del messaggio nel file csv, si è scelto di percorrere questa strada in quanto la cancellazione di un messaggio è un'operazione costosa, in quanto si deve ricostruire tutto il facendo utilizzo di un file secondario. Tale azione viene fatta dopo un lasso di tempo dal main thread e non ad ogni cancellazione, in tal modo si effettuano più cancellazioni fisiche contemporaneamente riducendo i costi. Per realizzare quanto detto si è fatto utilizzo del segnale SIGALARM, assegnandoli un apposito gestore (per maggiori informazioni a riguardo consultare Sezione 9: Gestione dei segnali).

Sezione 8: Gestione delle zone critiche.

Per evitare che più thread accedano contemporaneamente in modo errato in lettura o scrittura, si è fatto utilizzo di mutex, sono stati utilizzati per l'esattezza tre mutex, uno per la struttura bacheca in memoria, e due per i due file utilizzati per la permanenza: data_users.csv e data_messages.csv (per maggiori informazioni consultare Sezione 7: Permanenza dei dati.

È stata realizzata una libreria per la gestione dei mutex, in modo da gestire più efficientemente il loro utilizzo, le funzioni sono fornite dall'header file mutexLib.h .

Sezione 9: Gestione dei segnali.

Segnali che si è scelto di gestire: SIGINT, SIGALARM, SIGPIPE.

Da parte del server si è deciso di ignorare esplicitamente il segnale SIGPIPE, in quanto se il server si trovasse nello scenario di scrittura su una socket broken, esso non termini la sua esecuzione, ma bensì quando una write restituisca –1 si va a controllare se errno sia impostato ad EPIPE, riconoscendo la chiusura da parte del client, il server termina l'esecuzione del particolare thread nella maniera corretta senza compromettere l'intera applicazione. Il segnale SIGINT è stato catturato associandoli un apposito gestore (handlersignt implementato nel file server.c) che prevede la chiusura in maniera controllata dell'intera applicazione, chiudendo tutti i file descriptor e terminando in seguito l'applicazione.

Si è scelto di fare utilizzo di una sveglia periodica per la cancellazione di tutti i messaggi di cui si è chiesta la cancellazione (per maggiori informazioni a riguardo Sezione 7: Permanenza dei dati). Pertanto, al segnale SIGALARM è stato assegnato un gestore, periodicFunction la cui implementazione e gestione risiede nel file server.c .