LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 3

Server TCP con I/O Multiplexing

Antonio Mastrolembo Ventura

543644

March 7, 2025

1 Esercizio1

Dopo aver visto, nell'esercizio precedente, due approcci diversi per la creazione di un server, ed avendo concluso che entrambi i metodi non sono ottimali, vediamo adesso un meccanismo più efficace che sposta la logica generale a livello del kernel del sistema operativo, ovvero la syscall `select()`. Questa soluzione ci permette di monitorare multipli file descriptor (sempre a singolo thread), evitare i cicli di attesa attiva e di ottimizzare l'uso delle risorse, in particolare la CPU. Dunque quanto utilizziamo il termine "I/O Multiplexing" ci riferiamo alla capacità di un singolo processo di gestire più operazioni di input/output contemporaneamente su diversi file descriptor, senza doverli gestire uno per volta in modo bloccante.

1.1 Codice del Server con I/O Multiplexing

1.1.1 Server Socket non bloccante

```
1    // Imposto il socket non bloccante
2    set_nonblock(fdsocket);
3    printf("Server con select() in ascolto sulla porta %d\n", PORT);
```

La prima cosa da fare è impostare il socket del server in modalità non bloccante, questo perchè di default i socket in Linux sono bloccanti e, nel nostro caso, utilizziamo la syscall `select()` permettendo di aspettare eventi su più socket contemporaneamente senza bloccare il server.

1.1.2 File descriptor da monitorare

```
1    // Insieme dei file descriptor da monitorare
2    fd_set readfds;
3    int maxfd = fdsocket;
```

- `readfds` è una struttura usata da `select()` per monitorare i file descriptor pronti alla lettura;
- `maxfd` ci servirà per tracciare il file descriptor con il valore più alto (vedremo dopo nella select).

1.1.3 Loop principale e inizializzazione del set

```
1    while (1) {
2        FD_ZERO(&readfds);
3        FD_SET(fdsocket, &readfds);
```

All'interno del loop principale abbiamo le prime due macro:

- `FD_ZERO` inizializza il set resettandolo ad ogni iterazione perché rimuoviamo eventuali file descriptor appartenenti a client non più connessi.
- `FD_SET` mi permette di aggiungere file descriptor al set. In questo caso aggiungo il fd del socket del server perchè dovrà monitorare eventuali nuove connessioni da parte dei client.

1.1.4 Aggiunta dei client al set

```
1        for (int i = 0; i < MAX_CLIENTS; i++) {
2            if (fdclients[i] > 0) {
3                FD_SET(fdclients[i], &readfds);
4                if (fdclients[i] > maxfd) {
5                    maxfd = fdclients[i];
6                }
7            }
8        }
```

Utilizzo un array di appoggio `fdclients[MAX_CLIENTS]` per tenere traccia dei client effettivamente connessi. Questo array serve per aggiungere i client connessi al set ad ogni iterazione. Se nell'array alla posizione `i-esima` è presente un fd (< 0) allora viene aggiunto al set. Inoltre se questo fd ha un numero più grande del massimo attuale viene impostato quest'ultimo come tale.

1.1.5 Definizione del timeout

```
1        struct timeval timeout;
2        timeout.tv_sec = 5;
3        timeout.tv_usec = 0;
```

`struct timeval` è una struttura che contiene due campi:

- `tv_sec` tempo espresso in secondi e `tv_usec` tempo espresso in microsecondi.

Nel mio caso ho impostato un timeout di 5 secondi, ma cosa vuol dire? In poche parole questo timeout permette alla `select()` di aspettare massimo 5 secondi per vedere se ci sono eventi sui socket monitorati. Se in quei 5 secondi non succede nulla `select()` restituisce il controllo al programma permettendo al server di fare altro o banalmente di ripetere questa attesa. Uno strumento del genere ci permette di risolvere il problema del server bloccante all'infinito aggiungendo effettivamente un tempo limite per cui esso può rimanere in attesa.

1.1.6 Chiamata a select()

```
1         // Monitoro i file descriptor
2         if (select(maxfd + 1, &readfds, NULL, NULL, &timeout) < 0) {
3             perror("Select failed!");
4             exit(EXIT_FAILURE);
5         }
```

Adesso possiamo invocare la `select()` per monitorare i file descriptor. Questa chiamata prende come argomenti diversi parametri:

- `maxfd + 1` è il valore del fd con il valore più alto (quello settato precedentemente) aumentato di uno;
- `&readfds` è il set di file descriptor da monitorare per la lettura;
- `NULL` e `NULL` sono altri due set di file descriptor (`writefds` e `exceptfds`) in questo caso impostati su `NULL` perchè ci interessa soltanto la lettura;
- `&timeout` è il timeout impostato prima già descritto.

La `select()` ritorna tre possibili valori:

- < 0: ritorna il numero di fd per cui si è verificato un evento;
- = 0: timeout scaduto, dunque nessun fd è pronto;
- = -1: errore.

Inoltre la `select()` fa un'altra cosa importante: modifica i set di file descriptor che le vengono passati come argomenti (in questo caso soltanto `readfds`) lasciando i file descriptor che sono pronti per essere letti. Questo è il motivo per cui ad ogni iterazione resetto il set e lo riempio nuovamente con i client connessi, altrimenti mi potrei ritrovare con dei client effettivamente ancora connessi ma che non sono più presenti nel set.

1.1.7 Controllo nuove connessioni

```
1        // Controllo se ci sono nuove connessioni
2        if (FD_ISSET(fdsocket, &readfds)) {
3            if ((newsocket = accept(fdsocket, (struct sockaddr *)&
   client_addr, &clilen)) < 0) {
4                perror("Accept failed!");
5                exit(EXIT_FAILURE);
6            }
7
8            // Imposto il nuovo socket non bloccante
9            set_nonblock(newsocket);
10
11           printf("Client %d connesso\n", newsocket);
12
13           // Aggiungo il nuovo socket all'array dei client
14           for (int i = 0; i < MAX_CLIENTS; i++) {
15               if (fdclients[i] == 0) {
16                   fdclients[i] = newsocket;
17                   break;
18               }
19           }
20       }
```

A questo punto controllo se ci sono nuove connessioni da parte di client. Utilizzo la macro `FD_ISSET()` per verificare se il socket del server `fdsocket` è pronto per accettare nuove connessioni, quindi in questo caso se il socket ha una connessione in ingresso che può essere accettata con `accept()`.
Superato questo controllo, viene chiamata la funzione `accept()` che restituisce il fd

del client appena connesso. Questo fd viene subito messo in modalità non bloccante (esattamente come all'inizio per il server) e viene infine aggiunto all'array di appoggio nella posizione libera che trova.

1.1.8 Leggo i dati dei client

```
        // Controllo i client connessi per vedere se ci sono dati da
    leggere
        for (int i = 0; i < MAX_CLIENTS; i++) {
            if (FD_ISSET(fdclients[i], &readfds)) {
                int n = read(fdclients[i], buffer, BUFFER_SIZE);
                if (n == 0) {
                    printf("Client %d disconnesso\n", fdclients[i]);
                    close(fdclients[i]);
                    fdclients[i] = 0;
                } else {
                    buffer[n] = '\0';
                    send(fdclients[i], "ACK\n", 4, 0);
                    printf("Client %d: %s", fdclients[i], buffer);
                }
            }
        }
```

Questo ciclo itera su tutti i client connessi nell'array `fdclients`. La `select()`, come detto prima, ha già aggiornato il set `readfds` e la `FD_ISSET()` verifica semplicemente se quel determinato file descriptor del client è pronto per essere letto. A questo punto con la chiamata a `read()` leggiamo i dati inviati dal client e rispondiamo con un "ACK" come per avvisare quell'utente che il messaggio è stato ricevuto con successo.

1.2 Output ottenuto

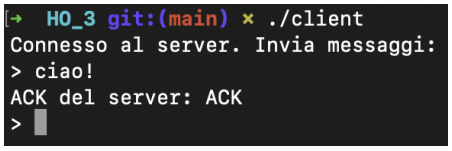

Figure 1: Client 4

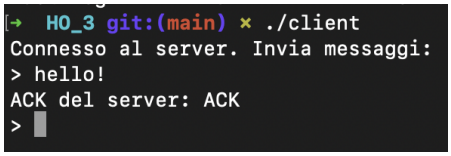

Figure 2: Client 5

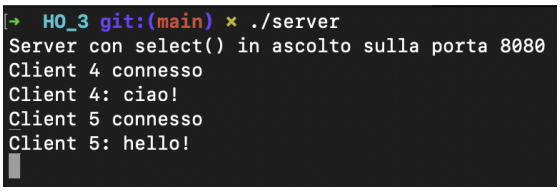

Figure 3: Server

1.3 Conclusioni

Per concludere, l'approccio con `select()` rappresenta un'ottima soluzione per gestire più client simultaneamente e allo stesso tempo, se il timeout è impostato in modo adeguato, consente al server di bilanciare efficacemente la gestione delle connessioni senza bloccarsi in attesa. Il costo computazionale dipende principalmente dal numero di file descriptor che il server deve monitorare, per cui avremo una complessità di tipo `O(n)`. Vedremo successivamente una tecnica che permette addirittura di raggiungere una complessità `O(1)`.