Esercitazione 2020-10-09

Scambio di messaggi client-server TCP - Esercizio 1.5

In questo esercizio dobbiamo inviare dal client al server un username, ed il server deve rispondere con una stringa del tipo "follen, sei il 2° utente del server!".

In aggiunta, dobbiamo inviare **solo i byte strettamente necessari** al server, quindi è necessario inviare prima un messaggio di informazione che dice al server quanti byte riceverà, e poi inviare i dati effettivi:

Lato client

```
printf("Inserisci il tuo nome: "); scanf("%s", buf);
  int len = strlen(buf);

  write(sd, &len, sizeof(len));

  printf("Byte trasmessi: %i\n",write(sd, &buf, strlen(buf)));
```

Andiamo quindi a leggere da tastiera l'username e ne calcoliamo l'effettiva lunghezza. Effettuiamo una prima write al server dove inviamo la lunghezza dell'username, successivamente inviamo l'username.

Lato server

```
int len = 0;
  read(sd2, &len, sizeof(len));

  int n = 0;
  while (n < len)
     {
        n += read(sd2, buf + n, len - n);
     }
```

Lato server andiamo prima a leggere la lunghezza dell'username, dopodichè leggiamo **al più** len bytes. Questo perchè **non sappiamo a priori** la lunghezza dei pacchetti inviati con il protocollo TCP, e dobbiamo quindi **leggere finchè i bytes non sono stati letti totalmente**.

Cosa vede wireshark?

Connessione preliminare

Come sappiamo, con il protocollo TCP viene eseguita una connessione preliminare tra i due endpoint.

Invio della lunghezza dell'username al server

Viene poi inviata la lunghezza della stringa ("follen") al server; infatti i bytes da inviare saranno 6.

Invio dell'username

Viene finalmente inviata la stringa contenente l'username.

Risposta del server

Il server risponde con una stringa (questa volta di 256 bytes) contenente la risposta.

🏁 1:07

Esercizi 2.x

Esercizio 2.1

Il primo esercizio è una variante degli esercizi visti fin ora, e prevede che il client legga da stdin delle stringhe in **modo continuo**, e che le invii al server, che non fa altro che stampare la stringa ricevuta sullo stdout.

La lettura delle stringhe da parte del client e la spedizione verso il server continua finchè non si leggerà una stringa che finisce con ".".

Esercizio 2.2 - Peer to Peer

L'API che abbiamo usato fin'ora è stata progettata per operare in architettura client-server; questo perchè da lato server invochiamo sulla socket costruita prima la listen() e poi l'accept(), che non usiamo lato client.

Partiamo da un modello client-server e proviamo a realizzare un canale di comunicazione p2p.

Il problema: se facciamo in modo che uno dei processi si ponga in ascolto per ricevere dei messaggi di richiesta dall'altro processo, non potrà inviare messaggi; questo perchè la listen è bloccante.

La soluzione: invece di usare il canale di comunicazione in maniera bidirezionale (cosa possibile tramite al protocollo TCP), possiamo pensare di sfruttare parzialmente il canale di comunicazione: una volta da A->B, ed un'altra da B->A; comunichiamo quindi in modo alterno.

Peer To Peer - Chat Half-Duplex

Invece di parlare di server e di client, parliamo di **peer**. Avremo, a seguito della costruzione iniziale del canale (sempre necessaria), il canale di comunicazione TCP diventa un canale che può essere usato in maniera **simmetrica** dai due interlocutori.

Il codice da scrivere per dare vita ai due processi, è praticamente lo stesso; sarà però eseguito in maniera diversa, in modo da tenere in considerazione **lo stato diverso** che i due peer avranno in ogni momento.

Quando il peer1 è abilitato a scrivere sul canale, il peer2 è abilitato a leggere. Se i due peer operano in questo stato, non ci sono problemi di "conflitto" sul canale, visto che solo un peer alla volta lo usa per scrivere.

Periodicamente, lo stato cambia; questo vuol dire che quando viene inviato un carattere speciale (ad esempio un trattino), viene cambiato lo stato, quindi se inizialmente il primo peer scriveva ed il secondo leggeva, con l'invio del carattere speciale il primo peer legge ed il secondo scrive.
Quando viene inviato un secondo carattere speciale (ad esempio un punto), **da**

uno dei due peer, la connessione viene chiusa.

Il codice

La parte più importante del codice di uno dei due processi è sostanzialmente un loop, che caratterizza un automa a stati. Abbiamo tre stati: **stato YOU, stato PEER e stato EXIT**; inizialmente il client è nello stato YOU, mentre il "server" è nello stato PEER.

All'interno del loop principale vi è uno **switch case**, che a seconda dello stato pone il processo nella fase di lettura o scrittura.

Fase di scrittura - YOU

Durante la fase di scrittura viene letta da tastiera una stringa, ne viene calcolata l'effettiva lunghezza che viene comunicata al "server", e poi viene inviata. Viene controllato l'ultimo carattere della stringa:

- Se l'ultimo carattere è '.' lo status viene cambiato in EXIT.
- Se l'ultimo carattere è '-' lo status viene cambiato in PEER, ed il controllo viene trasferito all'altro processo.
- In tutti gli altri casi non succede nulla.

Viene poi scritta sul canale di comunicazione prima la lunghezza della stringa letta, e poi viene inviata la stringa effettiva, che il server leggerà.

Questo processo viene ripetuto finchè il controllo non viene affidato all'altro processo.

Fase di lettura - PEER

Anche in questo caso abbiamo un loop; viene letta prima la lunghezza della stringa da ricevere, e poi la stringa stessa. ancora una volta si controlla la stringa:

- Se l'ultimo carattere è '.' lo status viene cambiato in EXIT.
- Se l'ultimo carattere è '-' lo status viene cambiato in YOU, ed il controllo è del processo corrente.
- In tutti gli altri casi non succede nulla.

Differenza tra i due processi

Ovviamente il codice tra i due processi è molto simile, ma diverso nella fase iniziale; questo perchè inizialmente uno dei due processi deve predisporsi a **ricevere una connessione**, quindi se dal client abbiamo una connect, dal "server" avremo la **listen() e poi accept()**.

Dopo questa fase iniziale, il codice è lo stesso.

Fine lezione 8