I/O su Socket TCP: read()

I socket TCP, una volta che la connessione TCP sia stata instaurata, sono accedibili come se fossero dei file, mediante un descrittore di file (un intero) ottenuto tramite una socket() o una accept() o una connect(). Con questo descrittore è possibile effettuare letture tramite la funzione read, che restituisce i byte letti dal flusso in entrata, e scritture tramite le funzioni write e send, che spediscono i byte formando il flusso in uscita.

ssize_t **read** (int fd, void *buf, size_t count);

cerca di leggere count byte dal file descriptor fd, scrivendoli nel buffer puntato da buf. Se count è zero la read restituisce zero.
Se count è maggiore di zero viene effettuata la lettura e viene restituito il numero di byte letti (maggiore di zero, se tutto è OK).
- Se viene restituito **0** (**zero**) significa end-of-file (fine stream), ovvero significa che l'altro end system ha volutamente chiuso la connessione e quindi il socket non potrà essere più utilizzato per leggere.
- Se viene restituito **-1** è accaduto un errore e viene settata la variabile globale **errno** definita in <errno.h> con un valore che indica quale errore è avvenuto. In particolare, se **errno** vale **EINTR** significa che il sistema operativo ha dovuto interrompere la system call read, ma il socket non è in stato di errore, quindi la read può essere ripetuta immediatamente con gli stessi parametri. Altro caso particolare è quando il socket è stato definito non bloccante nel qual caso se non ci sono byte disponibili viene restituito EAGAIN, ed il socket rimane utilizzabile. Se invece errno ha un altro valore (EBADF, EINVAL, EFAULT, …) il socket viene invalidato.

La funzione read, applicata ad un socket, presenta una particolarità. Può accadere che la read() restituisca meno byte di quanti richiesti, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la read (richiedendo il numero dei byte mancanti) fino ad ottenerli tutti. Os ssize_t è definito in <unistd.h> ed è un long.

I/O su Socket TCP: write() (1)

ssize_t **write** (int fd, const void *buf, size_t count);

cerca di scrivere fino a count byte nel buffer di sistema corrispondente al file descriptor fd perché siano poi trasmessi. I byte vengono letti dal buffer puntato da buf.
- Se count è zero la write restituisce zero e non scrive nulla.
- Se count è maggiore di zero viene effettuata la scrittura e viene restituito il numero di byte scritti.
- Se viene restituito -1 è accaduto un errore e viene settata la variabile errno con un valore che indica l'errore.

- Se il socket è configurato in modalità **bloccante** (è il default) **la write è bloccante**, cioè attende fino a che tutti i byte sono stati passati al buffer di sistema. Ciò significa che se non c'è abbastanza spazio nel buffer di sistema la write attende fino a che non sono stati spediti abbastanza byte da permettere la scrittura di tutti i byte passati. Durante questa attesa può capitare che al processo arrivi un segnale (es SIGUSR1) che deve essere gestito dal processo stesso; **in tal caso la write termina restituendo il numero di byte scritti sul buffer di sistema**. Se invece nessun byte è stato ancora scritto viene restituito -1 e indicato l'errore EINTR.
- Se invece il socket è configurato in modalità non bloccante la write scrive sul buffer di sistema il maggior numero di byte possibile senza produrre attesa e poi termina restituendo il numero di byte scritti. Se non è stato possibile scrivere nulla la write restituisce -1 senza attendere e setta errno al valore EAGAIN.

Altri possibili errori sono EBADF (file descriptor non valido), EINVAL (file descriptor non permette scritture), EFAULT (il buffer buf è fuori dallo spazio di memoria permesso), EPIPE (il socket è stato chiuso dall'altro end system).

I/O su Socket TCP: write() (2)

Indipendentemente da come i socket sono stati settati, **nel momento in cui viene invocata la write** può capitare che **il processo venga terminato** dall'arrivo di un segnale **SIGPIPE** che viene generato dalla write stessa per indicare che il socket che si sta usando è stato chiuso in modo anormale dall'altro end system (inviando un segmento col flag reset) o che non è più utilizzabile.

Per impedire la terminazione del processo ho due diverse possibilità:

1) istruire il processo per **far intercettare i segnali SIGPIPE**, nel qual caso **la write invece di inviare il segnale restituirà -1 indicando come errore EPIPE**.

```
// da eseguire solo una volta, in fase di setup del processo
if( signal(SIGPIPE, SIG_IGN) == SIG_ERR )
    {perror("signal SIGPIPE failed: ");exit(1);}
…..
// poi si possono fare tutte le chiamate alla write
ris=write(socketfd, buff, n);
if(ris<0){
     if(errno==EPIPE) { printf("chiusura anomala\n");exit(1); }
     ……
}
```

Così però intercetto tutti i segnale SIGPIPE indipendentemente da quale socket (usato dal processo) lo provoca.

2) utilizzare al posto della write la system call send, in cui **può essere specificato** di non generare il segnale SIGPIPE ma **di restituire -1 indicando come errore EPIPE**.
In questo modo, solo il segnale SIGPIPE di quella particolare invocazione viene intercettato.

I/O su Socket TCP: send() (3)

int **send** (int fd, const void *buf, size_t count, int flags);

cerca di scrivere fino a count byte nel buffer di sistema corrispondente al file descriptor fd perché siano poi trasmessi.
I byte vengono letti dal buffer puntato da buf.

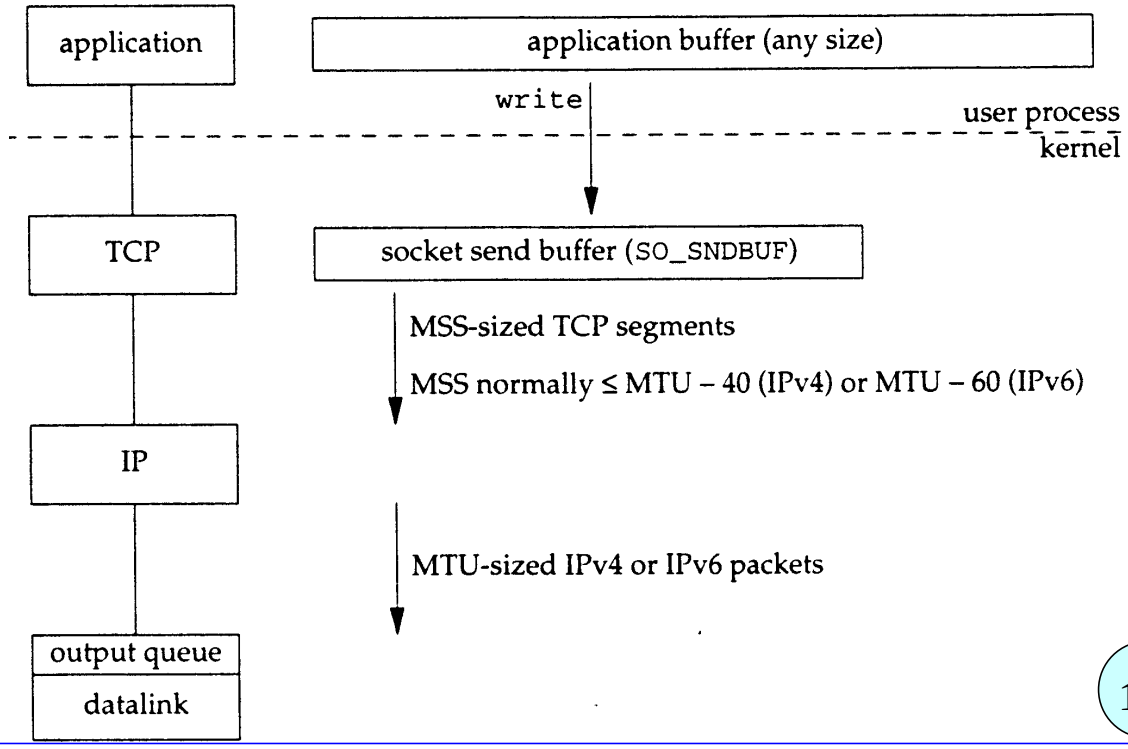
- Se count è zero la send restituisce zero e non scrive nulla.
- Se count è maggiore di zero viene effettuata la scrittura e viene restituito il numero di byte scritti.
- Se viene restituito -1 è accaduto un errore e viene settata la variabile errno con un valore che indica l'errore.

Il comportamento viene influenzato dal valore del parametro flags, il cui valore può essere 0 oppure viene assegnato mediante OR bit a bit delle seguenti costanti: MSG_OOB, MSG_DONTWAIT, MSG_NOSIGNAL.
Se il valore di flags è zero la send si comporta come la write.

Se viene specificato MSG_DONTWAIT la send non si blocca bensì scrive il numero di byte possibili nel buffer di sistema e termina restituendo il numero di byte scritti, eventualmente zero.
Se viene specificato MSG_NOSIGNAL la send non solleva l'eccezione di tipo SIGPIPE e quindi non rischia di far terminare il processo. Al contrario, in caso di chiusura anormale della connessione, restituisce -1 e setta la variabile errno al valore EPIPE.

I/O su Socket TCP: TCP Output (2)

Ogni socket TCP possiede un buffer per l'output (send buffer) in cui vengono collocati temporaneamente i dati che dovranno essere trasmessi mediante la connessione instaurata. La dimensione di questo buffer può essere configurata mediante un'opzione SO_SNDBUF.
Quando un'applicazione chiama write() per n bite sul socket TCP, il kernel cerca di copiare n byte dal buffer dell'appl. al buffer del socket. Se il buffer del socket e' più piccolo di n byte, oppure è già parzialmente occupato da dati non ancora trasmessi e non c'è spazio sufficiente, verranno copiati solo nc<n byte, e verrà restituito dalla write il numero nc di byte copiati.
Se il socket ha le impostazioni di default, cioè è di tipo bloccante, la fine della routine write ci dice che sono stati scritti sul buffer del socket quegli nc byte, e possiamo quindi riutilizzare le prime nc posizioni del buffer dell'applicazione. Ciò non significa affatto che già i dati siano stati trasmessi all'altro end-system.



I/O su Socket TCP : utility (1)

Per attendere di ricevere almeno un byte, o leggere i byte già arrivati, si usa la già descritta

ssize_t **read** (int fd, void *buf, size_t **n**);

Per attendere di ricevere TUTTI i byte richiesti, si implementa la seguente funzione readn, che

- -restituisce -1 in caso di errore, e setta errno
- -restituisce il numero di byte letti se l'altro end system chiude la connessione
- - restituisce il numero di byte chieste (e letti) se tutto ok.

```
ssize_t  readn (int fd, char *buf, size_t n)
{
        size_t      nleft;     ssize_t  nread;

        nleft = n;
        while (nleft > 0) {
                if ( (nread = read(fd, buf+n-nleft, nleft)) < 0) {
                        if (errno != EINTR)
                                return(-1);   // restituisco errore
                }
                else if (nread == 0) {
                        // EOF, connessione chiusa, termino
                        // esce e restituisco il numero di byte letti
                        break;
                }
                else  // continuo a leggere
                        nleft -= nread;
        }
        return(n - nleft);      // return >= 0
}
```

I/O su Socket TCP : utility (2)

Per attendere di consegnare al buffer di sistema TUTTI i byte richiesti
restituisce -1 in caso di errore e setta errno
restituisce il numero di byte da inviare ed inviati, se tutto OK.

```
ssize_t  writen (int fd, const char *buf, size_t n)
{
  size_t nleft;   ssize_t nwritten;   char *ptr;

  ptr = buf;  nleft = n;
  while (nleft > 0)
  {
      if ( (nwritten = send(fd, ptr, nleft,  MSG_NOSIGNAL  )) < 0) {
          if (errno == EINTR)   nwritten = 0;   /* and call write() again*/
          else                          return(-1);       /* error */
      }
      nleft -= nwritten;         ptr   += nwritten;
  }
  return(n);
}
```

Per consegnare da zero ad n byte da trasmettere, ma senza attendere:
restituisce -1 in caso di errore, se no restituisce il numero di byte scritti

```
ssize_t  write_nowait (int fd, const char *buf, size_t n)
{
  int  nwritten;

 do {
    nwritten=send ( fd, buf, n, MSG_DONTWAIT|MSG_NOSIGNAL);
  }while( (nwritten<0) && (errno==EINTR) );
  return(nwritten);
}
```

Interazioni tra Client e Server TCP

Per primo viene fatto partire il server, poi viene fatto partire il client che chiede la connessione al server e la connessione viene instaurata. Nell'esempio (ma non è obbligatorio) il client spedisce una richiesta al server, questo risponde trasmettendo alcuni dati. Questa trasmissione bidirezionale continua fino a che uno dei due (il client nell'esempio) decide di interrompere la connessione, e tramite la close() chiude la connessione. Infine il server chiude a sua volta la connessione.