`msg`: un semplice server per scambio messaggio di testo

Progetto del modulo di laboratorio dei corsi di SO A/B 2009/10

Indice

1 Introduzione

Il modulo di Laboratorio di Programmazione di Sistema del corso di Sistemi Operativi e Laboratorio (277AA) prevede lo svolgimento di un progetto individuale suddiviso in tre frammenti. Questo documento descrive la struttura

complessivia del progetto e dei vari frammenti che lo compongono.

Il progetto consiste nello sviluppo del software relativo a `msg`: un sistema che permette lo scambio di messaggi di testo fra un insieme di client. Il software viene sviluppato e documentato utilizzando gli strumenti, le tecniche e le convenzioni presentati durante il corso.

1.1 Materiale in linea

Tutto il materiale relativo al corso puó essere reperito sul sito Web:

`http://www.cli.di.unipi.it/doku/doku.php/informatica/sol/laboratorio/`

Il sito verrà progressivamente aggiornato con le informazioni riguardanti il 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 di del corso durante l'orario di ricevimento, le ore in laboratorio e/o per posta elettronica.

1.2 Struttura del progetto e tempi di consegna

Il progetto deve essere sviluppato dallo studente individualmente e può essere consegnato entro il 1 Febbraio 2011.

La consegna del progetto avviene *esclusivamente* attraverso il target `consegna3` del `Makefile` contenuto nel kit di sviluppo del progetto. Eventualmente, una copia del tar creato dal target `consegna3` può essere allegata ad un normale messaggio di posta elettronica. Le consegne sono seguite da un messaggio di conferma da parte del docente all'indirizzo di mail da cui la consegna è stata effettuata. Se la ricezione non viene confermata entro 3/4 giorni lavorativi, contattare il docente per e-mail.

I progetti che non rispettano il formato o non consegnati con il target `consegna` non verranno accettati.

La data ultima di consegna è il 01/02/2011. Dopo questa data gli studenti dovranno svolgere il nuovo progetto previsto per il corso 2010/11.

Inoltre, gli studenti che consegnano una versione sufficiente del progetto finale entro il 30 Giugno 2010 accumuleranno il terzo bonus di 2, che contribuisce al voto finale con le modalità illustrate nelle slide introduttive del corso (vedi sito).

1.3 Valutazione del progetto

Al progetto viene assegnato un punteggio da 0 a 26 di cui 6 punti esclusivamente determinati dalla *qualità della documentazione allegata*. La valutazione del progetto è effettuata 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
- modalità di testing
- aderenza alle specifiche
- qualità del codice C e dei commenti
- chiarezza ed adeguatezza della relazione (vedi Sez. 9.3)

La prova orale tenderà a stabilire se lo studente è realmente l'autore del progetto consegnato e verterà su tutto il programma del corso. Il voto dell'orale (ancora da 0 a 26) fa media con la valutazione del progetto per delineare il voto finale. In particolare, l'orale comprenderà

- una discussione delle scelte implementative del progetto e dei frammenti
- l'impostazione e la scrittura di script bash e makefile
- l'impostazione e la scrittura di programmi C + PosiX non banali (sia sequenziali che concorrenti)
- domande su tutto il programma presentato durante il corso.

Casi particolari Gli studenti lavoratori iscritti alla laurea triennale in Informatica possono consegnare i due frammenti e il progetto finale in un'unica soluzione in qualsiasi momento dell'anno ed essere valutati con votazione da 0 a 30. In questo caso è necessaria la certificazione da consegnare al docente come da regolamento di ateneo.

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

Gli studenti iscritti ai vecchi ordinamenti (nei quali il voto di Lab. 4 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: `msg`

Lo scopo del progetto è lo sviluppo di un sistema client server per scambiare messaggi di testo.

Il sistema è costituito dai seguenti processi concorrenti:

- `msgcli`: il processo client (uno per ogni utente connesso) che si occupa di interagire con il server per inviare e ricevere messaggi per conto dell'utente,
- `msgserv`: il processo server che verifica se un utente puo' accedere al servizio, mantiene traccia di tutti gli utenti connessi, riceve e spedisce i messaggi da e per i client

`msgcli` e `msgserv` sono i due processi multithreaded che devono essere realizzati nel progetto didattico.

Inoltre, deve essere realizzato uno script bash (`logpro`) che, dato un file di messaggi inviati dal server lo analizza come specificato in Sez. 6.

3 Il server

`msgserv` viene attivato da shell con il comando

```
$ msgserv file_utenti_autorizzati file_log
```

dove

- `file_utenti_autorizzati` è il file di testo che contiene tutti i nomi degli utenti che sono autorizzati a collegarsi al servizio di messaggeria. Ogni utente e' specificato da una sequenza di caratteri lunghezza al piu' 256 contenente solo lettere minuscole o maiuscole dell'alfabeto inglese e cifre decimali. Il formato del file prevede un utente per ogni linea con separatore newline (`\n`).
- `file_log` è il file di testo utilizzato per il log dei messaggi inviati con successo dal server. Il formato del file prevede un messaggio per ogni linea con separatore newline (`\n`).

Il formato di ogni messaggio in `file_log` e'

```
mittente:destinatario:testo
```

dove `mittente` specifica il nome dell'utente mittente, `destinatario` specifica il nome dell'utente destinatario e `testo` specifica il testo del messaggio. Ad esempio:

```
minnie:minnie:ciao!
minnie:pluto:Sono arrivata
```

rappresentano due messaggi spediti con successo dal server. Il primo e' inviato dall'utente `minnie` a se stessa, il secondo da `minnie` all'utente `pippo.`

Si noti che i messaggi inviati in broadcast a piu' utenti sono presenti su piu' linee nel file di log, una per ogni messaggio inviato dal server con successo. Ad esempio, se l'utente `minnie` invia in broadcast il messaggio `ciao` mentre sono connessi anche `pippo` e `pluto` il file di log conterra' le linee:

```
minnie:minnie:ciao
minnie:pluto:ciao
minnie:pippo:ciao
```

L'ordine in cui le linee compaiono nel file di log e' irrilevante.

Il funzionamento del server è il seguente. Se `file_utenti_autorizzati` esiste viene aperto in lettura e caricato in una tabella hash. La chiave della tabella è il nome dell'utente e per ogni utente si memorizza il descrittore della socket su cui e' connesso o l'indicazione di utente disconnesso.

Dopo la creazione dell'hash viene creato il file di log `file_log` (se il file già esiste viene troncato) e viene creata una socket AF_UNIX

```
./tmp/msgsock
```

su cui i client apriranno le connessioni con il server per inviare/ricevere messaggi. Il server è multithreaded in quanto ogni utente connesso interagisce con un thread `worker` attivato allo scopo di servire tutte le richieste di quell'utente, fino alla sua disconnessione.

All'avvio il server crea anche un thread `writer` che raccoglie i messaggi di log dai `worker` e li scrive su file. La struttura dati in cui i messaggi vengono scritti dai `worker` e letti dal `writer` deve essere decisa dallo studente e spiegata nella relazione.

Il server può essere terminato *gentilmente* inviando un segnale di `SIGTERM` o `SIGINT`. All'arrivo di uno di questi segnali il server deve eliminare la socket `msgsock`, scrivere sul file di log eventuali messaggi ancora non registrati e ripulire l'ambiente. Si noti che all ricezione del segnale i thread che lo compongono il server devono terminare *gentilmente*, in modo che eventuali richieste pendenti da parte dei client vengano completate correttamente.

Quando il server è attivo accetta dai client richieste di connessione e richieste di invio messaggi ad utenti connessi. Il protocollo di interazione è descritto nella Sezione 5.

4 Il client

Il client è un comando Unix che viene attivato da linea di comando con

```
$ ./msgcli username
```

Appena attivato, il client effettua il parsing delle opzioni, e se il parsing è corretto cerca di collegarsi con il server (su `./tmp/msgsock`) per un massimo di 5 tentativi a distanza di 1 secondo l'uno dell'altro. Se il collegamento ha successo invia un messaggio di `CONNECT` per collegarsi. Se il collegamento è accettato dal server il client crea i propri thread e si mette in attesa di messaggi da inviare (su standard input) o di messaggi da altri utenti da visualizzare (dal server).

Processo `msgcli` è costituito da due thread paralleli:

- un thread che si occupa di leggere i messaggi da inviare da standard input, di verificarne la correttezza e di inviarli al server
- un thread che ascolta i messaggi in arrivo dalla socket di connessione con il server e li visualizza sullo standard output

Il formato con cui l'utente effettua richieste sullo standard input e' il seguente. Per i messaggi in broadcast e' sufficiente specificare il testo del messaggio terminato da newline `\n` es:

```
ciao a tutti!
```

il messaggio o puo' contenere qualsiasi carattere eccetto `%` che serve per introdurre gli altri comandi che sono

```
%LIST
```

per avere la lista di tutti gli utenti connessi,

```
%EXIT
```

per terminare la sessione ed il processo client e

```
%ONE
```

per inviare un messaggi ad un singolo utente, come ad esempio in

```
%ONE pippo ciao!
```

in questo caso la prima parola dopo `%ONE` (separatore ' ' spazio) viene interpretata come nome dell'utente e il resto (fino al newline) come testo del messaggio.

I messaggi in arrivo dal server vengono visualizzati sullo standard output in ordine di arrivo preceduti dal loro tipo, ad esempio:

```
[LIST] paperino pluto pippo
```

visualizza gli utenti connessi (preceduti da `[LIST]` e separati da spazi). Mentre i messagi in broadcast vengono visualizzati come:

```
[BCAST][pippo] Ciao a tutti!
```

e i messaggi da uno specifico utente come

```
[pippo] Ciao!
```

dove il nome dell'utente da cui arriva il messaggio viene visualizzato fra parentesi quadre. Notare che la visualizzazione di ogni messaggio e' terminato da newline `\n`.

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 e' lasciato allo studente.

Il processo può essere terminato *gentilmente* inviando un segnale di `SIGTERM` o `SIGINT`. All'arrivo di questo segnale il processo deve visualizzare i messaggi gia' ricevuti, informare il server della terminazione e terminare gentilmente.

5 Protocollo di interazione

Il server ed il client interagiscono con un socket *socket AF_UNIX*.

I client si connettono al server attraverso la socket `./tmp/msgsock`, creata all'avvio del server[1].

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:

[1] Sarebbe più logico creare la socket nella directory `/tmp/` di sistema, ma per non avere problemi di interazioni indesiderate fra progetti diversi sulle macchine del cli è meglio creare tutte le socket in una directory `./tmp/` locale alla directory di progetto

```
#define MSG_CONNECT      'C'
#define MSG_ERROR        'E'
#define MSG_OK           'O'
#define MSG_NO           'N'
#define MSG_TO_ONE       'T'
#define MSG_BCAST        'B'
#define MSG_LIST         'L'
#define MSG_EXIT         'X'
```

Il campo `length` é un `unsigned int` che indica la dimensione del campo `buffer`. Se buffer e una stringa il suo valore comprende anche il terminatore di stringa '\0' finale che deve essere presente nel buffer. Il campo `length` vale `0` nel caso in cui il campo `buffer` non sia significativo. Il campo `buffer` é un puntatore a un array di caratteri.

5.2 Messaggi da Client a Server

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

`MSG_CONNECT` 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 `MSG_OK` se la connessione e' autorizzata, oppure `MSG_ERROR` e un messaggio di errore nel caso in cui l'utente sia gia' connesso.

`MSG_EXIT` Messaggio di disconnessione. Il campo `buffer` non contiene niente (perció il campo `lenght` contiene `0`).

`MSG_LIST` Messaggio per richiedere la lista degli utenti attualmente connessi. Il campo `buffer` non contiene niente (perció il campo `lenght` contiene `0`).

`MSG_TO_ONE` Messaggio per inviare ad un particolare utente. Il campo `buffer` contiene il nome dell'utente al quale spedire il messaggio, seguito dal terminatore di stringa '\0' e dal messaggio spedito.

`MSG_BCAST` Messaggio per inviare a tutti gli utenti connessi. Il campo `buffer` contiene il messaggio spedito.

5.3 Messaggi da Server a Client

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

`MSG_ERROR` Messaggio di errore. Questo tipo di messaggio viene spedito quando si riscontra un errore. Il campo `buffer` contiene la stringa che definisce l'errore riscontrato.

`MSG_OK` Messaggio di risposta a un tentativo di connessione da parte di un nuovo utente. Segnala che la connessione e' possibile.

`MSG_LIST` messaggio di riposta a una richiesta di listare tutti gli utenti connessi. Nel campo buffer e' contenuta la lista di tutti gli utenti separati da uno spazio bianco.

`MESSAGE_BCAST` Messaggio contenente un messaggio spedito da un utente a tutti gli utenti connessi. Il campo `buffer` contiene il nome dell'utente mittente fra parentesi quadre, e il messaggio, come nel formato di stampa su standard output (Sez. 4).

`MESSAGE_TO_ONE` Messaggio contenente un messaggio spedito da un utente ad un altro utente. Il campo `buffer` contiene il nome dell'utente mittente fra parentesi quadre, e il messaggio, come nel formato di stampa su standard output (Sez. 4).

6 Lo script `logpro`

`logpro` è uno script bash che elabora off-line il file di log generato dal processo `msgserv` e calcola il numero di caratteri inviati da ciascun utente.

```
$ ./logpro logfile
```

dove `logfile` e' un file di testo (ASCII) che contiene i messaggi inviati secondo il formato specificato in Sez. 3.

Lo script deve controllare la validità dei suoi argomenti, scorrere il file `logfile` e per ogni utente deve calcolare il numero complessivo dei caratteri inviati in tutti i messaggi presenti nel log di cui risulta mittente.

Il costo deve essere stampato su standard output nel formato

```
utente caratteri
```

la stampa degli utenti deve essere nello stesso ordine lessicografico del comando `sort`. Ad esempio il seguente file

```
minnie 494
PaPeRino45 344
pluto 192
```

rileva un traffico di 494 caratteri per l'utente `minnie`, 344 per l'utente `PaPeRino45` e 192 per `pluto`.

L'idea e' quella di utilizzare lo script a fine giornata per calcolare il traffico generato per fini statistici o per addebitare il costo del traffico rilevato.

7 Istruzioni

7.1 Materiale fornito dai docenti

Nei kit del progetto vengono forniti

- funzioni di test e verifica
- `makefile` per test e consegna
- file di intestazione (.h) con definizione dei prototipi e delle strutture dati
- vari `README` di istruzioni

7.2 Cosa devono fare gli studenti

Gli studenti devono:

- leggere *attentamente* i README e capire il funzionamento del codice fornito dai docenti
- implementare le funzioni richieste e testarle
- verificare le funzioni con i test forniti dai docenti (attenzione: questi test vanno eseguiti su codice già corretto e funzionante altrimenti possono dare risultati fourvianti o di difficile interpretazione)
- preparare la *documentazione*: vedi la Sez. 9.
- sottomettere i tre frammenti del progetto esclusivamente utilizzando il makefile fornito e seguendo le istruzioni nel README.

8 Parti Opzionali

Possono essere realizzate funzionalità ed opzioni in più rispetto a quelle richieste (ad esempio altri comandi del client).

Le parti opzionali devono essere spiegate nella relazione e corredate di test appropriati.

9 Codice e documentazione

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

9.1 Vincoli sul codice

Makefile e codice devono compilare ed eseguire CORRETTAMENTE su (un sottinsieme non vuoto del) le macchine del CLI. Il README (o la relazione) deve specificare su quali macchine è possibile far girare correttamente il codice. Inoltre, se si usano software e librerie non presenti al CLI: (1) devono essere presenti nel tar TUTTI i file necessari per l'istallazione in locale del/i tool e (2) devono essere presenti nel makefile degli opportuni target per effettuare automaticamente l'installazione in locale. Se questa condizione non è verificata il progetto non viene accettato per la correzione.

La stesura del codice deve osservare i seguenti vincoli:

- la compilazione del codice deve avvenire definendo delle regole appropriate nella parte iniziale del makefile contenuto nel kit;
- il codice deve compilare senza errori o warning utilizzando le opzioni `-Wall -pedantic`
- NON devono essere utilizzate funzioni di temporizzazioni quali le `sleep()` o le `alarm()` per risolvere problemi di race condition o deadlock fra i processi/thread. Le soluzioni implementate devono necessariamente funzionare qualsiasi sia lo scheduling dei processi/thread coinvolti

- DEVONO essere usati dei nomi significativi per le costanti nel codice (con opportune `#define` o `enum`)

9.2 Formato del codice

Il codice sorgente deve adottare una convenzione di indentazione e commenti chiara e coerente. In particolare deve contenere

- una intestazione per ogni file che contiene: il nome ed il cognome dell'autore, la matricola, il nome del programma; dichiarazione che il programma è, in ogni sua parte, opera originale dell'autore; firma dell'autore.
- 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, eventuali variabili globali utilizzate, effetti collaterali sui parametri passati per puntatore 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

9.3 Relazione

La documentazione del progetto consiste nei commenti al codice e in una breve relazione (massimo 10 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 particolaer NON devono essere ripetute le specifiche contenute in questo documento.* In pratica la relazione deve contenere:

- le principali scelte di progetto (strutture dati principali, algoritmi fondamentali e loro motivazioni)
- la strutturazione del codice (logica della divisione su più file, librerie etc.)
- la struttura dei programmi sviluppati
- la struttura dei programmi di test (se ce ne sono)
- le difficoltà incontrate e le soluzioni adottate
- quanto altro si ritiene essenziale alla comprensione del lavoro svolto
- README di istruzioni su come compilare/eseguire ed utilizzare il codice

La relazione deve essere in formato PDF.