Internetworking with TCP/IP (Douglas E. Comer) Vol. I and Vol III

http://gapil.truelite.it/

http://www.linuxdidattica.org/docs/altre_scuole/planck/socket/

Client-Server

Esistono vari modelli di architettura per le applicazioni di rete, i più importanti sono:

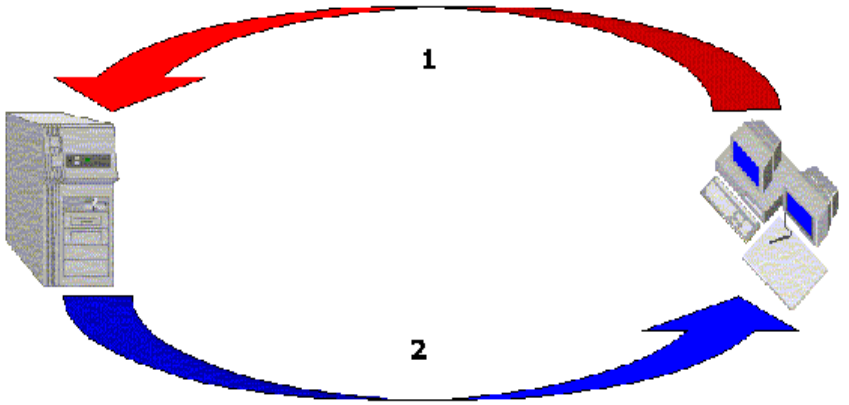
- cliente/ servente (*client/server*);
- paritetico (p2p, *peer to peer*);
- a tre livelli (*three-tier*). Architettura three-tier ("a tre strati"), e' un'architettura software che prevede la suddivisione del sistema in tre diversi moduli dedicati rispettivamente alla intefaccia utente, alla logica funzionale e alla gestione dei dati persistenti. Tali moduli sono intesi interagire fra loro secondo le linee generali del paradigma client-server

Il modello più importante in ambito GNU/Linux e, storicamente, in Unix e' pero' quello cliente/servente sul quale ci soffermeremo in modo particolare.

Client-Server

Il modello client-server e' cosi' denominato in base alla differenziazione concettuale che si effettua tra le due parti di una connessione di rete.

Il cosiddetto **lato client**, effettua la richiesta di esecuzione di un servizio. La sua controparte, il **lato server**, effettua l'esecuzione del sevizio richiesto.



Everything is a file

Ci sono in UNIX tre categorie di file:

1. I file veri e propri, chiaramente questa e' una categoria molto ampia comprendente ASCII files, files binari, eseguibili etc etc.

2. Le directories, files speciali che contengono una lista di files.

3. device files, sono i file mediante i quali e' possibile comunicare con i module e quindi con l'hardware.

Everything is a file

Le primitive di I/O Unix, derivanti da quelle presenti in Multics, seguono un paradigma talvolta denominato come **open-read-write-close**.

Prima che un processo possa eseguire opeazioni di I/O chiama una open sul file (o dispositivo) ed alla fine dell'interazione il processo chiama una close.

Le prime implementazione del TCP/IP sotto UNIX usavano il paradigma **open-read-write-close** mediante il device file /dev/tcp.

Successivamente ci si rese conto che i protocolli di rete sono piu' complessi dei normali dispositivi di I/O.

Socket

Le interfacce software o API (*Application program interface*) di comunicazione piu' utilizzate fra processi in rete, sono:

- socket di Berkeley, di Unix BSD;
- TLI (*Transport layer interface*) di Unix System V.

Fra le due la API basata sui socket e' senz'altro la più usabile e flessibile e quindi la piu' diffusa. Essa e' stata introdotta nel 1982 in Unix BSD 4.1c ed e' rimasta sostanzialmente invariata.

Socket

- I socket vengono usati per la comunicazione di rete in molti sistemi operativi, compresi Unix e GNU/Linux e possono essere utilizzati anche per far comunicare processi in esecuzione sulla stessa macchina (**vedi socketpair e differenza con le pipe**).

- La flessibilita' dei socket permette di usarli a fronte di stili di comunicazione diversi: ad esempio nel caso di invio dei dati come flusso di byte o suddivisi in pacchetti, di comunicazioni affidabili o non etc etc.

- Nel caso di comunicazione in rete i socket possono essere usati con protocolli diversi. Noi faremo riferimento unicamente all'uso dei socket con l'architettura TCP/IP (tratteremo quasi eslusivamente TCP).

Socket

La creazione di un socket:

int socket(int domain, int type, int protocol)

restituisce l'identificatore del socket oppure -1 in caso di errore. Il prototipo e' definito in **<sys/socket.h>**.

Vediamo adesso uno ad uno i vari argomenti della funzione socket.

Socket: endpoint per la comunicazione tra due processi

Socket

- **domain**: famiglia di protocolli da usare. In altre parole specifica come interpretare gli indirizzi una volta che vengono passati. Ad esempio **PF_INET** comunicazione con protocollo IPv4
- **type**: stile di comunicazione ad esempio **SOCK_STREAM** servizio affidabile di consegna degli stream. **SOCK_DGRAM** servzio di consegna del datagramma senza connessione. Oppure **SOCK_RAW** che consente a programmi privilegiati di accedere a protocolli di basso livello o ad interfaccie di rete.
- **protocol**: normalmente vale zero (normalmente dato il tipo di socket e la famiglia il protocollo e' unico).

Socket

- **Esempio socket** vediamo come creare un socket “UDP” ed uno “TCP” (provate prima da soli). La creazione di un socket serve solo ad allocare nel *kernel* le strutture necessarie (in particolare nella *file table*) e a indicare il tipo di protocollo da usare, non viene specificato niente circa gli indirizzi dei processi coinvolti nella comunicazione.

Socket - indirizzi

Visto che le varie famiglie di protocolli prevedono altrettanti tipi differenti di indirizzi, per le varie funzioni che hanno a che fare con i socket e' stata definita una struttura dati generica per gli indrizzi dei socket. Definita in **<sys/socket.h>**:

```
struct  sockaddr {
  sa_family_t sa_family;   // famiglia di indirizzi
  char        sa_data[14]; // indirizzo
};
```

Socket - indirizzi

Ogni famiglia di protocolli prevede poi una diversa struttura dati necessaria alla specifica degli indrizzi. In particolare ci interesseremo di quella relativa alla famiglia dell'IPv4, definita in **<netinet/in.h>**:

```
struct  sockaddr_in {
  sa_family_t sin_family;  // famiglia, deve essere
                           // = AF_INET
  in_port_t   sin_port;    // porta
  struct in_addr  sin_addr;   // indirizzo IP
  unsigned char   sin_zero[8] // per avere stessa
                              // dim. di sockaddr
};
```

Socket - indirizzi

sin_zero: e' definito affinche' la struttura dati abbia le stesse dimensioni di **sockaddr** e deve essere inizializzato a zero.

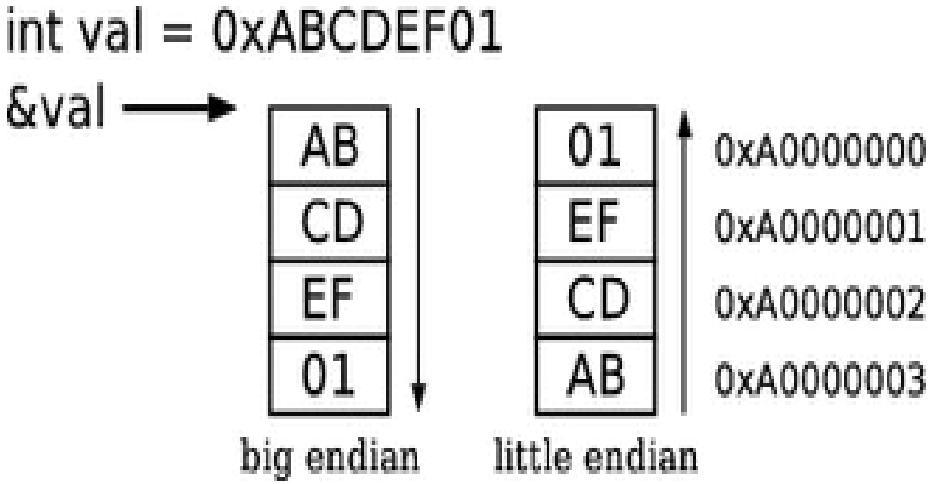
L'indirizzo IP vero e' proprio e' dunque contenuto in **sin_addr**:

```
struct  in_addr {
  u_int32_t  s_addr;  // indirizzo IPv4 di 32 bit
};
```

u_int32_t e' un tipo standard POSIX definito in **<sys/types.h>**

Socket - Indirizzi

Gli indirizzi IP e i numeri di porta devono essere specificati nel formato chiamato ***network order***, che corrisponde al ***big endian*** e che si contrappone al ***little endian*** (due parole sull'endianess).



Esempio endianness: scopriamo, mediante un semplicissimo programma l'endianess della macchina nella quale stiamo lavorando.

Socket - Indirizzi

Risulta chiara a questo punto l'importanza di funzione di marshaling and unmarshaling:

unsigned long int htonl(unsigned long int hostlong)
converte l'intero a 32 bit *hostlong* dal formato macchina al formato rete;
unsigned short int htons(unsigned short int hostshort)
converte l'intero a 16 bit *hostshort* dal formato macchina al formato rete;
unsigned long int ntohl(unsigned long int netlong)
converte l'intero a 32 bit *netlong* dal formato rete al formato macchina;
unsigned short int ntohs(unsigned short int netshort)
converte l'intero a 16 bit *netshort* dal formato rete al formato macchina.