

Sviluppo di software di rete con l'utilizzo di sistemi UNIX

*Utilizzo dei socket:
server concorrenti e nozioni
avanzate*

Claudio Vicari

Socket: comportamenti con read e write

- ➔ sui socket si legge e si scrive con read e write come avviene con gli altri file descriptor
- ➔ però, il comportamento di read e write è diverso dal solito... infatti, ci sono in gioco anche i buffer di rete, che potrebbero saturarsi o svuotarsi, dunque:
 - read potrebbe leggere meno byte di quelli richiesti se i buffer sono vuoti
 - write normalmente si blocca. Però, se è impostata per non essere bloccante, ritorna comunque, anche senza avere finito il suo compito
- ➔ queste situazioni non sono da trattare come errori... basta quindi riprendere e completare la scrittura da dove era rimasta

Possibili soluzioni: readn, writen

readn.c

writen.c

- ⇒ queste funzioni:
 - controllano se la lettura/scrittura ha usato tutti i byte richiesti
 - se non è così, riprendono la lettura/scrittura dopo aver aumentato l'offset nel buffer
 - continuano fino ad aver esaurito il numero di byte
- ⇒ attenzione a non leggere un solo byte alla volta! Ciò può portare ad un calo delle prestazioni

La funzione isfdtype

```
int isfdtype( int fd, int fdtype );
```

- ➔ questa funzione controlla se il file descriptor è del tipo specificato in fdtype
- ➔ nella pagina di man relativa a fstat (man 2 stat) troviamo elencate le costanti definite per fdtype
- ➔ in alternativa, possiamo usare la funzione fstat e alcune macro, sempre definite nella stessa pagina di manuale
- ➔ il tutto, è contenuto nell'header <sys/stat.h>
- ➔ Posix specifica il comportamento solamente per S_IFSOCK, anche se nella pratica le altre macro sono molto diffuse

Analizzare socket: getsockname, getpeername

```
int getsockname(int sfd, struct sockaddr *name,  
                socklen_t *namelen);  
int getpeername(int sfd, struct sockaddr *name,  
                socklen_t *namelen);
```

- ➡ queste funzioni scrivono i dati nelle strutture di tipo `sockaddr`
- ➡ `getsockname` restituisce i dati relativi all'estremo locale
- ➡ `getpeername` restituisce i dati relativi all'estremo remoto
- ➡ anche se si parla di `name`, ci si riferisce comunque all'indirizzo completo del socket
- ➡ la costante `MAXSOCKADDR` (in `unp.h`!) ci dice quanti byte bisogna allocare, nel caso peggiore

Esempio per getsockname

get_sp_name.c

- ➔ per getpeername, il funzionamento è quasi identico

Server: progettazione di server concorrenti

- ➡ i server che abbiamo visto sinora sono server *iterativi*, cioè gestiscono iterativamente le richieste che giungono dai client
- ➡ in casi reali, però, possono giungere molte richieste contemporaneamente, che vanno gestite senza provocare il blocco dell'esecuzione
- ➡ il metodo più semplice è utilizzare un singolo ciclo che faccia accept
- ➡ dopo ogni accept, il server farà una fork
 - il padre torna pronto per fare accept
 - il figlio invece utilizza il nuovo socket, e si dedica interamente a gestirlo
- ➡ ogni processo chiude i fd che non utilizza: se il server non lo facesse, esaurirebbe gli interi a disposizione

Esempio di server concorrente: echo

tcpserv01.c

- ➔ questo server si limita a ritrasmettere al client tutto quello che vi viene scritto.
Ascolta sulla porta 9877
- ➔ è un esempio di server concorrente
- ➔ possiamo metterlo alla prova con un programma qualsiasi, come netcat o telnet, piuttosto che con un client apposito

Problemi da gestire /1

- ➔ proviamo a vedere i processi a questo punto: ci sono molti processi zombie!
- ➔ quindi, anzitutto dobbiamo aggiungere la gestione di SIGCHLD
- ➔ e anche la gestione delle chiamate di sistema interrotte!
 - accept, read, write sono interrompibili

tcpserv02.c

sigchldwait.c

Problemi da gestire /2

- ➔ e cosa accade nei sistemi in cui i segnali non sono accodati? SIGCHLD non rimuove **tutti** i child, ma solo alcuni...
- ➔ soluzione: waitpid, con l'opzione WNOHANG
- ➔ il nuovo server è l'esempio tcperv04 sugli esempi di UNPv12e, che usa sigchldwaitpid.c

sigchldwaitpid.c

Problemi da gestire /3: SIGPIPE

- ➔ se un processo scrive in un socket che dall'altro estremo è chiuso, allora si genera un SIGPIPE
- ➔ possiamo gestire un SIGPIPE semplicemente: ignorandolo e intercettando l'errore sulla write
- ➔ notare che quando riceviamo il segnale, non sappiamo quale socket lo ha provocato: quindi, se abbiamo più socket aperti, non conosciamo il colpevole!
- ➔ esempio: questo client riceve un segnale di SIGPIPE a causa del comportamento del server daytime, che chiude subito la connessione

tsigpipe.c

Problemi da gestire /4:

passaggio di dati

- ➔ attenzione a cosa accade se inviamo dati binari! Per esempio, usando printf da un lato e scanf dall'altro lato...
- ➔ in questo caso, ricordiamo che:
 - i tipi di dato possono avere lunghezze differenti fra i due estremi: interi a 32 o 64 bit?
 - i sistemi possono essere big-endian o little-endian
 - le struct potrebbero essere allineate (lunghezze di 64 bit fisse...)
- ➔ soluzioni possibili:
 - inviare tutto come stringa
 - definire i tipi e fare le conversioni esplicitamente

`str_cli09.c`

`str_echo09.c`