`lcschat`: una chat distribuita

Francesco Nidito, Susanna Pelagatti e Claudio Scordino

Progetto del corso di LCS 2005/06

1 Introduzione

Il corso di Laboratorio di Programmazione Concorrente e di Sistema (AA538 – 6 crediti) prevede lo svolgimento di un progetto finale in gruppi di non più di 2 persone. Il progetto consiste nello sviluppo del software relativo ad un sistema client-server che realizza una chat distribuita (`lcschat`), e della relativa documentazione utilizzando gli strumenti presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso puó essere reperito su Web alla pagina

`http://www.di.unipi.it/~susanna/LCS/`

Tale pagina conterrà anche le informazioni più aggiornate sul progetto (es. FAQ, suggerimenti, avvisi), sul ricevimento e simili. Il sito è un Wiki e gli studenti possono registrarsi per ricevere automaticamente gli aggiornamenti delle pagine che interessano maggiormente. In particolare consigliamo a tutti di registrarsi alla pagina delle FAQ e degli 'avvisi urgenti'.

Eventuali chiarimenti possono essere richiesti consultando i docenti durante l'orario di ricevimento e/o per posta elettronica.

1.2 Struttura del progetto

Il progetto è suddiviso in 3 **frammenti** incrementali, a ciascuno dei quali è assegnata una data di scadenza. Ogni frammento corrisponde ad una parte del progetto complessivo e deve essere realizzato dagli studenti utilizzando un kit fornito dai docenti che contiene il software di supporto alla realizzazione del frammento, i test per valutare la correttezza del software sviluppato e l'occorrente per la consegna.

Il kit relativo ad ogni frammento verrà fornito (messo in linea sulla pagina del progetto) dai docenti circa 4 settimane prima della scadenza del frammento corrispondente. Durante le 4 settimane di sviluppo del frammento una o più esercitazioni verranno interamente dedicate allo sviluppo del codice del frammento. L'idea è di sviluppare ogni frammento per la maggior parte in aula, con l'aiuto dei docenti.

I gruppi che consegnano una versione corretta (che rispetta i test) e documentata di un frammento entro la scadenza corrispondente ottengono un Bonus che andrà ad incrementare la valutazione finale del progetto (1 bonus=2punti, massimo 6 punti di incremento).

1.3 Consegna del progetto

La consegna di un frammento o del progetto complessivo avviene per posta elettronica **esclusivamente** usando i makefile forniti dai docenti con i tre kit di sviluppo. Per la consegna seguire le istruzioni contenute nei file README dei kit. Il progetto si intende **consegnato** quando sono stati consegnati tutti i 3 frammenti che lo compongono.

Data ultima per la consegna è il 1 Febbraio 2007. La discussione deve essere effettuata da tutti i componenti del gruppo in una data vicina a quella della consegna.

Si noti inoltre che gli studenti che intendono discutere il progetto entro Luglio 2006 devono consegnare entro e non oltre il 31/06/06, gli studenti che intendono discutere il progetto entro Settembre 2005, devono consegnarlo entro e non oltre il 10/09/06, mentre la data ultima di consegna per gli altri è il 1/02/07.

1.4 Valutazione del progetto

Al progetto, se accettato, viene assegnata una **valutazione base** da 18 a 26, di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. Gli studenti sostengono inoltre una prova orale individuale che viene valutata da 0 a 26 e che fa media con il voto del progetto. Alla valutazione complessiva di base (progetto + orale) vengono sommati i bonus accumulati con le consegna dei frammenti intermedi per ottenere il voto finale (es. base=24, bonus = 6, voto = 30). La valutazione del progetto viene assegnata in base ai seguenti criteri:

- motivazioni, originalità ed economicità delle scelte progettuali
- strutturazione del codice (suddivisione in moduli, uso di makefile e librerie etc.)
- efficienza e robustezza del software
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della documentazione

L'orale serve a valutare l'apporto individuale dei vari componenti del gruppo alla stesura del progetto e comprende:

- una discussione delle scelte implementative,
- la scrittura di programmi C non banali sequenziali e concorrenti
- domande su tutto il programma presentato durante il corso.

Se la discussione evidenzia una scarsa o nulla partecipazione al progetto di uno degli studenti del gruppo, a questo soltanto vengono applicati dei **malus** di penalizzazione, eventualmente richiedendo la presentazione di un progetto integrativo basato su quello originale.

Il voto è in trentesimi ed è individuale.

Da considerazioni aritmetiche, il 30 è a portata di chi consegna almeno due frammenti entro la scadenza, mentre chi consegna il progetto completo fuori scadenza *in unica soluzione*, oltre a non fruire di assistenza, non può andare oltre il 26.

1.5 Casi particolari

Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare il progetto in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30.

Gli studenti che svolgono il progetto per mutuare LCS o LPS non devono sviluppare lo script bash e possono consegnare il progetto in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30.

Gli studenti che svolgono il progetto per abbreviazioni delle nuove lauree quinquennali sono invitati a contattare il docente.

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di LPS contribuiva al voto di SO) avranno assegnato un voto in trentesimi che verrà combinato con il voto del corso di SO corrispondente secondo modalità da richiedere ai due docenti coinvolti.

2 Il progetto: `lcschat`

Lo scopo del progetto è lo sviluppo di due programmi C, un server (*server*) ed un client (*client*) che realizzano una chat condivisa fra diversi utenti. Ogni utente può collegarsi alla chat utilizzando un processo client. Durante la connessione il client visualizza sul terminale i messaggi in arrivo dagli altri utenti e permette all'utente connesso di interagire con la chat per ottenere la lista degli utenti connessi, per inviare un messaggio ad un utente specifico, per inviare un messaggio a tutti gli utenti connessi, o per disconnettersi.

Il server mantiene informazioni su tutti gli utenti connessi e sui socket su cui sono connessi utilizzando una tabella hash che contiene coppie

```
char * user, int sock_id
```

in cui `user` è il nome dell'utente (la chiave) e `sock_id` è l'identificativo del socket di comunicazione con quel determinato utente.

3 Il server

Il server viene attivato da shell con il comando

```
$ ./server pathsocket
```

Una volta attivato, il server **va immediatamente in background**, e inizializza l'ambiente necessario per gestire una nuova chat che utilizza per le connessioni il socket di path `pathsocket`. Il server il crea il socket `pathsocket` (se non esiste) e si mette in attesa di richieste di connessione da parte dei client.

Il server può essere terminato *gentilmente* in ogni momento della sua esecuzione inviando un segnale di `SIGINT` o `SIGTERM`. All'arrivo di questi segnali il server deve segnalare agli utenti connessi che il servizio di chat sta per essere interrotto, eliminare i socket di ascolto ed eventuali file temporanei e terminare a sua volta.

Il server è organizzato secondo il modello multithreaded: all'attivazione viene attivato un thread *dispatcher* che riceve richieste di connessione (`connect`) da parte dei client sul socket `pathsocket`. Alla ricezione di una connessione da parte dell'utente U il dispatcher crea un nuovo thread T_U dedicato a servire le richieste dell'utente U. Come prima cosa, T_U invia all'utente un ACK di connessione attiva sul socket di ascolto S_U del cliente U creato dalla connect. Il messaggio di ACK contiene la lista di tutti gli utenti connessi a quel determinato istante. Dopo aver inviato l'ack, il server si mette in attesa di richieste da U sullo stesso socket S_U. Le richieste effettuabili sono

- invio di un messaggio a un utente singolo (comando `one`)
- invio di un messaggio a tutti gli utenti connessi
- richiesta di informazioni sugli utenti connessi (comando `list`)
- richiesta di disconnessione (comando `exit`)

La ricezione di un comando `exit` provoca la disconnessione dell'utente dal sistema e la terminazione del thread T_U che ne gestisce le richieste.

Il protocollo di interazione è descritto dettagliatamente nella Sezione 5.

4 Il client

Il client è un comando Unix che può essere invocato con

```
bash:~$ ./client pathsocket nome_utente
```

```
[MESSAGE]: minnie
[MESSAGE]: pluto
[MESSAGE]: paperone




----
>
```

Figura 1: Finestra di interazione del client

```
[MESSAGE]: minnie
[MESSAGE]: pluto
[MESSAGE]: paperone
[pippo] ciao!




----
> ciao!
```

Figura 2: Client: invio di un messaggio a tutti (compreso se stesso)

all'invocazione il client contatta il server inviando una richesta di connessione sul socket di path `pathsocket`. Se la connessione ha successo, il client prende il controllo del terminale e presenta all'utente una finestra come in Fig. 1. La finestra è divisa logicamente in due parti. La *finestra di ascolto*, sopra la barra `----` e *la finestra di comando* (sotto la barra). Nella finestra di ascolto verranno visualizzati i messaggi in arrivo dagli utenti connessi o le risposte al comando list. La linea di comando sarà invece utilizzata per inviare richieste al server.

In Fig. 2 viene mostrata una richiesta di invio di un messaggio a tutti gli utenti connessi: è sufficiente digitare il testo del messaggio terminato da 'invio'. Invece, Fig. 3 mostra la richiesta di invio di un messaggio da *pippo* a *minnie*: `one` specifica che il messaggio deve essere mandato ad un solo utente destinatario, ed é seguito dal nome dell'utente destinatario (*minnie*) ed dal testo del messaggio (sempre terminato da invio). Fig. 4 mostra una possibile schermata di ricezione da parte dell'utente *minnie*.

```
[MESSAGE]: minnie
[MESSAGE]: pluto
[MESSAGE]: paperone
[pippo] ciao!



----
> one minnie ciao2 !
```

Figura 3: Client: invio di un messaggio a un singolo destinatario (`minnie`)

```
...
[pippo] ciao!
[paperone] come stai?
[MESSAGE]: [To you from pippo] ciao2 !


----
>
```

Figura 4: Ricezione di messaggi da parte dell'utente `minnie`

Ricapitolando, nella finestra di ascolto, sopra la barra, il client visualizza i messaggi in arrivo da vari utenti, preceduti dal nome dell'utente che li ha inviati. Nella linea di comando, invece, è possibile richiedere le seguenti operazioni: (1) lista degli utenti connessi

```
> list
```

che verrà visualizzata nella finestra di ascolto preceduta da `[MESSAGE]` (vedi Fig. 5), oppure (2) un messaggio a singolo utente

```
> one pippo testo messaggio
```

oppure (3) un messaggio a tutti gli utenti

```
> testo messaggio
```

oppure (4) terminare la sessione

```
> exit
bye
bash:~$
```

```
...
[MESSAGE]: [To you from pippo] ciao2 !
[MESSAGE]: Users List:
minnie
pippo
paperone

----
>
```

Figura 5: Output comando `list`

L'inserimento di un comando scorretto provoca la stampa di un breve messaggio di uso che riassume i comandi e la loro semantica. Il formato di questo messaggio è lasciato agli studenti.

4.1 La libreria libMainWindow

La gestione delle due finestre del client (ascolto e comando) non deve essere programmata direttamente ma deve essere effettuata attraverso la libreria `libMainWindow`, scaricabile dal sito del progetto. La libreria mette a disposizione sette funzioni che permettono di scrivere e leggere dalle due sottofinestre senza gestire direttamente l'interfaccia a caratteri con chiamate a basso livello (usa *ncurses*). Per l'uso della libreria rimandiamo alla documentazione allegata (scaricabile assieme alla libreria).

È inoltre disponibile una versione della libreria (*stubMainWindow*) che legge e scrive direttamente da standard input ed output, e che è utilizzata per i test automatici dei vari frammenti.

5 Interazione client-server

Server e client interagiscono utilizzando i *socket*. Il server al momento della sua attivazione crea un server socket `pathsocket` su cui ricevere le richieste di connessione dei client. Quando una richiesta di connessione è accettata viene automaticamente creato un nuovo socket che verrà poi usato per tutte le comunicazioni con il client (bidirezionale).

5.1 Formato dei messaggi

I messaggi scambiati fra server e client hanno la seguente struttura:

```
typedef struct {
   char type;
   unsigned int length;
   char* buffer;
} message_t;
```

Il campo `type` é un `char` (8 bit) che contiene il tipo del messaggio spedito. `type` puó assumere i seguenti valori:

```
#define MESSAGE_ERROR        'E'

#define MESSAGE_HELLO        'H'
#define MESSAGE_BYE          'B'

#define MESSAGE_ALL_SEND     'A'
#define MESSAGE_ALL_RECEIVE  'A'

#define MESSAGE_ONE_SEND     'O'
#define MESSAGE_ONE_RECEIVE  'O'

#define MESSAGE_LIST_QUERY   'L'
#define MESSAGE_LIST_REPLY   'L'
```

Il campo `length` é un `unsigned int` (32 bit) che indica la dimensione del campo `buffer`. Il suo valore comprende anche il terminatore di stringa '\0' finale. Vale `0` nel caso in cui il campo `buffer` non sia significativo. Il campo `buffer` é un puntatore alla stringa contenente il messaggio vero e proprio. La stringa ha i vari campi separati dal terminatore di stringa '\0'. **La stringa finisce sempre con il terminatore di stringa '\0'**.

5.2 Messaggi da Client a Server

Nei messaggi spediti dal Client al Server, il campo `type` puó assumere i seguenti valori:

`MESSAGE_HELLO` Messaggio di login, spedito dal Client quando desidera connettersi al Server. In questo caso il campo `buffer` contiene il nome dell'utente che si sta connettendo. Il Server risponderá con un messaggio di errore nel caso in cui il nome sia giá utilizzato, oppure con un messaggio contenente la lista degli utenti attualmente connessi nel caso in cui il nome dell'utente sia valido e non sia attualmente in uso.

`MESSAGE_BYE` Messaggio "exit", spedito dal Client quando desidera disconnettersi. Il campo `buffer` non contiene niente (perció il campo `lenght` contiene `0`). Alla ricezione di un messaggio di questo tipo il Server risponde al Client inviandogli un messaggio dello stesso tipo.

`MESSAGE_LIST_QUERY` Messaggio "list", spedito dal Client per richiedere la lista degli utenti attualmente connessi. Notare che il Server spedisce un messaggio contenente tale lista automaticamente quando il Client si connette. Il campo `buffer` non contiene niente (perció il campo `lenght` contiene `0`).

`MESSAGE_ONE_SEND` Messaggio "spedisci ad un solo utente", spedito dal Client quando desidera spedire un messaggio ad un particolare utente. Il campo `buffer` contiene il nome dell'utente al quale spedire il messaggio, seguito dal terminatore di stringa '\0', dal messaggio spedito, e da un altro terminatore di stringa '\0'.

`MESSAGE_ALL_SEND` Messaggio "spedisci a tutti gli utenti", spedito dal Client quando desidera spedire un messaggio a tutti gli utenti connessi. Il campo `buffer` contiene il messaggio spedito seguito da un terminatore di stringa '\0'.

5.3 Messaggi da Server a Client

Nei messaggi spediti dal Server a ciascun Client, il campo `type` puó assumere i seguenti valori:

`MESSAGE_BYE` Messaggio "exit", rispedito dal Server al Client quando quest'ultimo desidera disconnettersi. Il campo `buffer` non contiene niente (perció il campo `lenght` contiene `0`).

`MESSAGE_ERROR` Messaggio di errore. Questo tipo di messaggio puó essere spedito, ad esempio, quando un Client cerca di connettersi usando un nome per l'utente giá in uso. Il campo `buffer` contiene l'errore riscontrato.

`MESSAGE_LIST_REPLY` Messaggio "lista utenti connessi", contenente nel campo `buffer` la lista degli utenti correntemente connessi. La lista degli utenti connessi é separata dal terminatore di stringa '\0'. Questo messaggio viene spedito al Client in risposta ad una connessione riuscita (messaggio `MESSAGE_HELLO`), oppure in seguito ad un messaggio di richiesta di lista (messaggio `MESSAGE_LIST_QUERY`).

`MESSAGE_ALL_RECEIVE` Messaggio contenente un messaggio spedito da un utente a tutti gli utenti connessi. Il campo `buffer` contiene il nome dell'utente mittente, seguito dal terminatore di stringa '\0', seguito dal messaggio spedito.

`MESSAGE_ONE_RECEIVE` Messaggio contenente un messaggio spedito da un utente ad un altro utente determinato. Il campo `buffer` contiene il nome dell'utente mittente, seguito dal terminatore di stringa '\0', seguito dal messaggio spedito.

6 Funzioni e strutture dati

Le funzioni per la comunicazione operano sulla struttura dati `message_t`.

Il socket su cui il Sever attende nuove connessioni da parte dei client é di tipo

```
typedef int serverChannel_t;
```

Il canale di comunicazione bidirezionale tra Client e Server é di tipo

```
typedef int channel_t;
```

6.1 Funzioni usate dal Server

- `serverChannel_t createServerChannel(const char* path);`

  crea un nuovo Server di ascolto su una porta predefinita, e ritorna un `serverChannel_t` su cui attendere richieste di connessione da parte dei client.

- `int destroyServer(serverChannel_t s);`

  é usata per distruggere un server creato mediante la funzione `createServerChannel()`, ed é usata quando il Server deve terminare.

- `channel_t acceptConnection(serverChannel_t s);`

  é usata per accettare nuove connessioni da parte di un client. Ritorna un `channel_t` che puó essere usato per ricevere e spedire nuovi messaggi ad un determinato Client.

6.2 Funzioni usate dal Client

- `channel_t openConnection(const char* host);`

  é usata dal Client per creare una connessione col Server.

6.3 Funzioni usate da Client e Server

- `int receiveMessage(channel_t s, message_t *msg);`

  é usata per ricevere nuovi messaggi.

- `int sendMessage(channel_t s, message_t *msg);`

  é usata per spedire nuovi messaggi.

- `int closeConnection(channel_t s);`

  é usata per chiudere connessioni create mediante la `openConnection` (sul Client) o mediante la `acceptConnection` (sul Server).

7 Testing

I kit dei vari frammenti forniscono dei programmi di test minimali per il software prodotto.

L'uso dei programmi di testing è vivamente **sconsigliato** in sede di stesura e messa a punto del codice, quando invece lo studente dovrà costruirsi una propria strategia di test. Il testing dovrà invece essere effettuato a stesura completata, prima di consegnare i frammenti o il progetto ai docenti.

8 Istruzioni

Materiale fornito dai docenti
Per ogni frammento verranno forniti:

- file di dati adeguatamente formattati
- `main()` di test e verifica
- `makefile` per test e consegna
- uno o più file di intestazione (.h) con definizione dei prototipi e delle strutture dati (eventuali)
- `README` di istruzioni e di specifica del frammento

Cosa devono fare gli studenti
Gli studenti devono:

- leggere attentamente il README e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle con un insieme di test individuato allo scopo
- testare il software complessivo con il main fornito dai docenti (usando il makefile). *È importante che questo test venga effettuato solo su un programma già funzionante altrimenti i risultati possono essere di difficile interpretazione, fuorvianti o inutili.*
- preparare la *documentazione*: la documentazione richiesta per i primi due frammenti consiste in adeguati commenti al codice delle funzioni prodotte (per i vincoli sul codice prodotto e le richieste sui commenti consultare le Sez. 9.1 e 9.3). La documentazione del terzo frammento (progetto complessivo) deve contenere anche la relazione e quanto discusso nella Sezione 9.4.
- sottomettere il frammento esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README

9 Codice e documentazione

In questa sezione, vengono riportati alcuni requisiti del codice sviluppato e della relativa documentazione.

9.1 Vincoli sul codice

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo un makefile appropriato;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- gli errori devono essere gestiti adeguatamente e per ogni situazione anomala devono essere stampati messaggi di errore significativi su stderr (usando perror())
- NON devono essere utilizzate funzioni di temporizzazione quali `sleep()` o `alarm()` per risolvere problemi di race condition o deadlock fra i processi.
- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)
- si deve deallocare tutta la memoria (eventualmente controllando con `mcheck` le situazioni dubbie)
- si deve evitare l'uso di `strcpy`, `strcat` ed altre funzioni pericolose su stringhe non correttamente terminate

9.2 Struttura dei sorgenti

La struttura dovrà attenersi alle indicazioni date nella specifica: **i frammenti già consegnati possono essere modificati solo marginalmente per risolvere bug o problemi riscontrati successivamente**. È quindi necessario sviluppare ogni frammento con un occhio all'uso all'interno del progetto complessivo piuttosto che 'andare a braccio'.

9.3 Formato del codice

Il codice sorgente deve adottare la convenzione di indentazione tipica del C (vedi `indent` o l'indentazione automatica di `emacs` o `vim`) e una convenzione di commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file con: il nome ed il cognome dell'autore/i, la versione del programma ed il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore/i; firma dell'autore/i.
- un commento all'inizio di ogni funzione che specifichi l'uso della funzione (in modo sintetico), l'algoritmo utilizzato (se significativo), il significato delle variabili passate come parametri ed i valori che possono assumere, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per riferimento etc.

- un breve commento che spieghi il significato delle strutture dati e delle variabili globali (se esistono);
- un breve commento per i punti critici o che potrebbero risultare poco chiari alla lettura
- un breve commento all'atto della dichiarazione delle variabili locali, che spieghi l'uso che si intende farne

9.4 Documentazione del progetto complessivo

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 8 pagine) il cui scopo è quello di descrivere la struttura complessiva del lavoro svolto. La relazione *deve rendere comprensibile il lavoro svolto ad un estraneo, senza bisogno di leggere il codice se non per chiarire dettagli implementativi*. In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file ed in piú funzioni, librerie, etc.)
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- istruzioni per l'utente su come compilare/eseguire ed utilizzare il codice (in quale directory deve essere attivato, quali sono le assunzioni fatte, etc.) (da riportare anche nel README)

La relazione deve essere in formato PDF.