Documentazione Progetto Reti Informatiche – Sansone Giacomo – 596397

Dopo che l'utente ha effettuato il login con il server, si trova la schermata di home nella quale può scegliere l'azione da eseguire. Digitando `chat username` si apre una chat con un utente, e sono prelevati dal file di memorizzazione i messaggi archiviati. All'invio del primo messaggio il device si mette in comunicazione con il server affinché questo memorizzi il messaggio se l'altro è offline o invii le informazioni per instaurare la connessione tra i due nel caso opposto. Sia server che device memorizzano le connessioni attive all'interno di una lista di `struct connessione`, nel quale si mantiene la porta di collegamento, il nome utente della connessione e il descrittore di socket. In questo modo non solo è immediato (con le funzioni di `gestoreConnessioni.c`) risalire alle informazioni sulla connessione, ma è possibile pure monitorare i *logout* degli utenti. Quando la `select` del *main loop* sveglia un socket e alla `recv` successiva in `connectionHandler` si riceve come valore di ritorno `0`, allora l'utente si è disconnesso: si chiude il descrittore di socket e si libera la memoria ad esso allocata. Tale funzionalità consente di avere costantemente coerente la lista delle connessioni con quelle effettivamente attive, senza necessità che il server cerchi di inviare più volte un messaggio ad un device per verificarne l'attività. Il registro degli accessi del server è di facile realizzazione. Il device può allo stesso modo gestire le disconnessioni degli altri dispositivi, inoltrando a quel punto le richieste di invio dei messaggi al server. Avrei potuto chiudere le connessioni al chiudere delle chat (comando `\q`), ma questo, oltre a creare overhead, impedisce l'uso della funzione `share` nel menu principale.

Ho usato sempre dei socket TCP bloccanti; la scelta del TCP è dovuta alla logica dell'applicazione: trovo irrealistico che un tale servizio perda messaggi, li alteri o risulti poco *resposive* nell'invio delle richieste al server. I socket sono poi bloccanti perché ci si aspetta che tutti si blocchino in attesa che l'altra parte (sia esso il device o il server) faccia qualcosa: un ciclo per verificarne l'attività sarebbe un'inutile sostituzione della `select` invece utilizzata. Il server è mono-processo: le problematiche di tale realizzazione sono tante, come il singolo *point of failure* e le possibili attese per i device. Allo stesso tempo, i processi avrebbero dovuto fare accessi concorrenti a strutture dati e risorse condivisi (i file), la cui gestione sarebbe risultata problematica. Un'alternativa che, piuttosto, sarebbe preferibile è quella di un *server multi-thread*, andando a sfruttare le funzionalità presenti nella libreria `pthread` per gestire tali mutue esclusioni.

Supponiamo che A e B abbiano una connessione, e A voglia aggiungere C online alla chat. Per prima cosa, A instaura la connessione con C, dopo di ché invia i dati di C a B sotto il comando di aggiunta al gruppo. A genera un *token* per il gruppo, che associa alle connessioni di B e C. B invia la richiesta di connessione a C, specificando in tal sede che A ne ha richiesto l'aggiunta al gruppo. In tal modo, anche B genera un *token* per A e C, e lo stesso fa C per A e B. La meccanica si ripete in modo identico per l'aggiunta del quarto membro. Quando a C arriva un messaggio da A, trova un *token* di gruppo associato alla connessione, e sa che quel messaggio fa parte del gruppo. La presenza del *token* (eventualmente diverso per i vari utenti) consente sia di raggruppare i partecipanti ad un gruppo, sia di salvare in memoria i messaggi (il destinatario sarà il *token* del gruppo) e visualizzarli successivamente dal file: usando come *token* il timestamp di generazione del gruppo, siamo certi che non ci siano collisioni. In più, è sempre possibile che un'utente appartenga a gruppi diversi (ma non che due utenti si trovino in due gruppi diversi contemporaneamente, essendo unico per utente l'identificatore del gruppo; per risolvere tale problema, basterebbe avere una lista di *token* di gruppi associati alle connessioni). Tale implementazione necessita di $\binom{N}{2}$ connessioni se N è il numero di partecipanti al gruppo, ma consente di semplificare la funzionalità di scambio dei file: quando A vuole mandare un file, scorre le sue connessioni, in cui sono presenti i membri dei gruppi, e invia a ciascuno il file.

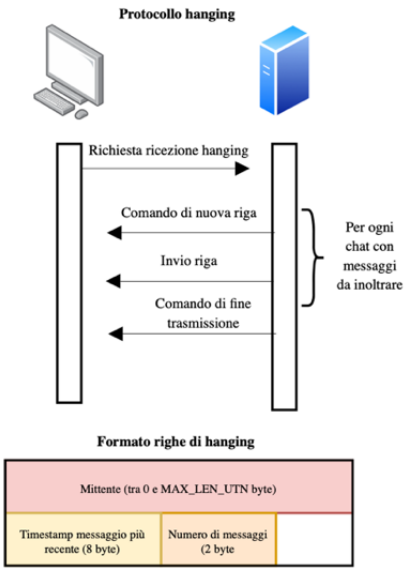

Figura 1

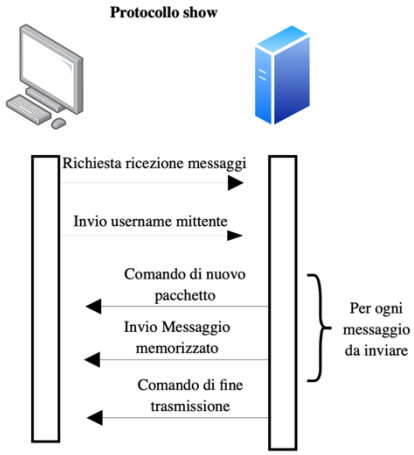

Figura 2

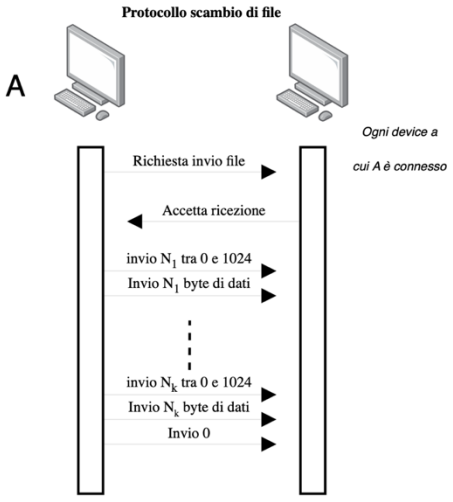

Figura 3

La funzionalità di `hanging` è implementata come in *figura 1*: il device fa una richiesta con un opportuno codice al server (tutte le richieste, di qualunque tipo esse siano, sono identificate da un codice specificato in `macro.h`); il server risponde inviano un opportuno codice di nuova riga e i dati successivi, con il quale il device potrà modificare la propria lista dei messaggi. Qualcosa di simile accade con il comando di `show` (*figura 2*). Nell'intera comunicazione tra le diverse parti ho adottato dei protocolli di tipo *binary*. Questo mi ha consentito da una parte di gestire meglio la fase di debug, dall'altra di ridurre il numero di byte inviati, specie quando ho dovuto scambiare dei *timestamp*. In questo modo, ogni volta che dovevo inviare una stringa, ho sempre preventivamente inviato anche il numero di caratteri che la costituivano su 2 byte.

Lo scambio dei file si realizza tramite un flooding dei dati dal mittente al destinatario (*figura 3*), dopo che il secondo ha acconsentito all'accettazione del file.

Un aspetto critico della mia implementazione è la gestione poco accurata degli errori che si possono verificare: piuttosto che gestirli singolarmente, magari dando nuovamente il via al protocollo in atto, preferisco interrompere l'esecuzione del processo.

Le disconnessioni improvvise dei device sono gestite automaticamente per come spiegato sopra. L'unica situazione problematica si ha quando il server è offline e la disconnessione è improvvisa. In tal caso, infatti, il device non salva l'ultimo *timestamp* di disconnessione, non permettendo di tenere aggiornato il registro del server.