

Programmazione in ambienti distribuiti I

(01FQT, 01FEL, 06EKE, 02HDF)

AA 2009-2010, Esercitazione di laboratorio n. 1 , 2 , 3

Traccia di soluzione

Esercizio 1.2 (connessione)

Usare le funzioni `Socket()` e `Connect()`, quest'ultima dopo aver risolto l'indirizzo e la porta tramite la funzione `Getaddrinfo()` o in alternativa, sconsigliate, le funzioni `inet_addr()` o `inet_aton()`, quest'ultima sconsigliata perche' non portabile su IPv6.

Esercizio 1.3 (dati ASCII)

Procedere come nell'esercizio precedente. Scrivere sul socket tramite `Write()` o `Send()` e leggere tramite, per esempio, `Readline()`. Tramite il valore di ritorno di una `sscanf()` e' possibile distinguere tra un messaggio di errore ed il numero che rappresenta la somma, e stamparli nella maniera appropriata.

Esercizio 1.4 (client-server UDP base)

Usare le funzioni `Socket()`, `Sendto()` e `Recvfrom()`. In alternativa e' possibile utilizzare la `Connect()` per impostare l'indirizzo di destinazione una volta per tutte e poi utilizzare la `Send()` per spedire.

Per gestire il fatto che il client non ha ricevuto risposta entro un certo tempo si possono usare due approcci: utilizzare la `Select()` o il segnale `SIGALRM`.

Con la `Select()`: chiamarla prima di leggere dal socket, per evitare di bloccarsi in lettura. Impostare il timeout al valore desiderato. Come primo parametro si puo' utilizzare il file descriptor che corrisponde al socket + 1.

Con il segnale `SIGALRM`: registrare una funzione che gestisce il segnale tramite la funzione `sigaction`. Prima di chiamare la `Recvfrom_timeout()`, impostare un timeout tramite la funzione `alarm()`. Se nessun pacchetto viene ricevuto, la `Recvfrom_timeout()` verra' interrotta dal segnale se l'ultimo parametro e' impostato a 1, cosicche' il programma possa terminare stampando il messaggio corretto, sulla base del valore di ritorno della `Recvfrom_timeout()`.

Per il server UDP, usare le funzioni `Socket()` e `Bind()` registrandosi, per esempio, all'indirizzo `INADDR_ANY`, e inserire in un ciclo infinito la coppia `Recvfrom()` e `Sendto()`. Si ponga attenzione a inviare solo la parte valida del buffer letto tramite la `Recvfrom()`.

Esercizio 2.1 (client UDP perseverante)

Utilizzare lo stesso approccio dell'esercizio precedente, inserendo i tentativi di spedizione in un ciclo opportuno.

Esercizio 2.2 (server UDP limitante)

Utilizzare lo stesso approccio dell'esercizio sul server UDP, memorizzando ogni volta l'indirizzo di provenienza del pacchetto in un vettore di 10 elementi, che simuli un buffer circolare. Per memorizzare l'indirizzo, utilizzare un vettore di strutture `sockaddr_storage` cosi' da facilitare la portabilita' del codice su IPv6 (che puo' essere realizzata come utile esercizio).

Esercizio 2.3 (server TCP iterativo)

Usare le funzioni `Socket()`, `Bind()` e `Listen()`. Inserire la funzione `Accept()` in un ciclo infinito, per gestire il servizio al client successivo una volta che il primo client abbia chiuso la connessione.

Dopo la `Accept()`, un secondo ciclo infinito riceve comandi dal client e invia il file richiesto o un messaggio di errore. Dal ciclo infinito di ricezione dei comandi si esce quando si riceve il comando `QUIT` (la connessione deve essere chiusa lato server tramite `Close()`) oppure quando non e' piu' possibile usare il socket. In quest'ultimo caso stampare il messaggio che la connessione e' stata chiusa dal client, e chiudere tramite `Close()` lato server la connessione. Attenzione a non utilizzare la `Readline()` per leggere i comandi dal client, ma realizzare in proprio il ciclo di lettura carattere per carattere fino al '\n', in quanto la `Readline()` bufferizza l'input e potrebbe creare problemi nelle successive `Read()`.

Esercizio 2.4 (dati in standard XDR)

Procedere come nel caso della realizzazione di un client TCP. Creare un buffer XDR tramite la `xdr_create()`, inserire i numeri tramite `xdr_int()`, e rilevare la quantita' di byte da inviare tramite la `xdr_getpos()`. In ricezione, poiche' non e' nota a priori, in generale, la quantita' di byte da leggere, per mantenere il codice portabile utilizzare `fdopen()` per ricavare un puntatore `FILE *` al socket, e passarlo alla `xdrstdio_create()` per inizializzare la struttura dati XDR. Procedere poi con chiamate `xdr_int()` secondo le necessita'. Per gestire l'eventuale messaggio di errore modificare sia client sia server introducendo un'informazione aggiuntiva che evidenzia se e' presente un errore, prima dell'intero o della stringa.

Esercizio 3.1 (server TCP concorrente)

Usare le funzioni `Socket()`, `Bind()` e `Listen()`. Inserire la funzione `Accept()` in un ciclo infinito, che subito dopo la `Accept()` chiama la `Fork()`. Nella parte del figlio dell'if sulla `Fork()` si procede come nell'esercizio 2.3 (ciclo infinito, con lettura dei comandi; si esce dal ciclo con comando `QUIT` o se l'uso del socket corrispondente alla connessione ritorna errore).

Per limitare il numero di processi a tre, una possibile soluzione e' la seguente. Mantenere nel padre un contatore di numero di figli, incrementato dopo l'`Accept()` nel ramo if della `Fork()` che rappresenta il padre. Prima di eseguire l'`Accept()` nel ciclo infinito, testare il valore del contatore di figli. Se tale valore e' uguale a tre, chiamare la funzione `wait()` in modalita' bloccante (che e' il default). Quando la funzione ritorna, decrementare il numero di figli di uno.

La soluzione precedente rischia di non accorgersi della terminazione dei figli: se il processo padre e' bloccato, per esempio nella `Accept()`, e due figli terminano con il contatore che vale tre, la `wait` nell'if precedente diminuisce di uno ma per il secondo figlio non viene effettuata la `wait()`. Una soluzione puo' essere, dopo il test sul numero di figli prima della `Accept()`, effettuare un ciclo di `waitpid()` non bloccanti (utilizzando l'opzione `WNOHANG` della `waitpid()`) finche' ci sono figli in attesa di `wait`, verificabile tramite il valore di ritorno.

L'accorgimento precedente risolve il problema del corretto aggiornamento del numero dei figli, ma finche' il flusso del programma non arriva nuovamente al ciclo sulla `wait()`, i figli che hanno terminato rimangono nel sistema come zombie (per esempio se il padre e' bloccato nella `Accept()`). Per evitare di avere figli zombie nel sistema e' necessario registrare un handler del segnale `SIGCHLD` che viene inviato al padre ogni volta che un processo figlio termina. Nell'handler verra' effettuato il ciclo di `waitpid()` non bloccanti. E' necessario un ciclo e non una singola `waitpid()` perche' non e' garantito che sia inviato un singolo segnale per ogni figlio che termina (per esempio se il flusso di esecuzione del processo padre si trova nell'handler, dopo aver effettuato le `waitpid()`, e nel frattempo terminano due figli, l'handler verra' nuovamente richiamato, ma una volta sola).

Notare che catturare il segnale `SIGCHLD` rende superfluo il ciclo di `waitpid()` non bloccanti prima della `Accept()`.

Esercizio 3.2 (client TCP con multiplexing)

Per gestire contemporaneamente input da utente e socket di comunicazione con il server, e' possibile procedere come segue. Dopo aver aperto il socket ed essersi collegati con il server, entrare in un ciclo infinito, la prima operazione delle quali e' una `Select()` sul socket e sul file descriptor corrispondente allo standard input (ossia zero, ricavabile anche come `fileno(stdin)`). I bit corrispondenti al socket e al file descriptor dello standard input si impostano con due chiamate a `FD_SET()`. La `Select()` avra' come timeout `NULL`, ossia attendera' finche' almeno uno dei due file descriptor si rende disponibile.

A questo punto due if indipendenti, di pari livello, verificheranno tramite `FD_ISSET()` se il socket si e' sbloccato e/o se lo standard input si e' sbloccato. Se il socket si e' sbloccato, si opera su di esso, tramite una lettura dallo stesso. Se lo standard input si e' sbloccato, si legge una riga dallo standard input e si interpreta il comando, eseguendo l'istruzione opportuna. Nel caso della `GET` si scrive immediatamente il comando sul socket (operazione sempre possibile).

Notare che non e' possibile effettuare piu' di una lettura dal socket nel primo if, perche' non c'e' garanzia che la seconda lettura dal socket non sia bloccante, cosa che e' invece assolutamente necessaria per garantire il multiplexing sempre attivo. Nel caso di lettura del file, quindi, prima di ogni lettura e' necessario che il flusso di esecuzione ripassi dalla `Select()`. Si puo' separare lo stato di lettura della risposta dal server (ricezione di "+OK") e del file memorizzandolo in un opportuno flag, per sapere dove vanno messi i dati da leggere. Notare inoltre che anche la lettura della risposta dal server va effettuata in modo da non bloccarsi, e non e' garantito che tutto il messaggio "+OK" sia ricevuto tramite un'unica lettura. Si deve quindi aver cura di mantenere lo stato di lettura dal socket per sapere se e' necessario terminare di leggere il comando o si sta gia' leggendo il file e i dati sono quindi relativi al file letto.

Notare infine che l'input da utente e' bufferizzato, quindi la `Select()` ritorna per lo sblocco dello standard input solamente nel caso di pressione di tasto `ENTER`. L'input di caratteri non e' mai bloccato, ma essi come accade solitamente in C vengono passati e gestiti dal programma solamente a fronte della pressione del tasto `ENTER`.

Si noti che una soluzione alternativa che utilizzi un processo figlio separato per la ricezione del file consente di avere l'input dall'utente non bloccato, in quanto gestito nel processo padre, ma complica la gestione del comando "A", che deve interrompere immediatamente il trasferimento. Tale funzionalita' puo' essere infatti solamente introdotta inviando un segnale, tramite `kill()`, al processo figlio, per terminarlo. La soluzione proposta, invece, non richiede la creazione di processi aggiuntivi.

Esercizio 3.3 (server TCP con pre-forking)

Usare le funzioni `Socket()`, `Bind()` e `Listen()` come di consueto. Effettuare un ciclo con 10 iterazioni in ognuna delle quali viene effettuata una `Fork()`. Nella porzione di codice del figlio, inserire la funzione `Accept()`, all'interno di un ciclo infinito, seguita dalla gestione dei comandi inviati dal server. Le `Accept()` di tutti i figli saranno inizialmente tutte bloccate. All'arrivo di una connessione, una di queste ritornera' il valore corrispondente alla connessione, e le `Accept()` negli altri figli continueranno ad essere bloccate. Il kernel del sistema operativo selezionera' automaticamente una delle `Accept()` bloccate ogni volta che ci sara' una nuova connessione da parte di un client.

Per terminare i figli in caso di terminazione del padre, catturare il segnale `SIGINT` e nell'handler di gestione del segnale, effettuare una `kill()` per ogni figlio, e quindi una serie di `wait()`.