Tesina Sistemi Operativi

FLAVIO SIMONELLI

MAT: 0310059

Progetto: Servizio di Messaggistica in UNIX

Specifiche del Progetto

Realizzazione di un servizio di scambio messaggi supportato tramite un server sequenziale o concorrente (a scelta).
Il servizio deve accettare messaggi provenienti da client (ospitati in generale su macchine distinte da quella dove risiede il server) ed archiviarli.

L'applicazione client deve fornire ad un utente le seguenti funzioni:
1. Lettura tutti i messaggi spediti all'utente.
2. Spedizione di un nuovo messaggio a uno qualunque degli utenti del sistema.
3. Cancellare dei messaggi ricevuti dall'utente.

Un messaggio deve contenere almeno i campi Destinatario, Oggetto e Testo.
Si precisa che la specifica prevede la realizzazione sia dell'applicazione client che di quella server.
Inoltre, il servizio potrà essere utilizzato solo da utenti autorizzati (deve essere quindi previsto un meccanismo di autenticazione).

Discussione delle scelte progettuali, delle tecniche e metodologie usate

Il presente progetto si propone di sviluppare un'applicazione di messaggistica distribuita per sistemi operativi UNIX, composta da un client e un server.

L'obiettivo principale di questo sistema è mettere in comunicazione gli utenti autorizzati, consentendo loro di scambiarsi messaggi attraverso un server centrale. Il server sarà concorrente e responsabile della gestione dei messaggi e delle credenziali degli utenti, che verranno memorizzate in file sulla memoria secondaria.

Gestione Multi-Thread e Accesso alle Strutture Dati Condivise nel Server:

La gestione multi-thread è fondamentale per il corretto funzionamento del server di messaggistica. Il thread principale è responsabile dell'inizializzazione e della gestione delle strutture dati, della configurazione della socket, dell'inizializzazione della libreria di criptazione e dell'inizializzazione della gestione dei segnali. Una volta completate queste operazioni, entra in uno stato di ascolto, pronto a ricevere nuove connessioni da parte dei client.

Appena un client si connette, il server attiva un nuovo thread dedicato a gestire le interazioni con quel particolare client. Questo approccio multi-threading consente al server di gestire simultaneamente più connessioni senza compromettere le prestazioni. Ogni thread associato a un client gestisce in modo indipendente le richieste provenienti da quel client specifico, migliorando l'efficienza complessiva del sistema.

Terminata la creazione del nuovo thread, il thread principale rientra in uno stato di ascolto garantendo una risposta immediata a eventuali richieste in arrivo da altri client.

L'accesso alle strutture dati condivise è possibile grazie alla struttura Sem, che dispone di un mutex per la scrittura e un semaforo per la lettura. Quando un determinato thread vuole leggere, controlla che sia presente il mutex in scrittura e poi aggiunge un token al semaforo delle letture. Un thread che vuole invece scrivere deve bloccare il mutex di scrittura e poi aspettare che non ci siano più token all'interno del semaforo delle letture (ciò significa che non sono presenti più thread lettori e che quindi la struttura dati è disponibile per la scrittura).

Approccio Single-Thread nel Client:

Nel contesto del client, l'approccio single-thread è stato ritenuto più adeguato. Il client, una volta avviato, gestisce le sue operazioni in modo sequenziale, interagendo con l'utente e inviando/ricevendo messaggi dal server.

L'implementazione single-thread semplifica notevolmente la gestione delle operazioni del client, evitando la complessità associata all'implementazione di concorrenza per thread multipli. Questa scelta di progettazione è spesso sufficiente per applicazioni di messaggistica leggere in cui il client è principalmente impegnato nell'interazione con l'utente e nell'invio/ricezione di messaggi, come in questo caso.

Inoltre, l'approccio single-thread semplifica anche il debug e la manutenzione del codice, poiché non è necessario gestire la sincronizzazione tra thread o i problemi di race condition.

Funzionamento della Socket nel Sistema:

Il sistema di messaggistica implementato utilizza le socket per la comunicazione tra client e server. Le socket consentono ai processi di scambiarsi dati attraverso la rete, sfruttando le funzionalità offerte dal protocollo TCP. Il protocollo TCP è ampiamente utilizzato per applicazioni che richiedono una trasmissione affidabile e in ordine dei dati, come nel caso di un'applicazione di messaggistica. TCP fornisce anche il controllo di congestione e di flusso, che sono essenziali per garantire che la rete non venga sovraccaricata da una quantità eccessiva di dati.

Per far funzionare sia il client che il server, è necessario specificare all'avvio il numero di porta su cui il server ascolta le connessioni in arrivo. Se l'indirizzo IP è *null*, il sistema prende l'indirizzo IP della macchina su cui è in esecuzione il server.

Il server è configurato con una costante a tempo di compilazione, chiamata BACKLOG, che rappresenta il numero massimo di client che può gestire contemporaneamente. Quando il numero di client connessi supera il valore di BACKLOG, il server inizia a rifiutare nuove connessioni. Questo è un meccanismo di sicurezza per evitare il sovraccarico del sistema.

Criptazione dei dati tramite libreria esterna libsodium:

La sicurezza dei messaggi trasmessi all'interno della socket è una caratteristica indispensabile, e, per garantire un livello elevato di protezione, ho implementato un algoritmo a chiave mista utilizzando la libreria libsodium.

La mia scelta deriva dal fatto che:

- libsodium è stata sviluppata da esperti di crittografia ed è basata sulla ben consolidata libreria NaCl (Networking and Cryptography library). La sua robustezza è stata ampiamente testata e verificata.
- La libreria è progettata per essere resistente ad attacchi comuni, come attacchi temporizzati (side-channel attacks) e altre vulnerabilità crittografiche.

- libsodium è distribuita con una licenza libera (ISC license), che consente un'ampia adozione in progetti open source e commerciali.

Attraverso la libreria libsodim ho applicato l'algoritmo argon2 per le password.

Argon2 è progettato per essere resistente a una vasta gamma di attacchi, tra cui attacchi brute-force, attacchi basati su dizionari e attacchi di tipo side-channel. Questo lo rende una scelta popolare per la memorizzazione sicura delle password, poiché aiuta a proteggerle dagli attacchi informatici.

Per il trasferimento di dati fra client e server ho invece applicato un algoritmo a chiave asimmetrica.

La **crittografia asimmetrica** è un metodo crittografico che utilizza due chiavi distinte: una chiave pubblica e una chiave privata. La chiave pubblica è utilizzata per cifrare i dati, mentre la chiave privata viene utilizzata per decifrarli. In questo modo, chiunque possa accedere alla chiave pubblica può cifrare i dati, ma solo il possessore della chiave privata può decifrarli.

Il processo di crittografia asimmetrica avviene in tre fasi principali:

- Generazione delle Chiavi: Inizialmente, il destinatario della comunicazione genera una coppia di chiavi: una chiave pubblica e una chiave privata. La chiave pubblica viene distribuita al mittente, mentre la chiave privata viene mantenuta segreta dal destinatario.
- Cifratura: Il mittente utilizza la chiave pubblica del destinatario per cifrare i dati. Una volta cifrati i dati, possono essere inviati al destinatario.
- Decifratura: Il destinatario utilizza la sua chiave privata per decifrare i dati.

La crittografia asimmetrica offre diversi vantaggi rispetto a quella simmetrica, in cui viene utilizzata la stessa chiave per cifrare e decifrare i dati. Ad esempio, semplifica la gestione delle chiavi (poiché la chiave pubblica può essere distribuita a chiunque), protegge le chiavi private (non vengono mai trasmesse sulla rete). Tuttavia, la crittografia asimmetrica è generalmente più lenta rispetto alla crittografia simmetrica e richiede più risorse computazionali.

Strutture Dati utilizzate:

Per la gestione dei dati, ho implementato una struttura basata su **tabella hash** sia per la gestione degli utenti che per la gestione delle chat.

La scelta di una tabella hash come struttura dati per la gestione dei dati è motivata da diversi fattori, tra cui l'efficienza delle operazioni e la velocità di accesso ai dati:

- Efficienza delle Operazioni: Le tabelle hash offrono un'implementazione efficiente per molte operazioni fondamentali, come l'inserimento, la ricerca e la rimozione di elementi. Il tutto si basa sull'utilizzo di una funzione di hash che individua in quale riga della tabella operare. L'inserimento usato nel server è quello in testa, poiché ha il costo computazionale minore (O(1)). Per quanto riguarda la ricerca e la rimozione, il costo dipende molto dal numero di nodi all'interno della lista di trabocco. Questa lista può essere più o meno grande in base al numero di righe della tabella hash, in entrambe le operazioni il caso peggiore rimane O(n).
- Spazio di Memorizzazione: Le tabelle hash richiedono meno spazio di memorizzazione rispetto ad altre strutture dati, come gli alberi, poiché non è necessario memorizzare i puntatori ai nodi figlio. Questo rende le tabelle hash adatte per applicazioni in cui lo spazio di memorizzazione è limitato.
  Inoltre, sono stati utilizzati nodi Chat e User che contengono solo il minimo indispensabile per identificarli e la posizione dei dati nel file specifico.

- Flessibilità: Le tabelle hash possono essere facilmente ridimensionate dinamicamente, in modo da adattarsi alle esigenze di memorizzazione dei dati. Questo le rende adatte per applicazioni in cui la quantità di dati può variare nel tempo.

Archiviazione dei dati nella memoria secondaria:

Per mantenere una coerenza dei dati anche quando il server viene chiuso (in modo corretto o anomalo) ho scelto di **salvare tutti i dati all'interno dei file** così nominati:

logCred.txt = per il file relativo a utente e password

logChats.txt = per il file relativo alle chat correttamente istanziate

nomeutente-nomeutente.txt = per i file chat contenente i messaggi scambiati dai due utenti

I primi due file vengono "puliti" dalle righe invalidate (con bit validate = 0) durante l'inizializzazione del server.

I file Chat invece vengono "puliti" ogni volta che un utente vi accede.

La soluzione con bit di validità è stata implementata, per una questione di efficienza computazionale nell'ambito dell'eliminazione dei dati, durante l'esecuzione del server. In particolare, questa soluzione è stata adottata per evitare la creazione di un overhead e il rischio di deadlock che si sarebbe potuto verificare se la procedura di eliminazione dei dati fosse stata implementata ogni qual volta sarebbe stata richiesta l'eliminazione di un dato.

Gestione dei segnali

Il server è stato progettato con una gestione avanzata dei segnali per garantire la robustezza e la coerenza dei dati durante il suo funzionamento.

In particolare, il server blocca tutti i segnali in entrata, assicurando un controllo completo sulle azioni da intraprendere in risposta a segnali specifici. Nel caso di un'interruzione del processo da parte dell'utente tramite **SIGINT** (ad esempio, con Ctrl+C), il server è in grado di rilevare questo segnale e attivare una procedura che permette di chiudere la socket in maniera ordinata, garantendo una corretta terminazione del processo. Inoltre, per garantire che la chiusura del terminale non comporti la chiusura inattesa del server, è stato implementato un meccanismo per gestire il segnale **SIGCLOSE**. In questo modo, il server continua a funzionare in modo stabile anche se il terminale viene chiuso, evitando interruzioni indesiderate e fornendo una maggiore affidabilità nell'esecuzione delle sue funzionalità.

Questa attenta gestione dei segnali contribuisce a preservare l'integrità dei dati e a garantire una chiusura controllata del server, mantenendo la coerenza del sistema anche in situazioni inaspettate.

Per quanto riguarda il client è stato invece utilizzato proprio il segnale **SIGINT** per permettere la corretta chiusura dell'unico processo.

Breve manuale d'uso dei programmi (come compilare e come eseguire)

Le scelte progettuali precedentemente indicate hanno portato ad una semplice realizzazione di questa fase.

Compilazione

Per compilare vi è l'obbligo di aver preinstallato la libreria libsodium in mancanza il processo di compilazione è destinata al fallimento.

Dopo l'installazione della libreria è necessario aprire il terminale nella cartella in cui sono presenti i file sorgenti e, per mezzo del makefile, utilizzare i comandi:

- make server
- make client

Verranno creati due file eseguibili omonimi.

Utilizzo del server

L'utilizzo del server è banale poiché una volta avviato sarà tutto automatico.

Per un avvio corretto occorre specificare a parametro la porta e l'indirizzo della socket; se l'indirizzo è il medesimo della macchina allora occorre specificare *null*.

esempio:

./server null 8080

Successivamente si avrà come unica opzione la chiusura attraverso il comando ***Ctrl+c*** .

Utilizzo del client

Per una corretta esecuzione, anche per il client, è indispensabile fornire gli stessi parametri forniti al server:

./client null 8080

Dopo l'invio del comando, il client ci notificherà la corretta connessione con il server.

Dopo la connessione con il server, il client propone le seguenti funzionalità:

- {0} Accedere
- {1} Registrarsi
- {2} Eliminare il proprio account (verranno eliminate tutte le chat relative all'utente)

Qualsiasi sia la scelta effettuata occorre successivamente autenticarsi con username e password.

Nelle ipotesi in cui si sia scelto di accedere o registrarsi il software proporrà una delle seguenti funzionalità:

- Inviare un messaggio
- Leggere una chat
- Eliminare un messaggio

Tutti i sorgenti del progetto

I file per eseguire correttamente il progetto sono:

- Makefile
- Client.c
- Client.h
- Server.c
- Server.h
- Inputuser.c
- Inputuser.h
- transfertsocket.c
- transfertsocket.h