Relazione progetto ChatFe

DANIELE TOLOMELLI
SIMONE VENTURELLI
June 5, 2015

Introduzione

Nella realizzazione di questo progetto abbiamo scelto di implementare prima una serie di funzioni di utilità, non vincolate direttamente al sorgente principale ma disegnate sulle sue esigenze, in modo da semplificare successivamente il lavoro. Questo sia per ottenere una migliore leggibilità sia per garantire l'inserimento di porzioni di codice precedentemente testato, al fine di restringere il campo nella ricerca di eventuali errori. Abbiamo cercato di usare il più possibile funzioni di sistema, sempre per ottenere codice più veloce e testato possibile.
Generalmente nel disegnare le funzioni abbiamo cercato di far passare all'utente il minor numero possibile di argomenti, in modo da ridurre le possibilità di errori nella chiamata delle funzioni. Un esempio soggetto a questo stile implementativo sono le funzioni di `message.c` le quali, pur rischiando di suonare ridondanti, spiegano nel nome il loro intento e limitano l'utente a fornire soltanto l'effettivo contenuto dei messaggi. Abbiamo ordinato queste funzioni in diversi file .c suddivise per categorie di utilizzo, e ne abbiamo raccolto i prototipi nel file `utils.h`.

- `hdata.c`
  Funzioni di gestione della hash table: salvataggio e caricamento da file e conversione dell'elemento in stringa e viceversa. La funzione `getDataFrom()`, restituendo il puntatore all'elemento, alleggerisce semplicemente la sintassi al sorgente chiamante.
- `log.c`
  Funzioni di gestione del file di log: scrittura dell'accesso di un utente al server, scrittura di un messaggio da parte di un utente e scrittura di un errore. A queste funzioni ha accesso unicamente il server.
- `stringList.c`
  Funzioni che gestiscono un tipo di dato astratto `StringList` che usi-

amo per memorizzare gli utenti attualmente connessi. Le funzioni sono di inserimento, cancellazione, verifica e listing degli elementi.

- `message.c`
  Funzioni di comunicazione tra Server e Client. Le iniziali nel nome della funzione indicano la direzione della comunicazione, prima lettera per il mittente e seconda per il destinatario.
- `ringBuffer.c`
  Funzioni che gestiscono le comunicazioni tra thread attraverso un buffer circolare. Implementando queste funzioni si sfoltisce notevolmente il codice nel sorgente principale, delegando tutto il lavoro a queste funzioni. Esse scrivono e leggono esattamente la lunghezza del messaggio, evitando così scritture di caratteri vuoti, ottimizzando così lettura e scrittura.
- `misc.c`
  Raccolta di varie funzioni, come `timestamp()` per l'inizializzazione del file di log e funzioni di marshalling. La funzione `cmdmatcher()` traduce il comando inserito dal client in un corrispettivo intero, in modo da poter scrivere il sorgente del client attraverso una struttura `switch`, adatta alle numerose richieste del client.

Precisazione

Abbiamo deciso, per non appesantire la lettura, di non ripetere nel dettaglio il funzionamento e l'organizzazione dei lati client e server, in quanto già ampiamente esposti nella consegna dell'esercitazione. Ci limiteremo ad elencare alcune delle scelte implementative più significative.

Server

Nel nostro progetto abbiamo preferito l'utilizzo della funzione `calloc()` al posto della suggerita combinazione `malloc()` - `bzero()` per riutilizzare locazioni di memoria allocate una sola volta e azzerate ad ogni ciclo. Riserviamo la `bzero()` solo per reinizializzare i dati all'inizio di un nuovo ciclo, in modo da mantenere la posizione in memoria e non occupare ulteriore spazio.

Client

La funzione `reglog()` in realtà esegue solo la registrazione, ma il controllo passa immediatamente alla fase di login, che viene quindi eseguito in automatico. In questo modo si sovraccaricano le operazioni di comunicazione,

dovendo ripetere un passaggio, riuscendo però a snellire il codice. Essendo un operazione che non viene eseguita molto frequentemente consideriamo accettabile questo compromesso.

Hash Table

La gestione dello user-file nel nostro programma e' divisa in 3 passaggi: caricamento da file di testo nella hash, modifica della hash e salvataggio nel file di testo. Purtroppo la via più semplice non è quella più efficiente. Infatti al momento del salvataggio nel file, cancelliamo il file precedente per inserire nuovamente ogni voce nella hash aggiornata con le ultime registrazioni, e se per qualsiasi motivo il programma terminasse, perderemmo interamente lo user-file. Anche se il programma terminasse in un altro punto, non riusciremmo a salvare le ultime modifiche alla hash che andrebbero irrimediabilmente perse. Per risolvere questi problemi sarebbe sufficiente aggiornare la user file ad ogni registrazione e per il salvataggio bisognerebbe creare un secondo file in cui salvare la hash e soltanto quando l'operazione è andata a buon fine, sovrascrivere il file originale con quello appena generato per proteggersi da accidentali perdite di dati.

Marshalling

La consegna specificava di gestire anche la lunghezza del messaggio, per poterlo leggere in modo preciso, senza impiegare risorse per leggere caratteri nulli. Tuttavia abbiamo preferito strutturare l'algoritmo diversamente: separiamo ogni campo da un carattere di delimitazione in modo da poter leggere il messaggio attraverso `strtok()`. Questa scelta implica però il fatto che leggiamo sempre blocchi di testo pari a 256 caratteri. Abbiamo ritenuto questo cambiamento accettabile poichè, effettuando un rudimentale profiling, ossia scambiando 2^{20} messaggi di lunghezza 256 e di lunghezza 32, abbiamo verificato che la differenza nei tempi di esecuzione non è sostanziale da poter giustificare un algoritmo più complesso. La maggior parte del tempo nella comunicazione viene spesa per gestire la comunicazione stessa e non nell'effettivo scambio di contenuto.
Nel nostro algoritmo quindi manteniamo aggiornato il valore della lunghezza del messaggio, anche se non viene effettivamente utilizzato.

Error Checking

Oltre alle funzioni richieste dalla consegna abbiamo implementato ulteriori controlli per rendere più stabile l'intera esecuzione, per garantire un

migliore esperienza all'utente ma anche a noi sviluppatori nelle fasi di testing. Per un elenco riassuntivo degli errori gestiti possiamo rimandare alle macro definite in `log.c`, elencate di seguito.

```
#define LOGIN_DONE_YET          "Utente gia' connesso"
#define USER_NOT_REGISTERED     "Utente non registrato"
#define USER_REGISTERED_YET     "Utente gia' registrato"
#define HASH_COLLISION          "Collisione nella Hash Table"
#define RECV_NOT_REGISTERED     "Destinatario inesistente"
#define RECV_OFFLINE            "Destinatario non connesso"
#define USER_TOO_LONG           "Username troppo lungo"
#define CMD_NOT_FOUND           "Comando non trovato\n"
```

Originariamente questa tabella era stata disegnata per rappresentare in un unico file tutti i messaggi di errore che Client e Server dovevano scambiarsi. Successivamente vi abbiamo scritto ogni messaggio che poteva essere stampato anche a schermo, mantenendo nel codice solo delle macro. Facciamo notare che forniamo una soluzione a ogni problema elencato, fatta eccezione per la collisione in hash table. Infatti le funzioni di accesso alla hash table fornite non gestiscono questo caso e abbiamo deciso di non implementarle non essendo richiesto dalla consegna.
Possiamo individuare 2 famiglie di errori: errori di *Login* ed errori di *Input*. Gli errori di Login non permettono l'accesso al server da parte del client con le attuali credenziali, sono particolarmente gravi perché arrivano a modificare la vita dei thread stessi, e richiedono quindi una buona gestione delle terminazioni, per evitare la chiusura inaspettata di un servizio.
Gli errori di Input avvengono dopo aver già stabilito una connessione, che viene comunque mantenuta, fanno semplicemente ignorare il comando appena inserito perché non rispetta la sintassi richiesta, ritornando comunque la shell all'utente.

Difficoltà incontrate

Le difficoltà maggiori, oltre ai normali problemi di implementazione, le abbiamo avute nell'affrontare per la prima volta un lavoro di queste dimensioni in gruppo, nelle funzioni di gestione della tabella hash, nella buona gestione della memoria e nella ottimizzazione generale del codice. Per la prima volta abbiamo dovuto affrontare un progetto approssimabile al migliaio di righe di codice e per questo è stato necessario organizzare al meglio il codice in modo da evitare ripetizioni, massimizzarne la leggibilità e l'efficienza. È stata necessaria la suddivisione in più file per mantenere il codice del programma principale il più pulito possibile.

Anche al lavoro di gruppo non eravamo molto abituati e ce ne siamo resi conto strada facendo. La capacità di dividersi i compiti mantenendo comunque l'implementazione più leggibile da entrambi e più corretta possibile è stato uno degli ostacoli maggiori da affrontare. E' capitato più volte che una porzione di codice scritta da uno dei due fosse poco chiara per l'altro, o che addirittura si rivelasse incompatibile con le funzioni precedentemente implementate.
Segnaliamo inoltre nell'implementazione della hash table la mancanza di una funzione per accedere ai dati. Abbiamo dovuto provvedere noi alla scrittura di quest'ultima.
Molti problemi sono nati dalla cattiva gestione della memoria, infatti ci è capitato svariate volte di non aver allocato la memoria dove serviva o di allocarla doppiamente, creando inefficienze nel codice. Nelle parti finali del controllo del programma abbiamo usato un memory profiler (valgrind) per renderci conto della qualità del nostro codice e ci siamo accorti che le funzioni di marshalling non erano scritte così bene come pensavamo. Infatti a ogni messaggio scambiato si perdevano definitivamente i 256 Byte del messaggio, e considerando la finalità del programma ci è sembrato necessario spendere alcune ore per risolvere questo problema.
Abbiamo scelto, sbagliando, di scrivere il makefile solo a progetto ultimato, e ovviamente abbiamo incontrato non poche difficoltà in quanto abbiamo dovuto interamente ridisegnare il sistema delle dipendenze tra file, che in questo progetto non sono pochi. Durante questo processo abbiamo rinominato il file `hash.h` in `hash.c` insieme alla sua implementazione di lista. Abbiamo scelto questo metodo perchè alla fine della scrittura funziona tutto. Siamo sicuri comunque che migliorando le dipendenze sarebbe possibile renderlo un comune file header, ma per ragioni di tempo consideriamo accettabile l'attuale implementazione. Aggiungiamo infine che le modalita di chiusura dei threads non rispettano la consegna. Effettuiamo una semplice `exit(0)` per terminare tutti i threads una volta liberate le risrse e salvata l'hash table, poiche non siamo stati in grado di controllare la terminazione di tutti i threads attraverso la variabile globale `go`.

Difficoltà evitate

Buone pratiche di sviluppo ci hanno evitato contrattempi che si sarebbero altrimenti verificati. Innanzitutto abbiamo deciso fin dal principio di scomporre grossi problemi nel maggior numero possibile di sotto-problemi facilmente risolvibili, in modo da assemblare client e server su un codice precedentemente testato e sicuro. Questo ha richiesto un notevole sforzo iniziale nell'implementazione di un ampio numero di funzioni, ma alla fine siamo stati più che soddisfatti della nostra scelta, poichè riteniamo il nostro codice

molto snello e leggibile. Inoltre attraverso l'uso di Git siamo riusciti più volte a recuperare il lavoro salvato precedentemente in un punto stabile, evitando di riscrivere intere porzioni di programma dopo un errore. Mantenere un corretto log con i vari `commit` di Git ci è stato di grande aiuto per avere una traccia del nostro lavoro, utile sia in corso d'opera sia al termine per ricontrollare il percorso seguito. Inutile sottolineare che avendo sempre disponibile l'ultima versione aggiornata ci ha evitato di perdere tempo nel salvataggio dei file di volta in volta.

Memory Leak

Dopo aver controllato la stabilità del nostro software abbiamo voluto spendere un po' di tempo nella sua ottimizzazione, studiando il suo buon funzionamento attraverso un memory profiler, valgrind. Grazie ad esso abbiamo individuato diversi memory leak, che siamo riusciti a sopprimere quasi completamente, soprattutto dal lato server.
Dall'inizio della fase di testing al termine della sua esecuzione l'eseguibile `chat-server` lasciava sempre una traccia in memoria di circa 65KB non recuperabili, in seguito a varie imperfezioni di volta in volta corrette.
Sorgenti di errore sono state soprattutto l'operazione di listing degli utenti connessi, operazioni di marshalling, che si ripercuotevano sui singoli messaggi e sulle scritture sul buffer circolare. Segnaliamo che le funzioni per gestire la hash table non forniscono un metodo per la sua liberazione, metodo che abbiamo provveduto a scrivere. In generale non siamo riusciti a marginare tutti i memory leak, ma siamo riusciti, riscrivendo porzioni di codice, a limitare quelli che contribuivano maggiormente ad aumentare le dimensioni del leak ad ogni operazione, quindi quelli più pericolosi. I punti interessati sono stati:

- Hash Table

```
hash
12,705 (11,964 direct, 741 indirect) bytes in 997 blocks are definitely lost
   at 0x402A17C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x8049915: CREALISTA (in /home/ventu/tolve/chat-server)
   by 0x8049B75: CREAHASH (in /home/ventu/tolve/chat-server)
   by 0x8048E8C: Server (in /home/ventu/tolve/chat-server)
```

  Nonostante liberassimo la memoria dalla hash table dichiarata nel server, restavano allocate ancora informazioni riguardanti le liste. Scrivendo la funzione `DISTRUGGIHASH()` siamo riusciti ad accedere a tutte le informazioni della hash eliminando un memory leak delle dimensioni di 12-13KB.

- stringList

```
8,192 bytes in 1 blocks are definitely lost
   at 0x402C109: calloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x804AB05: listLoggedUser (in /home/ventu/tolve/chat-server)
   by 0x80497A5: Worker (in /home/ventu/tolve/chat-server)
   by 0x4054F6F: start_thread (pthread_create.c:312)
   by 0x4156BED: clone (clone.S:129)
```

  Nel passaggio della stringa al chiamante degli utenti connessi non tenevamo conto che questo sapesse esattamente quanti fossero gli utenti connessi. Quindi allocavamo la massima stringa possibile, ovvero `MAXLOGGEDUSER*USERNAMELENGTH`, lasciando generalmente vuota la quasi totalità dei byte. Aggiungendo come parametro il numero di utenti connessi riduciamo questa inefficienza adattandola al caso. Una diversa implementazione potrebbe riuscire a liberare tutta quanta la memoria allocata, ma ci limitiamo a questa soluzione per motivi di tempo.

- Marshal

```
3,143 bytes in 13 blocks are definitely lost
   at 0x402C109: calloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x8049FB8: marshal (in /home/ventu/tolve/chat-server)
   by 0x80496E1: Dispatcher (in /home/ventu/tolve/chat-server)
   by 0x4054F6F: start_thread (pthread_create.c:312)
   by 0x4156BED: clone (clone.S:129)

3,072 bytes in 12 blocks are definitely lost
   at 0x402C109: calloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x8049F90: marshal (in /home/ventu/tolve/chat-server)
   by 0x80496B9: Worker (in /home/ventu/tolve/chat-server)
   by 0x4054F6F: start_thread (pthread_create.c:312)
   by 0x4156BED: clone (clone.S:129)

3,584 bytes in 14 blocks are definitely lost
   at 0x402C109: calloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
   by 0x8049FBE: marshal (in /home/ventu/tolve/chat-server)
   by 0x8049761: Worker (in /home/ventu/tolve/chat-server)
   by 0x4054F6F: start_thread (pthread_create.c:312)
   by 0x4156BED: clone (clone.S:129)
```

  L'operazione iniziale di marshalling non poteva permettere alcuna free, in quanto il valore di ritorno veniva allocato e immediatamente restituito come parametro a un'altra funzione. Abbiamo scelto di racchiudere la marshal in una `marshalDirect` che prende un valore di uscita come argomento, dando la possibilita' al chiamante di allocare e liberare la risorsa su cui la marshal agisce. Essendo inoltre sempre inglobata in una operazione di write, abbiamo creato una routine chiamata `freeWrite()` che gestisce la memoria correttamente inviando

il messaggio, riducendo al minimo il numero di istruzioni nel server. Così facendo siamo riusciti a risolvere problemi di leak nell'invio di messaggi singoli, broadcast e scrittura su buffer circolare, il tipo di leak più pericoloso in quanto incrementava le sue dimensioni ad ogni invio di un messaggio.

- Attraverso queste ottimizzazioni siamo riusciti a ridurre i memory leak da 64-65KB, con errori che potevano incrementare questo numero, a 2KB fissi, fatta eccezione per un leak al login che continua ancora ad accumulare errori. Lo riteniamo un errore trascurabile essendo il login una operazione meno comune dell'invio di un messaggio.

```
LEAK SUMMARY:
   definitely lost: 2,797 bytes in 147 blocks
   indirectly lost: 1,160 bytes in 101 blocks
     possibly lost: 136 bytes in 1 blocks
   still reachable: 1,348 bytes in 8 blocks
        suppressed: 0 bytes in 0 blocks
```