

LABORATORIO DI RETI E SISTEMI DISTRIBUITI

HANDS-ON 4

Server TCP su Linux con epoll()

Antonio Mastrolemba Ventura

543644

March 11, 2025

Indice

1	Introduzione	2
2	Definizione del problema	2
3	Metodologia	3
3.1	Server Socket non bloccante	3
3.2	Istanza di epoll	3
3.3	Specifica degli eventi monitorati e aggiunta di fd al set	4
3.4	Ciclo principale e attesa di eventi	4
3.5	Gestione eventi - Richiesta di connessione	5
3.6	Gestione eventi - Lettura	6
4	Presentazione dei risultati	7
5	Conclusioni	8

1 Introduzione

Nei sistemi di creazione ed implementazione dei server è importante scegliere il giusto approccio da utilizzare per trarne beneficio in termini di efficienza. Un server deve essere in grado di gestire molteplici connessioni simultaneamente, riducendo al minimo il tempo di attesa e l'uso delle risorse di sistema. Nell'esercizio precedente abbiamo analizzato il meccanismo della chiamata di sistema `select()` ed i vantaggi che ci offre rispetto ad un approccio di tipo bloccante o non bloccante. Tuttavia, `select()` presenta delle limitazioni che ne riducono l'efficacia quando il numero di connessioni aumenta significativamente. Per affrontare queste problematiche, i sistemi Linux offrono una soluzione più efficiente: `epoll()`.

2 Definizione del problema

Quando un server deve gestire un numero crescente di connessioni, l'approccio visto basato su `select()` diventa sempre meno efficiente. Il motivo è legato al fatto che `select()` controlla tutti i file descriptor monitorati a ogni chiamata con complessità computazionale di $O(n)$, con n che rappresenta il numero di file descriptor attivi. Questo significa che il carico di lavoro del server aumenta all'aumentare dei client connessi. Inoltre, `select()` ha un limite nel numero massimo di file descriptor gestibili, ovvero 1024 tipicamente.

Per affrontare questi problemi, Linux mette a disposizione un meccanismo che consente di monitorare in modo più efficiente eventi di I/O su un gran numero di file descriptor: `epoll()`. Questo meccanismo utilizza delle strutture dati nel kernel molto più efficienti (un albero e una linked list) che permettono di ridurre la complessità computazionale a $O(1)$.

3 Metodologia

Lo sviluppo del problema prevede l'implementazione del metodo `epoll()` nel linguaggio C attraverso la creazione di un server che comunicherà con dei client.

3.1 Server Socket non bloccante

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

Come per `select()`, anche in questo caso è fondamentale impostare il socket del server in modalità non bloccante. Di default i socket in Linux sono bloccanti; impostando il socket in modalità non bloccante, le chiamate di lettura e scrittura non si bloccheranno mai, ma restituiranno immediatamente con un errore se non è possibile completare l'operazione.

3.2 Istanza di epoll

```
1 // Creo l'istanza di epoll
2 int epollfd = epoll_create1(0);
3 if (epollfd == -1) {
4     perror("Epoll creation failed!");
5     exit(EXIT_FAILURE);
6 }
```

A questo punto introduciamo la prima funzione fondamentale che ci permette di creare un'istanza epoll: `epoll_create1()`. Questa chiamata di funzione ci restituisce un file descriptor che rappresenta l'istanza di epoll e che ci permetterà di monitorare più file descriptor per eventi di I/O. Se la creazione dell'istanza di epoll non riesce, il programma stampa un messaggio di errore e termina.

3.3 Specifica degli eventi monitorati e aggiunta di fd al set

```
1 // Specifico il tipo di evento da monitorare
2 event.events = EPOLLIN;
3 event.data.fd = fdsocket;
4
5 // Aggiungo il socket del server all'istanza di epoll
6 if (epoll_ctl(epollfd, EPOLL_CTL_ADD, fdsocket, &event) == -1) {
7     perror("Epoll control failed!");
8     exit(EXIT_FAILURE);
9 }
```

Con le prime due assegnazioni stiamo intervenendo su una struct, in particolare la struct `epoll_event event`. Stiamo dicendo che vogliamo monitorare eventi di lettura (`EPOLLIN`) e associamo all'evento il file descriptor del socket del server (`fdsocket`) per monitorare nuove connessioni in arrivo.

Successivamente introduciamo la seconda funzione fondamentale che ci permette di aggiungere all'istanza di `epoll` che abbiamo creato precedentemente, i file descriptor dei socket che ci interessa monitorare. `epoll_ctl()` è la funzione che si occupa di questo compito, in questo caso in particolare stiamo aggiungendo il socket del server per monitorare le connessioni in arrivo. Con `EPOLL_CTL_ADD` indichiamo che vogliamo aggiungere un nuovo file descriptor all'istanza di `epoll`.

3.4 Ciclo principale e attesa di eventi

```
1 // Ciclo principale
2 while (1) {
3     // Aspetto che si verifichi un evento I/O su uno dei socket
4     int num_events = epoll_wait(epollfd, events, MAX_EVENTS, -1);
5     if (num_events == -1) {
6         perror("Epoll wait failed!");
7         exit(EXIT_FAILURE);
8     }
9 }
```

Questo è il ciclo principale del programma che gestisce la parte di attesa degli eventi di I/O su uno o più socket monitorati. La funzione è `epoll_wait()` il cui valore di ritorno sarà il numero di file descriptor pronti per essere elaborati.

3.5 Gestione eventi - Richiesta di connessione

```
1      // Ciclo sugli eventi I/O ritoranti da epoll_wait
2      for (int i = 0; i < num_events; i++) {
3          // Se l'evento e' relativo al socket del server allora e'
4          // una richiesta di connessione
5          if (events[i].data.fd == fdsocket) {
6              newsocket = accept(fdsocket, (struct sockaddr *)&
7              client_addr, &clilen);
8              if (newsocket == -1) {
9                  perror("Accept failed!");
10                 exit(EXIT_FAILURE);
11             }
12
13             // Imposto il socket non bloccante
14             set_nonblock(newsocket);
15
16             printf("Client %d connesso\n", newsocket);
17
18             // Specifico il tipo di evento da monitorare
19             event.events = EPOLLIN | EPOLLET;
20             event.data.fd = newsocket;
21
22             // Aggiungo il socket del client all'istanza di epoll
23             if (epoll_ctl(epollfd, EPOLL_CTL_ADD, newsocket, &
24             event) == -1) {
25                 perror("Epoll control failed!");
26                 exit(EXIT_FAILURE);
27             }
28         }
```

Una volta che `epoll_wait()` restituisce il numero di file descriptor pronti, il programma entra in un ciclo per gestire ogni evento. Il primo "caso" da gestire

riguarda un evento di richiesta connessione da parte di un client effettuato sul socket del server. In particolare, se la condizione del primo `if` è verificata con successo allora un nuovo client sta cercando di connettersi.

A questo punto controlliamo se `accept()` ha avuto successo e, in tal caso, settiamo il socket del client appena connesso come non bloccante. Successivamente configuriamo il nuovo socket per `epoll`, in particolare (come visto precedentemente) impostiamo la struct event dicendo che vogliamo monitorare eventi di lettura (`EPOLLIN`) e abilitiamo la modalità `edge-triggered` (`EPOLLET`), che riduce il numero di chiamate a `epoll_wait()`, notificando solo quando ci sono nuove operazioni disponibili (la modalità di default è `level-triggered` che invece notifica l'evento finché i dati non vengono completamente letti).

Il prossimo passo è quello di aggiungere il socket del client all'istanza di `epoll` tramite la chiamata a `epoll_ctl()`, in modo che possa essere monitorato per eventi futuri. Se questa operazione fallisce, viene stampato un messaggio di errore e il programma termina.

3.6 Gestione eventi - Lettura

```
1         } else {
2             // Se l'evento e' associato ad un socket di un client
3             // allora e' una richiesta di lettura
4             int fd = events[i].data.fd;
5             int nread = read(fd, buffer, BUFFER_SIZE);
6             if (nread == -1) {
7                 perror("Read failed!");
8                 exit(EXIT_FAILURE);
9             } else if (nread == 0) {
10                // Il client ha chiuso la connessione
11                printf("Client %d disconnesso\n", fd);
12                close(fd);
13            } else {
```

```

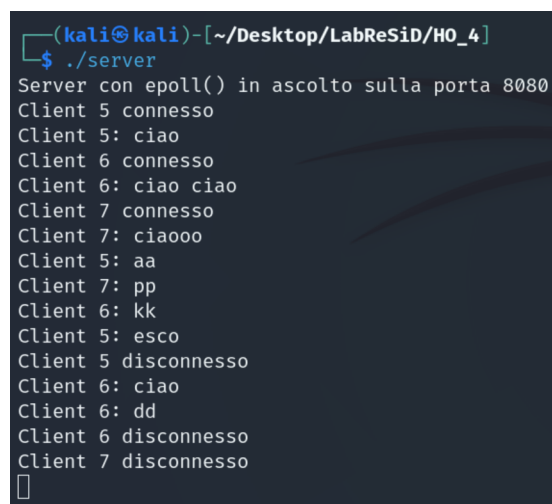
13         // Stampo il messaggio ricevuto e invio un "ACK"
    al client

14         buffer[nread] = '\0';
15         send(fd, "ACK\n", 4, 0);
16         printf("Client %d: %s", fd, buffer);
17     }
18 }

```

Il secondo "caso" da gestire riguarda un evento di lettura su un socket di un client già connesso. Se il file descriptor associato all'evento non corrisponde al socket del server, significa che un client ha inviato dei dati. A questo punto il programma, con la chiamata `read()` tenta di leggere dei dati dal socket. Se il risultato della `read` sarà 0 allora il client avrà chiuso la connessione, mentre se avrà un valore positivo viene letto il contenuto del buffer e mandato al client un messaggio "ACK" per segnalare che la lettura è andata a buon fine.

4 Presentazione dei risultati



```

(kali@kali) - [~/Desktop/LabReSiD/H0_4]
$ ./server
Server con epoll() in ascolto sulla porta 8080
Client 5 connesso
Client 5: ciao
Client 6 connesso
Client 6: ciao ciao
Client 7 connesso
Client 7: ciao
Client 5: aa
Client 7: pp
Client 6: kk
Client 5: esco
Client 5 disconnesso
Client 6: ciao
Client 6: dd
Client 6 disconnesso
Client 7 disconnesso

```

Figure 1: Output del Server

Durante il test, sono stati avviati diversi client che inviavano messaggi al server; esso ha gestito correttamente ogni richiesta stampando i messaggi ricevuti e inviando una conferma ("ACK") in risposta, visibile al client.

5 Conclusioni

In questo esercizio abbiamo analizzato l'uso di `epoll` per la gestione efficiente di connessioni multiple in un server TCP. A differenza dell'uso di `select`, che diventa inefficiente all'aumentare del numero di file descriptor monitorati, `epoll` offre una soluzione migliore grazie alla sua gestione basata sugli eventi e alle strutture dati ottimizzate nel kernel Linux. La complessità della gestione degli eventi si riduce a $O(1)$ rispetto alla $O(n)$ di `select`.

In conclusione, `epoll` rappresenta una soluzione altamente efficiente per la gestione di server ad alte prestazioni; risulta infatti molto utile in scenari in cui è necessario gestire molte connessioni contemporaneamente.