Lorenzo Rossi, matr. 20031485 e-mail: 20031485@studenti.uniupo.it

Relazione di laboratorio per il corso di Reti I

È stato richiesto di scrivere un'applicazione client-server che realizzi una comunicazione tra due programmi seguendo un protocollo. In particolare, i messaggi trasmessi tra client e server saranno identificati dalle seguenti keywords:

- **TEXT** per richiedere o fornire un'analisi del testo;
- **HIST** per richiedere o fornire la distribuzione dei caratteri ASCII 32 a 7 bit memorizzati dal server;
- **EXIT** per richiedere o fornire la distribuzione dei caratteri ASCII 32 a 7 bit memorizzati dal server e, successivamente, chiudere la connessione su socket;
- **QUIT** per chiudere la connessione su socket, senza invio di dati dal server.

Questi quattro comandi sono preceduti, nei messaggi mandati dal server, da una stringa che specifica l'esito di una richiesta del client:

- **OK** per comunicare correttezza ed accettazione di un comando;
- **ERR** per comunicare la ricezione di un comando errato.

I programmi client e server dialogano su un socket identificato dall'IPv6 a cui alloggia il server e dal numero di porta su cui questo lancia la funzione *listen()* per poter accettare nuove connessioni.

Note preliminari

La cartella 20031485.zip contiene i seguenti documenti di testo:

- **client.c** e **server.c**, i codici sorgente dei programmi client e server;
- **library.h**, contenente variabili globali (tra cui le dimensioni dei buffer e i codici numerici identificativi dei comandi), stringhe statiche corrispondenti ai delimitatori del protocollo e ad eventuali messaggi di errore, e i prototipi delle funzioni utilizzate nei programmi principali;
- **library.c**, contenente le definizioni delle funzioni utilizzate dal client e dal server:
  - **Funzioni del client:**
    - *void print_file(char *filename):* dato il nome di un file passato come argomento, stampa a schermo il contenuto del file carattere per carattere.
    - *int valid_chars(char *string):* data una stringa passata come argomento, restituisce l'intero corrispondente ai caratteri ASCII 32 della classe *alnum*.
    - *void print_hashtogram(char *string):* data la stringa contenente le coppie *<carattere>:<quantità>* ricevuta dal server e ripulita dei delimitatori protocollari, il client stampa a schermo un numero di *#* indicativo delle numero di copie di ogni carattere inviato al server.
  - **Funzioni del server:**
    - *void string_to_hist(char *string, int hist[], int dim):* data una stringa, un array di interi (ad ogni indice del quale corrisponde un carattere ASCII 32 di classe *alnum*) e la sua dimensione *dim* (verrà passata la variabile *HIST_DIM* = 62), la funzione incrementa l'elemento in posizione *i*-esima di *hist[]* ogni volta che il carattere ad esso corrispondente compare nella stringa *string*.
    - *void print_hist(char *string, int hist[], int dim):* data una stringa vuota, l'array *hist[]* definite come nella precedente funzione e la sua dimensione (di nuovo identificabile con *HIST_DIM*), la funzione scrive su *string* le coppie *<carattere>:<quantità>* dove *<quantità>* è l'intero alla *i*-esima posizione dell'array *hist[]*, a cui corrisponde proprio *<carattere>* (ad esempio, se *hist[5]=10*, dato che l'indice 5 corrisponde al carattere F (per come la funzione è implementata), su *string* verrà caricato *F:10*).
- **client.c** e **server.c**, i codici sorgente dei programmi client e server;
- **welcome.txt** e **options.txt**, due files di testo che contengono rispettivamente un messaggio di benvenuto e le opzioni selezionabili dall'utente;
- **dante.txt** e **manzoni.txt**, due files di testo che l'utente può usare per testare il funzionamento dell'applicazione
- **makefile**, per la compilazione dei sorgenti ed il linking a *library.c*.

Dopo la compilazione ed il lancio, i due programmi saranno in grado di dialogare sul socket a cui sono connessi.
Nei due programmi client e server sono stati utilizzati dei *flag*, ad esempio per rimanere in un certo ciclo *while* finchè il flag non viene impostato, oppure per accodare due *case* in uno stesso *switch* (come nel caso del client che invia il comando *EXIT* al server).
Tutte le stringhe scambiate tra client e server sono terminate dal carattere "\n".

Server

Il programma server viene lanciato con, come argomento, la porta da cui attendere connessioni con il comando:

./server <port_no>

Il server è strutturato su due parti, una di gestione della connessione ed una di elaborazione dei messaggi ricevuti ed invio di accettazione o negazione dei comandi ricevuti. Quest'ultima parte è composta da un ciclo *while* che riassume le azioni compiute dal server in lettura dei comandi dal server (*read()*) e risposta (*write()*). La richiesta del client verrà convogliata nello scope di elaborazione corretto grazie ad uno *switch* interno al ciclo operativo del server.

La prima interazione di protocollo tra client e server è eseguita dal server e consiste nell'invio al client di una stringa di benvenuto del tipo *OK START <messaggio di benvenuto>.*

Una volta inviata questa stringa, il server si mette in ascolto delle richieste del client, da cui potrà ricevere quattro macrocategorie di comandi che occuperanno la testa della stringa ricevuta:

- **TEXT:** il server si aspetta che un comando *TEXT* sia seguito dalla stringa da analizzare e che l'elemento immediatamente precedente il delimitatore di protocollo "\n" sia il contatore dei caratteri ASCII 32 presenti nel testo inviato. Il server chiama la funzione *string_to_hist()* per memorizzare tutti i caratteri ASCII 32 ricevuti dal client (senza i delimitatori di protocollo, opportunamente rimossi) in un array *hist[]* di 62 elementi, dove ad ogni indice corrisponde il numero di copie di un certo carattere che sono state finora ricevute. Nel caso dell'accettazione della stringa mandata dal client, il server elaborerà i dati ricevuti, controllando la coerenza tra il contatore ricevuto ed il conteggio eseguito personalmente, e risponderà al client con una stringa del tipo *OK TEXT <contatore>* se non ci sono stati errori. Gli errori relativi al comando *TEXT* possono essere sintattici o semantici: in caso di errore semantico, il server invierà al client una stringa del tipo *ERR TEXT <messaggio di errore>*; in caso di errore sintattico, invece, il server invierà *ERR SYNTAX <messaggio di errore>.* Dopo aver inviato un qualunque *ERR*, il server chiude la connessione con il client con *close()*.
- **HIST:** il server prepara una stringa composta da *OK HIST* (necessari per dichiarare l'accettazione del comando) e richiama la funzione *print_hist()*, che stampa coppie *"<lettera>:<n_copie>"* su una stringa data in input, dove *<n_copie>* è il numero di copie di ciascun carattere (memorizzato nell'array *hist[]*). Nel caso in cui una lettera abbia ricevuto 0 copie, la coppia corrispondente non verrà inserita nella stringa. Se la stringa da inviare è complessivamente più lunga del buffer, il server invierà più messaggi che iniziano con *OK HIST* e terminano con "\n" e che conterranno le parti della stringa di partenza. Quando l'ultimo dato è stato inserito nel messaggio per il client, questo viene inviato e viene preparata ed inviata la stringa *OK HIST END*, che sancisce la fine dell'esecuzione del comando *HIST* (questa stringa viene inviata indipendentemente dal numero di caratteri finora ricevuti, che può essere anche zero);
- **EXIT:** il server si comporta esattamente come se avesse ricevuto un HIST, con l'unica differenza che, dopo aver inviato *OK HIST END*, invia un altro messaggio del tipo *OK EXIT <messaggio di commiato>* e chiude la connessione sul socket con la funzione standard *close()*;
- **QUIT:** il server invia al client una stringa del tipo *OK QUIT <messaggio di commiato>* e chiude la connessione con *close()*.

Client

Il programma client viene lanciato con, come argomenti, l'indirizzo IPv6 a cui alloggia il server ed il numero di porta da cui quest'ultimo attende richieste di connessione, digitando:

./client <IP_addr > <port_no>.

Il programma client è strutturato su due parti: una parte di gestione della connessione al socket ed una parte di ciclo operativo protocollare. Quest'ultimo è composto da un ciclo *while* che si apre con una *read()* ad un ipotetico messaggio del server, che sappiamo essere il primo dei due ad interloquire. A seconda del messaggio ricevuto in risposta, il client si comporterà di conseguenza.

Se è stata ricevuta una stringa che inizia per OK, guardo qual è il comando seguente nella stringa:

- **START:** il server riferisce che la connessione è appena cominciata ed il client, continuando a leggere sulla stringa, si aspetta un messaggio di benvenuto.
- **TEXT:** il server ha analizzato con successo un testo che ha precedentemente ricevuto dal client, e notifica che non ci sono stati errori di formulazione della richiesta. Continuando a leggere sulla stringa, il client si aspetta di trovare il numero dei caratteri ASCII 32 inviati nella stringa precedente.
- **EXIT o QUIT:** il server ha accolto la richiesta di disconnessione, quindi il client può tranquillamente chiamare la funzione *close()*.

Se è stata ricevuta una stringa che inizia per ERR, questa prosegue con identificatore del tipo di errore:

- **TEXT:** errore semantico commesso dal client;
- **SYNTAX:** errore sintattico commesso dal client;

Dopo ogni identificatore del tipo di errore, ci si aspetta di leggere una stringa contenente una descrizione dell'errore che viene stampata a schermo per l'utente.

Dopo il controllo in lettura, il client presenta un altro ciclo *while* contenente le varie richieste che possono essere formulate al server, elencate in uno *switch*. Tramite la stampa di un file di testo contenente indicazioni per l'utente, è possibile selezionare una (alla volta) tra sette opzioni:

- **TEXT_BY_STRING:** premendo 1, l'utente può digitare una stringa lunga fino a 2048 caratteri totali per mandarla al server. Se i caratteri inseriti sono troppi per il buffer, la stringa inserita in input verrà suddivisa in più sottostringhe della dimensione adatta. Ad ognuna di esse verrà anteposto il delimitatore *TEXT* e verrà fatto un conteggio dei caratteri validi tramite la funzione *valid_chars()*. Questa cifra verrà messa in coda alla stringa, che verrà terminata da un carattere "\n" prima dell'invio. Infine si uscirà dallo *switch*, rientrando nel *while* più esterno e, quindi, accettando un messaggio del tipo *OK TEXT <contatore>* dal server se non ci sono stati errori nell'invio della stringa.
- **TEXT_BY_FILE:** premendo 2, l'utente può digitare il pathname di un file di testo, il quale verrà caricato carattere per carattere sul buffer e verrà inviato al server per l'analisi del testo. Testi che hanno un numero di caratteri superiore alla capienza del buffer verranno inviati a pacchetti di *sizeof(buffer)* bytes (includendo i delimitatori del protocollo) e, ad ogni pacchetto ricevuto, il server risponderà al client con una stringa *OK TEXT <contatore>*. Ad ogni stringa di questo tipo che il client riceve, viene incrementato un contatore di *<contatore>* unità, che verrà stampato a schermo solo dopo che è stato accettato dal server l'ultimo pacchetto. In questo modo, l'utente visualizzerà la totalità dei caratteri ASCII 32 che componevano il testo inviato al server.
- **TEXT_BY_WORD:** premendo 3, l'utente può inviare al server una parola alla volta grazie ad un ciclo di controllo: dopo aver inserito ogni parola, si può infatti scegliere se continuare ad inserirne oppure se uscire. Anche digitando più combinazioni di caratteri ASCII 32 separate da spazi, verranno considerati al più solamente i primi 39 caratteri digitati. *TEXT_BY_WORD* è equivalente al comando *TEXT_BY_STRING*, con la differenza che la lunghezza è ridotta e che viene considerato solo il contenuto presente prima di uno spazio, se presente. La parola inserita diventa parte di una stringa del tipo *TEXT <parola> <contatore>*, dove il contatore è ottenuto chiamando *valid_chars()* su *<parola>*. Si avrà un feedback dal server ogni volta che verrà premuto invio dopo aver digitato una parola.
- **GET_HIST:** premendo 4, il client invierà al server una stringa contenente il comando *HIST*. Il server allora si comporterà come descritto in **HIST** nella sezione **Server**, scrivendo sul socket le coppie *<carattere>:<quantità>* che ha memorizzato. Il client leggerà dal socket questi dati carattere per carattere, rimuovendo i delimitatori di protocollo e concatenandoli in una stringa d'appoggio che verrà passata alla funzione *print_hashtogram()*, che stamperà a schermo un istogramma grafico della distribuzione dei caratteri.

- **GET_HIST (EXIT):** se l'utente ha premuto 5, il caso precedente viene leggermente alterato. Non viene più inviato al server un comando *HIST* ma un *EXIT*. La ricezione dei dati dell'istogramma non cambia fino alla lettura della stringa *OK HIST END*, ma in questo caso si esce dallo *switch* per rientrare nella prima parte del ciclo operativo del client, in cui si attende un messaggio inviato dal server. Dopo aver mandato *EXIT* ed aver ricevuto l'istogramma, il client si aspetta una stringa del tipo *OK EXIT <messaggio di commiato>*. Dopo i controlli e la rimozione dei delimitatori protocollari, viene stampato a schermo il *<messaggio di commiato>* inviato dal server e il client chiude la connessione al socket con *close()*.
- **EXIT_NO_HIST:** premendo 6, il client invia al server una stringa contenente unicamente il comando *QUIT*, uscendo dallo *switch* per rientrare nel *while* di lettura dei messaggi del server. Se il server ha ricevuto *QUIT*, risponde con una stringa del tipo *OK QUIT <messaggio di commiato>*, che viene letta dal client e privata dei suoi delimitatori protocollari. Il client stampa poi a schermo il *<messaggio di commiato>* del server e chiude la connessione al socket con *close()*.
- **OPTIONS_AGAIN:** premendo 7, viene stampato a schermo il file di testo *options.txt*, contenente le possibili opzioni selezionabili dall'utente. In questo caso, non vi è interazione con il server.
- **default:** se per caso dovesse persistere un errore di assegnazione della variabile *option* (argomento dello *switch*), viene stampato un messaggio di errore e viene permesso all'utente di provare ad inserire un'opzione valida. Anche in questo caso, non vi è interazione con il server.