Tesina di Sistemi Operativi

Autore : Simone Nicosanti

Matricola 0266910

Anno Accademico 2019 - 2020

Specifica della tesina

Servizio di messaggistica

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 di 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).

La tesina è stata sviluppata per sistemi Unix.

Discussione Generale

Il Protocollo scelto è TCP. Sebbene sia una scelta particolare, visto che si tratta di un servizio di scambio di messaggi, ho preferito usare questo protocollo per garantire affidabilità e ordine nello scambio di messaggi, sia quelli inviati dal client al server per richiedere un servizio, sia quelli inviati dal server al client per rispondere. Altro motivo che mi ha portato a questa scelta è stato il fatto che, avendo scelto TCP come protocollo, e quindi un modello di comunicazione stream, il socket atto a gestire la comunicazione con il client a seguito della connessione viene istanziato dinamicamente, senza bisogno di lavoro ulteriore.

Ho cercato di costruire il Client e il Server in maniera il più possibile speculare, di modo da facilitare la gestione delle due implementazioni.

readFromSocket e writeOnSocket

Le due funzioni principali per la comunicazione tra client e server sono la readFromSocket e la writeOnSocket.

*Void writeOnSocket(char *msgPtr , int totWrite , int sizeToWrite)*

La writeOnSocket è tale per cui prende come parametro il puntatore al messaggio da scrivere sul socket, il numero di bytes che è già stato scritto e la taglia totale del messaggio da scrivere.

La scrittura continua fino a quando totWrite < sizeToWrite e quindi fino a quando non ho scritto tutto il messaggio.

Per controllare un'eventuale chiusura del capo opposto della comunicazione, ho scelto di utilizzare la system call send con il flag NO_SIGNAL. Questo mi permette di non ricevere il segnale di SIGPIPE in caso di chiusura dell'altro capo, ma di avere comunque la variabile errno impostata a EPIPE. Nel caso in cui errno sia impostato ad EPIPE, invoco la funzione di chiusura della comunicazione del client o del server.

*Void readFromSocket(char *msgPtr , int totRead , int sizeToRead)*

La readFromSocket è speculare alla writeOnSocket.

Prende come parametro il puntatore alla locazione in cui andrò a scrivere il messaggio che sto leggendo, la taglia che ho già letto e la taglia totale del messaggio da leggere.

In questo caso, per rilevare la chiusura dell'altro capo della comunicazione uso la system call *poll*. Questa system call mi permette di chiedere informazioni riguardo lo stato di un oggetto di I/O. Nello specifico, chiedo informazione sul socket e sulla condizione di POLLRDHUP: questa condizione è verificata quando il peer del socket viene chiuso. Verificando se il numero di bytes letti è pari a zero, riesco a rilevare la condizione di fine dello stream: non ci sono più bytes da leggere e non ci sono dei peer che possono scriverli. Quando queste due condizioni si verificano insieme, invoco la funzione di chiusura della comunicazione del client o del server, in base al processo in cui mi trovo.

Tipi di Messaggi

Ci sono sei tipi di messaggi.

- ANSWER_MESSAGE
  - Si tratta del messaggio che il Server invia al Client come risposta al servizio richiesto.
  - Tipi di risposta:
    - POS_TYPE : Servizio svolto correttamente
    - NEG_TYPE_0 : Errore lato server per lo svolgimento del servizio
    - NEG_TYPE_1 e NEG_TYPE_2 : Altro tipo di errore nell'esecuzione del servizio

- LOGIN_MESSAGE
  - Messaggio che Client invia al Server per richiedere iscrizione o accesso al servizio di messaggistica
  - Campi :
    - userName : username utente – Massimo USER_NAME_MAX_SIZE caratteri
    - passwd : password inviata dall'utente – Massimo PASSWD_MAX_SIZE caratteri

- EXIT_MESSAGE
  - Messaggio che Client invia al Server per richiedere il logout dal servizio

- SEND_MESSAGE
  - Messaggio che Client invia al Server e che deve essere consegnato ad un altro Client.

    - Campi:
        - destUser : Campo Destinatario
        - sendUser : Campo Mittente
        - object : Oggetto del messaggio – Massimo OBJECT_MAX_SIZE
        - payload : Campo contenuto del messaggio – Massimo MESSAGE_MAX_SIZE

- READ_MESSAGE
    - Messaggio inviato dal Client per richiedere la lettura dei messaggi ricevuti
    - Campi :
        - sourceUser : Mittente dei messaggi da leggere. Se non nullo permette di leggere solo i messaggi inviati solo da uno specifico utente.
        - object : Oggetto dei messaggi da leggere. Se non nullo permette di leggere solo i messaggi con uno specifico oggetto.

Ciascun messaggio è provvisto di un HEADER con campi:

- type : codice numerico che identifica in modo univoco il tipo del messaggio.
- msgNum : codice numerico di gestione del messaggio. Usato dal Server per gestire il messaggio e dal Client per identificare in modo univoco il messaggio.

Questi codici numerici viaggiano tramite il socket codificati in Network Order: quando il Server o il Client inviano messaggi all'altro li codificano in Network Order, quando gli arrivano messaggi lo portano in Host Order.

Il Server

Discussione Generale

Il processo Server supporta un massimo numero di Clients iscritti al servizio, MAX_CLIENT_NUM. Questo limite è dato per sua natura dal salvataggio delle informazioni di login dei Clients all'interno di un file: essendo la dimensione di un file limitata, allo stesso modo è limitato il numero di Clients supportabili.

Per lo stesso motivo è limitato il massimo numero di messaggi che è possibile archiviare per ogni Client, MAX_MESSAGE_NUM.

Il server utilizza due strutture dati ausiliarie.

- CLIENT_NODE
  - Si tratta di un nodo istanziato dinamicamente per ogni Client iscritto al servizio. Permette di gestire l'accesso dei thread del Server all'interno del file di archiviazione messaggi del Client.
  - Campi
    - clientMutex : Mutex per l'accesso al file del client
    - userName : Nome dell'utente cui il nodo si riferisce
    - leftChild / rightChild : I nodi dei Clients sono organizzati in una struttura ad albero. Questo mi permette di avere un costo di ricerca del nodo che nel caso di albero bilanciato è di O(lg n), con n numero di Utenti attualmente iscritti al servizio. In caso di sbilanciamento dell'albero ho che il costo della ricerca degenera a O(n).

- CLIENT_FILE_HEADER
  - Struttura dati per la gestione del file del client. Si tratta di un Header che ogni thread del Server legge nel momento in cui si trova ad operare su un file client
  - Campi:
    - freePos. Un array che indica quali sono gli slot liberi nel file che possono essere occupati da nuovi messaggi
    - startMessagePos. Un array che contiene l'indice all'interno del file che indica la locazione in cui si trova il messaggio. Nello specifico il significato della segnatura startMessagePos[i] = k significa "i-esimo messaggio arrivato occupa slot k"
    - totMsg. Indica il numero di messaggi validi presenti all'interno di quel file.
    - Si vedano le funzioni serverSendMessage, serverDeleteMessage e serverReadMessage per maggiori specifiche sull'utilizzo di questo header.

serverMain.c

int main(void)

Per prima cosa, il Server crea, se non esistente, una nuova Directory, chiamata *serverDirectory*, all'interno della quale verranno inseriti tutti quanti i file che i suoi vari thread vanno a gestire.

Si provvede poi ad istanziare un socket, di dominio internet, comunicazione stream e protocollo standard (TCP). Su questo socket viene fatta una bind con indirizzo IN_ADDR_ANY, in modo che ci si possa connettere passando da varie interfacce della rete, compreso l'indirizzo di loop back nel

caso in cui un Client si trovi sulla stessa macchina del Server. Viene poi invocata la listen con backlog MAX_CLIENT_NUM e si imposta il riutilizzo veloce degli indirizzi.

Si crea, se non esistente, un nuovo file all'interno della *serverDirectory*, chiamato *clientsInfoFile*, all'interno del quale vengono scritte le utenze dei client.

Sono poi inizializzati dei semafori unnamed per gestire accessi concorrenti di thread sia al *clientsInfoFile* sia all'albero dei clients. Questi semafori sono entrambi di due entry e sono gestiti allo stesso modo:

- Prima Entry. Unico Token di Scrittura
- Seconda Entry. MAX_CLIENT_NUM Tokens di Lettura: infatti non posso avere un numero di thread lettori che supera MAX_CLIENT_NUM.

Viene invocata la funzione *fillClientsList*, che ha lo scopo di inizializzare l'albero dei clients.

A questo punto si invoca la accept e per ogni nuova connessione si lancia un nuovo thread, con parametro il puntatore al descrittore del socket istanziato dinamicamente e funzione di avvio *serverStartFunction*().

Viene impostata anche la gestione dei segnali. Nello specifico, per tutti i thread del server viene bloccata la consegna di tutti i segnali, tranne che per un thread. Per questo thread, il *controlThread*, vengono bloccati tutti i segnali, tranne quello di *SIGINT: l'*hanlder di gestione che viene invocato in caso di consegna di questo segnale è tale per cui si prendono tutti i token, sia di scrittura che di lettura, dai due semafori e si prendono tutti i mutex all'interno dell'albero. Fare questo significa bloccare l'esecuzione di qualsiasi attività da parte dei thread del server, tranne l'arrivo di una nuova connessione(anche se arrivasse una nuova connessione, verrebbe lanciato un nuovo thread che poi entrerebbe in blocco in attesa o di ricevere dati dal socket o di prendere tokens da uno dei semafori). A questo punto di può terminare l'esecuzione del processo server invocando la exit(0) e terminando quindi tutti i thread.

void fillClientList()

Questa funzione serve per inizializzare l'albero dei clients.

Le iscrizioni dei clients devono preservarsi attraverso avvii successivi del server, ovvero quando il server viene attivato ci possono essere già dei clients iscritti.

Il nodo di un client deve essere allocato anche se l'utente non è connesso, in quanto ogni altro utente può volergli spedire un messaggio e di conseguenza il thread del mittente deve eseguire un accesso sincronizzato al file del destinatario.

Ho scelto di utilizzare una struttura dati espandibile come un albero per due motivi:

1. Permette un inserimento semplice dei nodi
2. Mi permette di evitare frammentazione interna nel momento in cui gli utenti iscritti al servizio sono pochi rispetto al massimo numero di utenti possibili.

Se infatti avessi adottato un array di MAX_CLIENT_NUM entry, avrei avuto una grande zona di memoria allocata che avrebbe rischiato di non essere utilizzata pienamente con uno scarso numero di utenti iscritti.

Con un albero questo si evita, sebbene abbia un costo aggiuntivo di inserimento e di ricerca.

La funzione altro non fa che leggere i nomi degli utenti dal clientsInfoFile e allocare memoria per un CLIENT_NODE. La lettura del file avviene riga per riga: per ogni riga ho salvati username e password dell'utente (si veda la *serverSubscribeFunction* per specifiche sul formato del clientsInfoFile). Dopodiché si imposta il nodo, inizializzando il mutex, copiando il nome dell'utente nel nodo e inserendolo nell'albero.

In questa fase non ho bisogno di implementare un accesso sincronizzato né clientsInfoFile né all'albero dei nodi in quanto non ci sono ancora thread concorrenti attivi, che verranno lanciati solo dopo la chiamata di accept.

Un controllo ulteriore che vado a fare è quello dell'esistenza del file di archiviazione dei messaggi per un certo client. Infatti, quando un utente è iscritto al servizio, il suo file dovrebbe già esistere; per proteggere da eventuali perdite del file, utilizzo la funzione *clientFileInit* con parametro ANOMALY: ovviamente questa funzione non permette il recupero dei messaggi che erano scritti nel file perso, ma garantisce la creazione di un nuovo file che, da questo momento in poi, mi fa operare correttamente per questo utente.

serverLog.c

*void *serverStartFunction(void *param)*

All'inizio della procedura il parametro della funzione, che è il descrittore del socket tramite cui il thread del server dovrà comunicare con il client, viene copiato in una variabile TLS : questo mi permette di evitare di passare alle varie funzioni il descrittore come parametro e di averlo sempre a disposizione come variabile globale. D'altronde ogni thread serve un solo utente connesso, quindi non ha necessità di usare socket diversi dal suo.

A questo punto si comincia,in un do-while, a leggere dal socket il prossimo messaggio, finché la risposta che il server deve mandare al client non è positiva.

Viene letto prima l'header del messaggio: in base al campo type infatti posso conoscere la taglia del messaggio totale ed eventualmente completare la lettura.

Ci sono solo tre messaggi che Client può inviare a Server in questa fase e sono:

- Login
- Subscribe

- Exit

Nel caso in cui il messaggio sia un Login o un Subscribe si termina la lettura del resto e si invoca una delle due funzioni ausiliarie, altrimenti si invoca la funzione di uscita.

Solo se il Login o il Subscribe danno esito positivo si esce dal do-while e si può accedere al servizio di scambio dei messaggi tramite la *serverMessageFunction.*

Dopo che la funzione invocata ha ritornato un valore, qualunque esso sia, si invoca la funzione *answerToClient* passandolo come parametro: esso sarà il campo *type* del messaggio di risposta.

*int serverSubscribeFunction(LOGIN_MESSAGE *logMsgPtr)*

La funzione apre per prima cosa uno stream verso il clientsInfoFile: questo mi permette di fare una lettura formattata del file e di avere informazioni più comode da analizzare. Per poter accedere in lettura al file però devo poter prendere almeno un token di lettura: a questo punto, infatti, ci possono essere più thread concorrenti che ci stanno lavorando.

Finché non si arriva alla fine del file si continua a leggere una nuova riga. Le righe del file sono organizzate in questa formattazione “username\tpassword”. Si fa quindi una tokenizzazione e si controlla se esiste già un utente con lo stesso username. Trattandosi infatti di una iscrizione, non posso permettere che ci siano utenti con lo stesso nome, altrimenti potrei avere ambiguità di identificazione. Nel caso in cui ci sia, allora si libera la memoria allocata per leggere, si rilascia il token di lettura, si chiude lo stream verso il file e si ritorna NEG_TYPE_1, che in questa funzione ha significato di “Esiste Utente con Stesso Nome”.

Se non esiste un utente con nome uguale, allora bisogna inserirlo, e per farlo si deve scrivere sul file. Si preleva quindi il token di scrittura e tutti i token di lettura, per evitare che ci siano accessi mentre la modifica è in corso; si scrive la nuova riga con la formattazione di cui sopra (la riga viene aggiunta in fondo al file, essendo lo stream utilizzato per scrivere lo stesso che si è utilizzato per leggere), si chiude lo stream e si rilasciano i token presi. Posso chiudere lo stream e rilasciare i token di accesso al file perché non ho più operazioni da eseguire.

Si inizializza quindi il nodo del client e si prendono i token di scrittura e di lettura sull’albero: dovendolo modificare, dobbiamo evitare accessi concorrenti. Il nodo viene inserito mediante un normale inserimento in un albero binario di ricerca: se uno username è maggiore o minore (non può essere uguale) di un altro è stabilito tramite la comparazione lessicografica data dalla *strcmp.*

Prima di rilasciare i token di accesso all’albero però, devo istanziare e inizializzare anche il file messaggi del nuovo utente. Infatti se prima rilasciassi i tokens e poi istanziassi il file, potrei avere dei problemi di inconsistenza nella *serverSendMessage* (si veda più avanti per i dettagli) : la sendMessage, infatti, prima cerca il nodo del destinatario e, una volta trovato, apre il suo file ; se rilasciassi prima i tokens e poi creassi il file, un eventuale thread concorrente mittente, potrebbe trovare il nodo, ma non il file.

Il file viene creato nella serverDirectory e con nome "*userName*_messages" e inizializzato con la sua struttura header (si vedano la *serverSendMessage* e la serverDeleteMessage per i dettagli della gestione del file) tramite *clientFileInit.*

Fatto ciò si rilasciano i tokens di accesso all'albero e si ritorna il POS_TYPE: l'operazione di subscribe richiesta è andata a buon fine.

Void clientFileInit(int fileDesc , int how)

Questa funzione ha lo scopo di inizializzare il file del client.

Nelle condizioni standard, ho che l'header deve essere inizializzato nel seguente modo:

- startMessagePos. Tutte le entry sono -1: non ci sono posizioni di start per nessun messaggio.
- freePos. Tutte entry ad 1: tutte le posizioni sono libere.
- totMsg. Valore 0: non ci sono messaggi salvati nel file.

Nel caso in cui vi sia stata anomalia, allora devo comunicarlo al client: si esegue quindi una scrittura di un messaggio di servizio con mittente fittizio il server. L'operazione di scrittura di questo messaggio è analoga a quella che si fa in caso di send del messaggio: si occupa lo slot zero, si segna come primo messaggio quello nello slot appena occupato e si pone il numero di messaggi ad 1.

Alla fine, sia che vi sia stata anomalia o che non vi sia stata, si scrive l'header nel file e si ritorna.

*Int serverLoginFunction(LOGIN_MESSAGe *logMsgPtr)*

Anche in questa funzione si apre lo stream verso il clientsInfoFile, si prende il token di lettura sul file e si legge riga per riga. La riga letta si tokenizza in due stringhe: username e password.

Se il campo userName di logMsgPtr coincide con lo username nel file, allora significa che esiste un utente con quel nome e si controlla la password. Quando ho uguaglianza di username, posso già rilasciare il token di lettura e chiudere lo stream verso il file, in quanto posso avere solo due possibili esiti:

- Le password coincidono, quindi devo ritornare POS_TYPE perché il login ha avuto successo
- Le password non coincidono e, poiché non ci sono altri utenti con lo stesso nome, ritorno NEG_TYPE_1, che in questo caso ha significato di "Password errata".

Se si arriva alla fine del file e non sono state trovate corrispondenze con lo username dato dall'utente si ritorna NEG_TYPE_2, "Non Esiste Utente".

Void answerToClient(int ansType)

Quando viene eseguita una qualunque operazione, la funzione che la esegue ritorna un codice di risposta. Questa funzione prende come parametro il codice di risposta, lo impacca in un messaggio ANSWER_MESSAGE mettendolo come campo tipo, codificato secondo Network Order, e inviandolo in risposta al client tramite la *writeOnSocket.*

In questo modo il client può reagire in base alla risposta che gli arriva.

Void serverExitFunction()

Si tratta della funzione che viene invocata quando il server riceve un messaggio di chiusura. In questo caso viene chiuso il descrittore del socket. La variabile *myFileDesc* è una variabile TLS che viene inizializzata a valore diverso da -1 nella funzione *serverMessageFunction:* nel caso in cui sia diversa da -1 significa che il thread ha un file associato su cui operare ed esso quindi, all'atto della chiusura della sessione, deve essere chiuso.

serverMessage.c

*void serverMessageFunction(LOGIN_MESSAGE *logMsgPtr)*

Dopo il login (o a subscribe), il thread del server, sulla base del *logMSgPtr* dato come parametro, cerca il nodo del suo client e apre un descrittore verso il file: in questo modo per le operazioni di delete e di read non bisogna ogni volta cercare il nodo o aprire il file. Il *myFileDesc* è una variabile TLS: ho scelto di usare questo tipo di variabile sia per il descrittore del socket sia per il descrittore verso il file perché in questo modo, all'atto della chiusura, un thread può chiudere i canali di I/O e permettere all'applicazione di riutilizzare gli slot della tabella dei descrittori. La chiusura infatti può avvenire sia su richiesta esplicita del client, sia perché il thread, durante la *readFromSocket* o la *writeOnSocket*, può rendersi conto di una chiusura di sessione da parte del client: in questo ultimo caso, se non avessi usato variabili TLS, avrei avuto difficoltà a passare questi descrittori alla *serverExitFunction* per la chiusura.

Dopo il login (o la subscribe), gli unici messaggi che il server può ricevere sono:

- Messaggi di Send
- Messaggi di Read
- Messaggi di Delete
- Messaggi di Chiusura

Come nella *serverStartFunction* ci si mette in attesa di ricevere il prossimo messaggio: se ne legge prima l'header e poi si fa uno *switch* sul suo campo tipo, richiamando la funzione di gestione adatta e passandole un puntatore all'header, di modo che, se necessario, si possa terminare la lettura della parte finale del messaggio.

Ognuna di queste funzioni ritorna un valore ret che viene passato alla *answerToClient* e usato come campo tipo del messaggio di risposta.

Il ciclo while per la lettura del messaggio continua finché il campo tipo non è EXIT_TYPE: in questo caso il ciclo si interrompe e viene invocata la *serverExitFunction*.

*Int serverSendMessage(HEADER *msgHeader)*

La prima cosa che la *serverSendMessage* fa è terminare la lettura del resto del messaggio, e quindi di tutto ciò che non è l'header, ricostruendo il messaggio finale.

Dopo aver preso un token di lettura dell'albero, basandosi sul campo *destUser* del messaggio ricevuto, si cerca il nodo del destinatario tramite la funzione *treeSearchNode:* questa funzione fa una normale ricerca in un albero binario di ricerca e ritorna NULL nel caso in cui il nodo del destinatario non sia presente. Posso avere due casi: se il valore di ritorno è NULL si ritorna subito il valore NEG_TYPE_1, che ha significato di "Destinatario non esiste"

Se invece il nome del destinatario esiste, si può procedere con la scrittura del messaggio sul file del destinatario. Grazie al nodo, si prende subito il mutex su questo file: dovendo infatti scrivere sul file, non posso permettere accessi concorrenziali.

Subito dopo si apre il file e se ne prende il descrittore, il quale verrà chiuso al termine della funzione.

Dopo aver aperto il file, il thread ne legge subito l'header.

Prima di scrivere il messaggio controlla se il numero di messaggi scritti nel file non è il valore massimo: nel caso lo fosse, il messaggio non può essere scritto e, dopo aver chiuso e rilasciato il mutex, si ritorna NEG_TYPE_2 "Spazio di Archiviazione Destinatario Terminato".

A questo punto è venuto il momento di spiegare come viene gestito un file di archiviazione dei messaggi da parte del Server.

Nel pensare a come implementare la gestione del file da parte del Server mi sono basato su un principio fondamentale: evitare traslazione di informazioni già presenti all'interno del file per far posto a nuovi messaggi. Ho preso quindi ispirazione dalla gestione della memoria RAM tramite paginazione. Ho considerato il file come diviso in MAX_MSG_NUM slot: ognuno di questi slot

può essere in stato *Libero* o *Occupato;* questo stato è gestito tramite l'array *freePos* della struttura CLIENT_FILE_HEADER, in cui *freePos[i]* indica lo stato dello slot i-esimo del file.

Quindi, il thread che deve scrivere il messaggio come prima cosa cerca uno slot libero e, una volta che lo ha trovato (e lo trova sicuramente perché si è seguito il controllo precedente sul numero di messaggi), lo imposta come occupato. Supponiamo che lo slot sia lo slot i-esimo. L'indice di questo slot viene impostato anche come *msgNum* nel campo header del messaggio: questo codice mi serve per identificare in modo univoco il messaggio tramite la sua posizione all'interno del file.

Tuttavia questa gestione non è sufficiente: infatti devo anche stabilire un ordine nei messaggi affinché essi possano essere estratti in maniera corretta e secondo una politica LIFO (o FIFO, a seconda di come viene implementata la *serverReadFunction).* Questo è lo scopo dell'array *startMessagePos* all'interno del CLIENT_FILE_HEADER: la segnatura *startMessagePos[i] = k* significa che il messaggio i-esimo, in termine di ordine di lettura, è quello con codice univoco k e che quindi occupa anche lo slot k-esimo nel file.

Poiché stiamo inserendo un nuovo messaggio devo modificare *startMessagePos[0]*, ma prima devo traslare le altre informazioni all'interno dell'array: utilizzo quindi la *memmove (*devo usare la memmove piuttosto che la memcpy perché zona sorgente e zona destinazione si sovrappongono) per traslare le informazioni nell'array e alla fine aggiorno la entry *startMessagePos[0]* al codice univoco del nuovo messaggio.

Condizioni Iniziali

- Array freePos (suppongo 1 = Libera e 0 = Occupata)

1	1	1	0	1	0	1	1	0	0

- Array startMessagePos

0	1	2	4	6	7	-1	-1	-1	-1

- totMsg = 6

Condizioni Intermedie

- Array freePos

1	1	1	1	1	0	1	1	0	0

- Array startMessagePos

0	0	1	2	4	6	7	-1	-1	-1

- totMsg = 6

Condizioni Finali

- Array freePos

1	1	1	1	1	0	1	1	0	0

- Array startMessagePos

3	0	1	2	4	6	7	-1	-1	-1

- totMsg = 7

Dopo aver eseguito la traslazione e aver modificato la entry 0-esima di startMessagePos, si aggiorna il numero di totMsg. A questo punto abbiamo terminato la modifica dell'header: possiamo riscriverlo all'inizio del file.

L'ultimo passo consiste nello scrivere il messaggio nello slot che gli abbiamo assegnato. Lo slot inizia dopo l'header a partire da i * (dimensione messaggio di tipo send) : ci riposizioniamo a questo punto e scriviamo il messaggio.

Alla fine si libera il mutex, si chiude il descrittore e si torna POS_TYPE: "tutto è andato a buon fine".

*Int serverReadMessage(HEADER *msgHeaderPtr , CLIENT_NODE *myClientNode)*

Anche in questa funzione come nella *serverSendMessage* terminiamo la lettura del messaggio dal socket.

Dal nodo passato come parametro prendiamo il mutex sul file per impedire che durante la lettura del file ci siano altri thread che scrivono.

Si legge l'header del file e si controllano i campi del messaggio, per stabilire se si deve eseguire una lettura di tutti i messaggi oppure una lettura per mittente/oggetto.

A questo punto eseguiamo la vera e propria lettura.

In questo caso i messaggi sono letti e inviati all'utente secondo una politica LIFO: l'ultimo messaggio arrivato è il primo messaggio che viene rimandato all'utente che ha chiesto la lettura. Per garantire una lettura LIFO, per come le posizioni dei messaggi sono state organizzate nel vettore

startMessagePos, devo andare a leggere a partire dall'inizio del vettore fino alla entry di indice totMsg-1, in cui si troverà l'indice di slot del messaggio più vecchio.

Noto il numero di slot *i* posso riposizionarmi all'interno del file sul primo byte del messaggio, il quale si troverà in posizione *(FILE_HEADER + i * SEND_MSG_SIZE).* Una volta riposizionato non mi resta che leggere il messaggio e portarlo all'interno dell'array *msgToSendArray e* codificare il campo tipo e il numero del messaggio in Network Order. Una volta che tutti i messaggi sono stati letti posso rilasciare il mutex sul file.

A questo punto, posso inviare i messaggi al Client: se è stata richiesta una lettura di tutti i messaggi li invio tutti, altrimenti confronto il campo della ricerca.

Al termine della lettura ritorno positivo: "Lettura avvenuta con successo".

Il motivo per cui ho scelto di utilizzare un array intermedio per mettere i messaggi letti, piuttosto che inviarli al client subito dopo la lettura del singolo messaggio è il seguente: nel secondo caso, se un client richiedesse la lettura dei messaggi e subito dopo, per un motivo qualsiasi, dovesse terminare, il thread del server che gestisce la comunicazione con lui terminerebbe a sua volta; ma quel thread ha anche il lock sul mutex del file e, una volta terminato, non lo rilascerebbe, portando ad entrare in stallo ogni altro thread che tentasse di inviare un messaggio a quel client.

Una cosa importante su cui è necessario porre attenzione è che all'interno del messaggio che viene inviato al client, il campo msgNum ha un ruolo importante: è il codice identificativo del messaggio, che lo identifica in modo univoco e questo è importante in fase di cancellazione.

*Int serverDeleteFunction(HEADER *msgHeaderPtr , CLIENT_NODE *myClientNode)*

In questa funzione non ho bisogno di leggere parti del messaggio rimaste nel socket: il DEL_MSG infatti contiene solo HEADER.

La prima cosa che viene fatta è identificare, partendo dal campo msgNum scritto nel messaggio e specificato dall'utente, il codice identificativo del messaggio da cancellare.

Anche qui, come nelle precedenti funzioni, si prende il mutex, ci si riposiziona all'inizo del file e si legge l'header.

Dopo la lettura dell'header si scorre l'array *startMessagePos* finché non si arriva alla fine dei messaggi validi o finché non si trova il messaggio che è stato chiesto di cancellare.

Nel caso in cui si trovi il codice del messaggio in *startMessagePos*, allora si segna lo slot come libero, ponendo freePos[delNum] ad 1, si trasla l'array *startMessagePos*, in modo che il messaggio non sia più tra quelli presi in considerazione tra quelli validi, si diminuisce il numero di messaggi e si riscrive l'header.

Supponiamo che delNum = 5

Condizioni Iniziali

- freePos (1 = Libero , 0 = Occupato)

1	0	0	0	1	0	0	1	1	0

- startMessagePos

1	2	3	5	6	9	-1	-1	-1	-1

- totMsg = 6

Condizioni Intermedie

- freePos

1	0	0	0	1	1	0	1	1	0

- startMessagePos

1	2	3	6	9	9	-1	-1	-1	-1

- totMsg = 6

Condizioni Finali

- freePos

1	0	0	0	1	1	0	1	1	0

- startMessagePos

1	2	3	6	9	-1	-1	-1	-1	-1

- totMsg = 5

Alla fine si rilascia il mutex e si ritorna POS_TYPE : “Cancellazione eseguita con successo”.

Nel caso in cui il *delNum* non identifichi un messaggio valido, oppure sia maggiore di MAX_MESSAGE_NUM, l’operazione è una don’t care e comunque si ritorna POS_TYPE.

Anche qui come nella *serverSendMessage*, Si riesce, grazie alla gestione del file mediante l’header, ad evitare, all’atto della cancellazione, degli spostamenti di messaggi all’interno del file, semplicemente cambiando lo stato dello slot e traslando l’array delle posizioni di inizio dei messaggi validi: ovviamente la traslazione di un array ha costo minore della traslazione dei messaggi nel file che richiederebbe per ogni messaggio una lettura, un riposizionamento e una nuova scrittura.

Il Client

clientMain.c

int main()

Il client lancia una connessione verso il socket del server, il cui IP numeber è memorizzato in SERVER_IP_ADDR e SERVER_PORT_NUM. Se non esiste un Server con Socket avente questo indirizzo l’operazione fallisce e si esce: il server deve essere già attivo quindi all’atto del lancio del Client.

Il descrittore del socket usato per lanciare la connessione viene salvato in una variabile globale: in questo modo posso evitare di passarlo ad ogni funzione.

clientLog.c

void clientStartFunction()

In questa fase ci sono tre possibili operazioni che il client può richiedere:

- Login, cui viene associato codice di opzione LOG_CODE (“a”)
- Subscribe, cui viene associato codice opzione SUB_CODE (“b”)
- Exit, con associato codice di opzione EXIT_CODE (“z”)

Si comincia a leggere in un ciclo while il codice dell'opzione che inserisce il client, fino a quando il valore *ret*, che è il codice del messaggio che invia il server come risposta al servizio richiesto, è minore di zero, ovvero fino a quando il server ha inviato una risposta di tipo NEG_TYPE_%d.

La lettura del codice viene fatta attraverso la *fgets* da stdin. Ho scelto di utilizzare la *fgets* perché mi permette di avere controllo sulla massima quantità di bytes che possono essere messi nel buffer: infatti la *fgets* legge un numero di bytes che è al massimo il secondo parametro -1.

Il problema principale di questa funzione è che il '\n' usato per confermare l'input viene inserito anche lui nel buffer. Invoco quindi la funzione *myStrtok* che pone tutti i caratteri '\n' in '\0'. Ho deciso di implementare una nuova funzione invece di usare la *strtok* perché quest'ultima non modifica il primo carattere: un utente però potrebbe, alla richiesta di informazioni, inserire direttamente uno '\n' che, con la sola strtok, non sarebbe rimosso dalla stringa.

Inoltre, dopo aver letto la stringa, devo garantire che il buffer di input sia vuoto per la prossima lettura. Il buffer di input presenta ancora dei caratteri che non sono letti solo se la stringa inserita in input ha dimensione uguale a quella massima -1: in questo caso infatti mi rimane fuori dall'inserimento nel buffer almeno lo '\n'. Nel caso in cui la condizione si uguaglianza delle lunghezze sia verificata, il buffer di input viene svuotato, per evitare che dati spuri siano letti successivamente.

Dopo la lettura del codice dell'opzione, se il codice è valido, si invoca la funzione di gestione dell'opzione, altrimenti si visualizza un messaggio di invalidità del codice e si torna a leggere.

Se l'input è uguale a LOG_CODE o SUB_CODE, si richiama la funzione *clientLoginSubscribeFunction* passando come parametro il *msgType* corrispondente.

Int clientLoginSubscribeFunction(int msgType)

A lato client, le informazioni che si devono inviare al Server sono le stesse sia che si stia eseguendo un Login o un Subscribe.

Con la stessa regola di lettura di sopra si leggono username e password inseriti dall'utente, annullando l'operazione in caso in cui la lunghezza dell'uno o dell'altro sia nulla. Nel caso in cui le lunghezze non siano nulle si controlla che nessuno dei due contengano caratteri non validi: è importante evitare la presenza del tab in particolare, perché viene usato per separare username e password all'interno del file del server che gestisce le utenze.

Eseguiti i vari controlli si impaccano le informazioni in un messaggio avente per tipo quello dell'operazione richiesta (LOG_TYPE o SUB_TYPE a seconda del caso) e si invia il tutto tramite un LOGIN_MESSAGE.

Si attende poi la risposta tramite la *waitForAnswer* e si visualizza un messaggio a seconda del tipo di risposta ricevuto:

- NEG_TYPE_0 : Errore Lato Server
- LOG_TYPE
  - NEG_TYPE_1 : Password Errata
  - NEG_TYPE_2 : Non Esiste Utente
- SUB_TYPE
  - NEG_TYPE_1 : Esiste Utente Con Stesso Nome
  - NEG_TYPE_2 : Raggiunto Massimo Numero di Iscritti

Se si riceve un messaggio con tipo POS_TYPE, si salva l'username in una variabile globale, di modo che possa essere usato per operazioni successive.

Alla fine si ritorna il tipo della risposta.

Int waitForAnswer()

Si invoca la lettura del messaggio di risposta dal socket tramite la *readFromSocket* con dimensione ANS_MSG_SIZE. Viene ritornato quindi il tipo del messaggio dopo averlo convertito in Host Order.

Void clientExitFunction(int how)

Viene invocata quando ho la chiusura della connessione.

Se how vale zero oppure SIGHUP, allora la chiusura è su richiesta del client: in questo caso si invia prima un messaggio di chiusura al server e, senza aspettare risposta (essa non sarà mandata dal server), si chiude il socket della comunicazione e si termina il programma.

Se how vale uno, allora la chiusura è avvenuta a causa di una chiusura anomala del server: la funzione è stata richiamata dalla funzione *readFromSocket* o *writeOnSocket* e in questo caso non si invia nessun tipo di messaggio e si chiude direttamente.

clientMessage.c

void clientMessageFunction()

Solo dopo un login o una subscribe avvenuti con successo si passa ad eseguire questa funzione. Questa è la funzione che permette di accedere alle funzionalità dell'applicazione. In questa fase ci sono cinque funzioni a cui il Client può accedere:

- Lettura di Tutti i Messaggi
- Lettura per Mittente / Oggetto
- Invio di nuovo messaggio
- Cancellazione
- Uscita

Seguendo lo schema di presa dell'input delle funzioni precedenti, si legge la scelta fatta dal Client e si invoca la funzione adatta a gestire l'opzione.

Se il codice dell'opzione non è valido, si visualizza un messaggio di non validità e si passa alla prossima iterazione.

Void clientSendFunction()

Si prendono le varie informazioni che l'invio del messaggio richiede. Nello specifico:

- Non è consentito inviare un messaggio senza destinatario
- Si può mandare un messaggio senza oggetto
- Si può mandare un messaggio a contenuto nullo
- In ogni caso non si possono superare le taglie massime per nessuno dei campi del SEND_MESSAGE

Nel caso in cui uno degli input non risulti valido o violi qualche condizione si ritorna il controllo alla *clientMessageFunction.*

Il mittente del messaggio, ovvero l'username dell'utente che sta eseguendo l'accesso, viene preso dalla variabile globale inizializzata nella *clientLoginSubscribeFunctio.*

Una volta prese tutte le informazioni vengono inserite in un SEND_MESSAGE e inviate al Server. Dopodiché si attende la risposta e si visualizza un messaggio a seconda dell'esito dell'operazione.

*Void clientReadFunction(char *inputRow)*

In questo caso alla funzione viene passato come parametro il codice dell'opzione scelta dall'utente nel menù principale, di modo che si possa discriminare a seconda delle informazioni che devono essere lette per l'invio della richiesta.

Nel caso in l'utente abbia scelto una lettura per Mittente o per Messaggio, allora si provvede per prima cosa alla lettura di queste informazioni. Si alloca un buffer di dimensione adatta a contenere un campo Username o un campo Object. Anche qui non è consentito che i campi superino la massima dimensione che la fgets permette di leggere (ovvero la massima dimensione del campo -1).

Dopo che queste informazioni, se necessario, sono state lette, si invia un READ_MESSAGE al Server. A questo punto il Server risponde con una sequenza di messaggi di tipo SEND_TYPE, terminati da un ANSWER_MESSAGE: si leggono i messaggi dal socket finché il loro tipo è un SEND_TYPE, provvedendo nel mentre a scriverli su terminale, e poi si ritorna il controllo.

Questa modalità di lettura è uno dei motivi principali che mi ha spinto a scegliere dei socket di tipo Stream: se infatti non avessi ordine e affidabilità nel ricevere i messaggi che mi arrivano dal server, rischierei da un lato di non ricevere tutti i messaggi che sono stati inviati all'utente, e dall'altro di ricevere prima l'ANSWER_MESSAGE rispeto ad altri messaggi, lasciando il socket non vuoto e creando problemi per operazioni successive.

Void clientDeleteFunction()

Viene richiesto l'inserimento del codice del messaggio da cancellare. Una volta che è stato inserito, dovendo inviare un codice numerico al Server, si controlla che l'input dato sia effettivamente un numero: si controlla quindi che i caratteri inseriti ricadano nel range della codifica ASCII riservato alle cifre (il range è da [48 ; 57]): nel caso in cui ciò non avvenga, significa che l'utente ha inserito almeno un carattere che non è un numero e quindi non possiamo inviarlo al server. Questa verifica mi permette anche di verificare che non vengano dati in input numeri negativi: i codici identificativi dei messaggi sono maggiori o uguali a zero, quindi non ci possono essere messaggi negativi; in questo caso si evita direttamente l'invio del messaggio, per evitare dei problemi nella cancellazione lato Server. Se questo controllo viene passato allora l'input viene convertito in codice numerico.

Una volta ricevuto un codice maggiore o uguale a zero, esso viene scritto nel campo msgNum dell'HEADER di un DEL_MESSAGE e si invia al server ; si attende risposta e si ritorna il controllo.

Il new_client

A partire dal client di base ho pensato di sviluppare una seconda versione, costruendo una piccola interfaccia grafica basata sulle librerie GTK e sull'applicazione Glade.

Le funzionalità implementate sono le stesse, con l'unica differenza che le varie informazioni vengono richieste tramite interfaccia.

L'applicazione Glade permette di realizzare l'interfaccia per poi importarla all'interno del programma e utilizzarla in esso. Una funzionalità implementata è quella che permette di specificare l'handler di gestione per i segnali che colpiscono la finestra.

clientResource.c

Si tratta di un file all'interno all'interno del quale sono codificate le informazioni relative alle varie finestre che l'applicazione utilizza.

clientLog.c

void clientStartFunction()

Viene per prima cosa inizializzato l'utilizzo della libreria Gtk. Si istanzia poi un nuovo builer: si tratta di una struttura dati che ci permette di accedere alle varie informazioni sulle finestre definite nel sorgente *clientResource.c.* In realtà le informazioni sulle finestre possono essere prese sia a partire dal file sorgente, sia a partire da un file esterno *clientGUI.glade:* il come il builder prende le informazioni viene stabilito a tempo di compilazione a seconda delle macro definite.

Si invoca la funzione di connessione dei segnali delle finestre alle varie funzioni di gestione.

Dopodiché si prende la *loginWindow* e per le due entry che vi si trovano all'interno si imposta la massima lunghezza ammessa per l'input.

Si avvia poi il ciclo per la gestione della finestra.

Quando il ciclo termina, si controlla la variabile globale serverAns: se essa ha valore positivo, il che significa che il login o il subscribe hanno avuto esito positivo, allora si passa ad eseguire la *clientMessageFunction* altrimenti significa che è stato interrotto il ciclo di gestione della finestra senza che si sia acceduti al servizio, quindi è stata richiesta una chiusura: in questo caso si invoca la *clientExitFunction.*

Void on_loginButton_clicked() e void on_subcribeButton_clicked()

Quando uno dei due bottoni viene premuto, allora significa che è stato richiesto uno dei servizi: si prendono le informazioni contenuti nelle entry per l'inserimento dell'input e si passono come parametri alla *clientLoginSubscribeFunction*. Quest'ultima opera come la sua omonima base, con due sole differenze: prende username e password come parametri ; il messaggio con l'esito del servizio viene scritto in una label che compare nella finestra.

Nel caso in cui la *clientLoginSubscribeFunction* torni POS_TYPE, allora il servizio ha avuto successo: in questo caso si nasconde la finestra di login (ormai non serve più) e si termina il loop. La terminazione del loop porta, nella *clientStartFunction*, ad eseguire la *clientMessageFunction*

clientMessage.c

void clientMessageFunction()

Dal costruttore si prendono le informazioni sulle strutture dati per la gestione delle varie finestre che bisogna gestire in questa parte del programma.

Chiediamo di mostrare la finestra e avviamo al ciclo per la gestione. La finestra che viene mostrata presenta un bottone per ognuno dei servizi che può essere richiesto in questa fase.

Void on_mainProgramWindow_destroy()

Premere il bottone di chiusura, viene mappato sulla *clientExitFunction* e sulla terminazione del programma.

Void on_readAllButton_clicked() , on_readSourceButton_clicked , on_readObjectButton_clicked , on_searchEnterButton_clicked

Le prime tre funzioni sono gli handler di gestione dell'evento di click sui pulsanti di lettura di tutti i messaggi, lettura per oggetto o lettura per mittente.

Nella *on_readAllButton_clicked* quello che si fa è semplicemente invocare la *clientReadFunction* con parametro il READ_ALL_CODE.

Nella *on_readSourceButton_clicked* e nella *on_readObjectButton_clicked* viene nascosta la finestra del menù e mostrata una finestra con una entry per il campo della ricerca. Si scrive in una label il tipo di ricerca che si sta eseguendo e si imposta la massima dimensione di input della entry nella finestra a quello del campo della ricerca.

Inoltre si imposta la variabile globale *searchCode* in modo che sia coerente con il tipo di lettura.

La funzione on_searchEnterButton_clicked, è la funzione che viene eseguita quando si preme il tasto di conferma di lettura tramite campo di ricerca. Quello che viene fatto è nascondere la finestra di ricerca, mostrare la finestra principale e richiamare la funzione di lettura dei messaggi con il codice impostato precedentemente nella variabile globale *searchCode*.

Void clientReadFunction()

La funzione è molto simile alla sua versione di base.

Nel caso si stia facendo la lettura per oggetto o per mittente, allora si prende il campo della ricerca dal buffer nella finestra. Si controlla che il campo soddisfi tutte le condizioni. Nel caso in cui il campo ricerca non soddisfi le condizioni, viene scritto l'esito dell'operazione su una label nella finestra del menù.

A questo punto si invia un messaggio di lettura al server.

I messaggi ricevuti come risposta vengono scritti in un file nascosto (il file *.readMessageFile*), in modo che non compaia all'utente. Quando tutti i messaggi ricevuti sono stati scritti nel file, il contenuto del file viene preso e scritto in un buffer utilizzato per popolare una finestra di visualizzazione del testo. Il file viene chiuso e unlinkato in modo che sia eliminato: in questo modo non rimangono file nascosti sparsi all'interno del sistema ; esso viene quindi utilizzato solo come file ausiliario per la sola operazione di lettura.

La finestra viene visualizzata e si scrive nella label del menù che l'operazione è andata a buon fine.

Questa finestra di lettura è l'unica che può rimanere aperta insieme ad altre (sia quella del menù sia le altre): infatti un utente potrebbe, ad esempio, voler leggere i messaggi che sta cancellando mentre ne richiede la cancellazione.

Questa finestra di lettura viene aggiornata solo a seguito di una nuova lettura: quindi dall'ultima lettura potrebbero essere arrivati nuovi messaggi, che saranno visualizzati solo con una nuova lettura e non in automatico.

Void on_sendMessageButton_clicked() e void on_confirmSendButton_clicked()

Quando viene premuto il bottone per l'opzione di invio del messaggio, viene chiusa la finestra menù e si apre una finestra per inserire i dati.

Anche per questa finestra vengono impostate le massime lunghezze per il campo destinatario e per il campo oggetto.

Quando sono stati inseriti i dati ed è stato premuto il bottone di conferma si attiva la funzione di gestione *on_confirmSendButton_clicked:* viene invocata la funzione di clientSendFunction e, quando essa ritorna, si nasconde la finestra per l'inserimento dei dati e si rimostra quella del menù.

Void clientSendFunction()

Le informazioni scritte all'interno della finestra di invio del messaggio vengono prese e controllate. Nel caso in cui ci siano dati che non rispettano alcune delle loro condizioni, si ritorna, dopo aver scritto nella label di esito il motivo dell'impossibilitò di portare a termine l'operazione.

Se tutti i campi rispettano i loro vincoli, allora le informazioni vengono inserite in un SEND_MESSAGE e quest'ultimo viene inviato al server.

Anche qui si aspetta il responso del server e si scrive un messaggio all'utente nella solita label.

Void on_deleteMessageButton_clicked() e on_confirmDeleteButton_clicked()

Quando viene premuto il bottone per chiedere la cancellazione di messaggio, viene invocata la prima funzione. Questa nasconde la finestra principale e imposta il range per il contatore nella finestra che viene mostrata in questo momento.

All'interno di questa finestra abbiamo un spin button che permette di inserire un numero. Nel caso in cui si inserisca un numero negativo, esso viene arrotondato a -1, nel caso in cui venga inserito un

numero maggiore a MAX_MESSAGE_NUM, questo viene arrotondato a quest'ultimo valore. Il valore inserito tramite questo bottone è il codice numerico del messaggio da cancellare.

Quando viene premuto il bottone di conferma si attiva la seconda funzione: questa nasconde la finestra per l'inserimento del numero e mostra di nuovo la finestra principale, dopo aver invocato la *clientDeleteFunction*.

Void clientDeleteFunction()

La funzione opera allo stesso modo della sua equivalente, con la differenza che il codice del messaggio viene preso dallo spin button della finestra.

Se esso è negativo, il messaggio non viene inviato, altrimenti si invia un DEL_MESSAGE con *msgNum* questo codice numerico.

Anche qui si visualizza un messaggio di conferma nella label del menù.

*Gint on_*nomeFinestra*_delete_event()*

Le funzioni di gestione del delete event, ovvero l'evento che si verifica all'atto del premere il pulsante di chiusura delle finestre, nascondono le finestre e fanno riapparire la finestra del menù.

L'unica finestra che fa eccezione è la *readMessageWindow,* per la quale viene nascosta solo la finestra di lettura dei messaggi, in quanto la finestra di lettura e quella del menù possono rimanere aperte contemporaneamente, a differenza delle altre.

Quindi chiudere una finestra significa in sostanza annullare l'operazione a cui quella finestra si riferisce per tornare al menù.

Compilazione e Avvio

Server

Il Server può essere compilato con gcc linkando insieme i tre sorgenti e usando -pthread in fase di compilazione: si tratta infatti di un'applicazione multithread. È importane all'atto della compilazione avere nella directory in cui si compila i file : *messageTypes.h, serverDeclaration.h* e *serverStructs.h.*

Può essere avviato da terminale e può essere terminato attraverso la ricezione di un segnale SIGINT (contol + c).

Client

Il Client può essere compilato con gcc linkando i tre file sorgente. Anche qui bisogna avere nella directory di compilazione i file *messageTypes.h* e *clientDeclaration.h.*

Può essere avviato da terminale e si può terminarlo dall'interno dell'applicazione tramite l'opzione di uscita oppure chiudendo il terminale di lancio (e quindi mandando il segnale SIGHUP al processo).

new_client

Per compilare questo tipo di Client bisogna avere installati nel sistema le librerie Gtk-dev. In questo caso allora si possono compilare insieme i tre file sorgente e aggiungere in compilazione le opzioni *'pkg-config –cflags gtk+-3.0'* e *'pkg-config –libs gtk+-3.0'* che indicano al compilatore dove trovare le funzioni che sono usate per le librerie Gtk. È Necessario aggiungere in compilazione anche l'opzione *-rdynamic,* che permette di collegare correttamente le funzioni di gestione degli eventi alle finestre.

Nel caso in cui le informazioni sulle finestra si prendano da un file risorsa (il *clientResource.c*), allora bisogna aggiungere anche questo file in compilazione e bisogna metterlo come primo file della sequenza di compilazione.

Il file risorsa può essere creato da linea di comando tramite:

glib-compile-resources –target=clientResource.c –generate-source clientGui.xml

Il *clientGUI.xml* è un file attraverso il quale è indicato il file da cui il file di risorsa deve essere generato.

Anche questo Client può essere avviato da terminale. In questo caso è possibile chiudere il terminale di lancio senza terminare l'applicazione; può essere terminata attraverso il tasto di chiusura della finestra del login o della finestra del menù.