

LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 2

Server TCP Bloccante / Non-Bloccante

Antonio Mastrolembro Ventura

543644

March 5, 2025

1 Esercizio1

Per il seguente esercizio sono stati creati tre file differenti: *client_tcp.c*, *server_tcp_block.c* e *server_tcp_nonblock.c*. L'idea è quella di mettere a confronto e di vedere che differenze ci sono fra un server di tipo bloccante e un server di tipo non bloccante.

- **Server Bloccante:** utilizza **chiamate bloccanti**, ovvero operazioni che fermano l'esecuzione del programma fino a quando non vengono completate. In sostanza l'esecuzione del processo rimane in attesa senza poter fare altro.
- **Server Non-Bloccante:** utilizza chiamate **non-bloccanti** quindi il programma continua l'esecuzione senza fermarsi. Si verifica quella che è la cosiddetta **Attesa Attiva**: una tecnica per cui il programma continua a controllare ripetutamente una condizione fino a quando non viene soddisfatta.

Il client sarà sempre lo stesso, la cosa che cambierà sarà l'approccio utilizzato per il server.

1.1 Codice del Server Bloccante

```
1  while (1) {
2      // Accetto una connessione in modo bloccante
3      if ((fdclient = accept(fdsocket, (struct sockaddr *)&
4      client_addr, &clilen)) < 0) {
5          perror("Accept failed!");
6          continue;
7      }
8      printf("Client connesso\n");
9
10     // Inizializzo la sequenza di Fibonacci per ogni nuova
11     connessione
12     int fib1 = 0, fib2 = 1;
13     int first = 1;
14     int second = 1;
15
16     // Leggo i dati che mi arrivano dal client
```

```

16     while (1) {
17         int n = read(fdclient, buffer, BUFF_SIZE - 1);
18         if (n <= 0) {
19             printf("Client disconnesso\n");
20             break;
21         }
22
23         buffer[n] = '\0';
24         int client_number = atoi(buffer);
25
26         // Ottengo il prossimo numero della sequenza di Fibonacci
27         int next = next_fibonacci(&fib1, &fib2, &first, &second);
28
29         printf("Ricevuto: %d, Aspettato: %d\n", client_number,
30 next);
31
32         if (client_number == next) {
33             write(fdclient, "OK\n", 3);
34         } else {
35             printf("Numero errato, chiusura connessione.\n");
36             close(fdclient);
37             break;
38         }
39     }

```

In questo caso la funzione `accept()` è di tipo bloccante, quindi il programma si ferma fin quando un client non si connette. Una volta che il client si connette, il server entra nel secondo ciclo `while` in cui riceve i numeri da parte del client. Se il client invia un numero errato (in base alla sequenza di Fibonacci), la connessione viene chiusa e il server torna in attesa di un nuovo client.

1.2 Codice del Server Non-Bloccante

```
1 // Funzione per settare il socket in modalita' non bloccante
2 void set_nonblock(int fdsocket) {
3     int flags = fcntl(fdsocket, F_GETFL, 0);
4     if (flags == -1) {
5         perror("Fcntl failed!");
6         exit(EXIT_FAILURE);
7     }
8
9     if (fcntl(fdsocket, F_SETFL, flags | O_NONBLOCK) == -1) {
10        perror("Fcntl failed!");
11        exit(EXIT_FAILURE);
12    }
13 }
```

Questa funzione imposta un socket in modalità non bloccante, in modo che le chiamate di sistema come `accept()`, `recv()`, e `send()` non blocchino il programma se non ci sono dati disponibili o connessioni in arrivo.

- `fcntl()` è una chiamata di sistema che permette di modificare le proprietà di un file descriptor (file control);
- Con il flag `O_NONBLOCK` rendo il socket **non bloccante**.

Vediamo adesso come implementarlo correttamente nel main:

```
1 // Setto il socket in modalita' non bloccante
2 set_nonblock(fdsocket);
3 printf("Server non bloccante in ascolto sulla porta: %d\n", PORT)
4 ;
5 while (1) {
6     // Accetto la connessione
7     if ((fdclient = accept(fdsocket, (struct sockaddr *)&
8         client_addr, &clilen)) < 0) {
9         if (errno == EWOULDBLOCK) {
10            // Nessuna connessione in arrivo, continuo il ciclo
11            continue;
12        }
13    }
14 }
```

```

11         } else {
12             perror("Accept failed!");
13             exit(EXIT_FAILURE);
14         }
15     }
16
17     printf("Client connesso!\n");
18
19     // Setto il socket del client in modalita' non bloccante
20     set_nonblock(fdclient);
21
22     // Inizializzo la sequenza di Fibonacci per ogni nuova
23     connessione
24     int fib1 = 0, fib2 = 1;
25     int first = 1;
26     int second = 1;
27
28     // Leggo i dati inviati dal client con attesa attiva
29     while(1) {
30         int n = recv(fdclient, buffer, BUFFER_SIZE - 1, 0);
31         if (n > 0){
32             buffer[n] = '\0';
33
34             int client_number = atoi(buffer);
35             int next = next_fibonacci(&fib1, &fib2, &first, &
second);
36
37             printf("Ricevuto: %d, Aspettato: %d\n", client_number
, next);
38
39             if (client_number == next) {
40                 write(fdclient, "OK\n", 3);
41             } else {
42                 printf("Numero errato, chiusura connessione.\n");
43                 close(fdclient);
44                 break;
45             }
46         }
47     }

```

```

45         } else if (n == 0) {
46             printf("Client disconnesso\n");
47             break;
48         } else {
49             if (errno == EWOULDBLOCK) {
50                 // Nessun dato in arrivo, continuo il ciclo
51                 continue;
52             } else {
53                 perror("Recv failed!");
54                 exit(EXIT_FAILURE);
55             }
56         }
57     }
58 }

```

Stavolta la funzione `accept()` invece di restare in attesa se non ci sono connessioni in arrivo, restituisce `-1` con `ERRNO == EWOULDBLOCK` quindi continua il ciclo. Una volta che una connessione è accettata, il server imposta anche il socket del client (`fdclient`) in modalità non bloccante, in modo da evitare che il server venga bloccato in attesa di dati dal client.

1.3 Codice del Client

```

1     printf("Connesso al server. Invia numeri della sequenza di
    Fibonacci\n");
2
3     while (1) {
4         printf("> ");
5         fgets(buffer, BUFFER_SIZE, stdin);
6
7         send(sockfd, buffer, strlen(buffer), 0);
8
9         // Riceve risposta dal server
10        int n = read(sockfd, buffer, BUFFER_SIZE - 1);
11        if (n <= 0) {
12            printf("Connessione chiusa dal server.\n");
13            break;

```

```

14     }
15     buffer[n] = '\0';
16     printf("ACK del server: %s", buffer);
17 }

```

1.4 Output ottenuto

```

(kali㉿kali)-[~/Desktop/LabReSiD/H0_2]
$ ./client
Connesso al server. Invia numeri della sequenza di Fibonacci
> 0
ACK del server: OK
> 1
ACK del server: OK
> 1
ACK del server: OK
> 2
ACK del server: OK
> 3
ACK del server: OK
> 5
ACK del server: OK
> 5
Connessione chiusa dal server.

```

Figure 1: Output del client

```

(kali㉿kali)-[~/Desktop/LabReSiD/H0_2]
$ ./server_block
Server bloccante in ascolto sulla porta: 7500
Client connesso
Ricevuto: 0, Aspettato: 0
Ricevuto: 1, Aspettato: 1
Ricevuto: 1, Aspettato: 1
Ricevuto: 2, Aspettato: 2
Ricevuto: 3, Aspettato: 3
Ricevuto: 5, Aspettato: 5
Ricevuto: 5, Aspettato: 8
Numero errato, chiusura connessione.

```

Figure 2: Output del server

1.5 Verifica dei Server

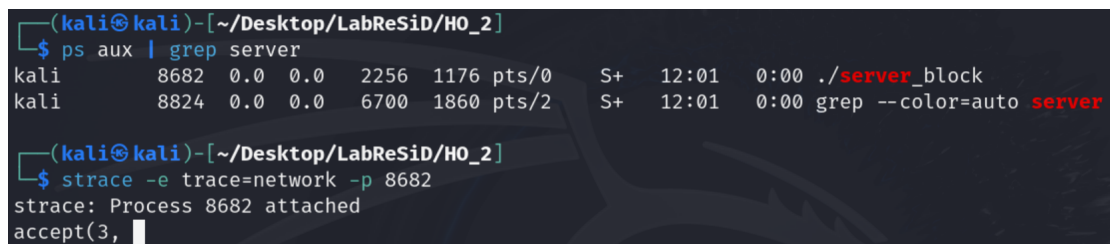
A questo punto possiamo verificare il comportamento del server bloccante e del server non bloccante, come? Con il comando `strace`. Questo è uno strumento che traccia le chiamate di sistema (system calls) e i segnali generati da un processo in esecuzione. In particolare con il comando completo: `strace -e trace=network -p PID`:

- `-e trace=network`: questa opzione specifica che `strace` deve tracciare solo le chiamate di sistema relative alla rete. Queste chiamate di sistema possono includere operazioni come l'apertura di socket, invio di dati tramite `send()`, ecc.
- `-p PID`: indica che `strace` deve monitorare un processo con il PID inserito.

A questo punto è possibile ottenere il PID del server tramite il comando `ps aux | grep server`. Con `ps aux` ottengo una lista di tutti i processi in esecuzione sul sistema e con `grep server` filtro i processi che contengono la parola "server".

1.5.1 Verifica Server Bloccante

Per quanto riguarda il server di tipo bloccante avremo un output di questo tipo:



```
(kali㉿kali)-[~/Desktop/LabReSiD/H0_2]
$ ps aux | grep server
kali      8682  0.0  0.0   2256   1176 pts/0    S+   12:01   0:00  ./server_block
kali      8824  0.0  0.0   6700   1860 pts/2    S+   12:01   0:00  grep --color=auto server

(kali㉿kali)-[~/Desktop/LabReSiD/H0_2]
$ strace -e trace=network -p 8682
strace: Process 8682 attached
accept(3, [
```

Figure 3: Server Bloccante

Come possiamo vedere la chiamata `accept()` resta in attesa di un client che si connetta.

1.5.2 Verifica Server Non-Bloccante

L'output del Server Non-Bloccante sarà invece:

```
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
accept(3, 0xfffff5f48ab8, [16]) = -1 EAGAIN (Resource temporarily unavailable)
```

Figure 4: Server Non-Bloccante

```
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
recvfrom(4, 0xfffff5f48a30, 127, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
```

Figure 5: Server Non-Bloccante

Come possiamo vedere abbiamo **attesa attiva**, dunque un polling continuo.

1.6 Conclusioni

Per concludere, dopo aver visto il comportamento di entrambi gli approcci, possiamo affermare che i due metodi presentano comunque delle soluzioni non ottimali. Con un server bloccante la CPU non lavora inutilmente, ma il server non può fare altro nel frattempo. Con un server non-bloccante invece la CPU continua a lavorare, ma il server

è reattivo e può gestire altre operazioni senza bloccarsi; il problema è che l'approccio non bloccante può portare a un aumento del consumo della CPU, poiché il server deve costantemente verificare la disponibilità di eventi da gestire.