

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

socket: introduzione e utilizzo delle strutture dati

Claudio Vicari

Un esempio per cominciare...

daytimetcpcli.c

- ➔ bisogna avere un processo già attivo, in ascolto sulla porta 13 (daytime) TCP, per esempio usando xinetd
- ➔ a questo punto:
 - daytimetcpcli 127.0.0.1
- ➔ ci dà l'ora del sistema, in accordo con il protocollo
- ➔ fermiamo daytime, e usiamo (almeno su linux):
 - echo "prova di socket" | netcat -l -p 13
- ➔ che crea una connessione in ascolto sulla porta 13, che trasmette la stringa in input...

Specificare gli indirizzi per i socket

- ➔ prima di vedere come si usano i socket, bisogna comprendere come si manipolano gli indirizzi...
- ➔ le funzioni sui socket utilizzano delle strutture che hanno nomi del tipo `sockaddr_*`, con suffissi diversi a seconda del protocollo
- ➔ tipi di socket disponibili:
 - IPv4
 - IPv6
 - Unix domain (socket locali)
 - datalink (comunicazione diretta con ethernet)
 -
- ➔ vedremo anche diverse funzioni in grado di effettuare conversioni di vario genere, utili per i nostri scopi

Struttura per socket IPv4

```
struct in_addr {  
    in_addr_t s_addr;  
};
```

```
struct sockaddr_in {  
    uint8_t      sin_len;    /* lunghezza della  
                             struct, non sempre presente */  
    sa_family_t  sin_family; // AF_INET  
    in_port_t    sin_port;   // numero di porta  
  
    struct in_addr sin_addr;  // indir IPv4, 32bit  
  
    unsigned char sin_zero[8]; // non usato  
};
```

Dettagli

- ➔ le strutture `sin_family`, `sin_port`, `sin_addr` sono le uniche sempre presenti per standard. Possono però esserci altri membri – `sin_zero` per esempio, serve solamente a portare la struttura a 16 byte di lunghezza
- ➔ il campo `sin_len`:
 - non è sempre presente – anche Linux non lo dichiara
 - e anche se è presente, generalmente non serve
 - un campo che indichi la lunghezza è utile per strutture di dimensione variabile

Dettagli /2

- ➔ `sin_family` specifica il tipo di indirizzo specificato dalla struttura `sockaddr` – in questo caso, `AF_INET`
- ➔ `sin_port` è un `in_port_t`, normalmente definito come `uint16_t`, cioè intero a 16 bit senza segno. Rappresenta i numeri di porta TCP e UDP, da 0 a 65535
- ➔ `sin_addr` contiene un intero a 32 bit, sufficiente per rappresentare l'intera gamma di indirizzi IP
- ➔ sia `sin_port` che `sin_addr` sono rappresentati in ***network byte order*** – vedremo fra poco cosa vuol dire e cosa comporta

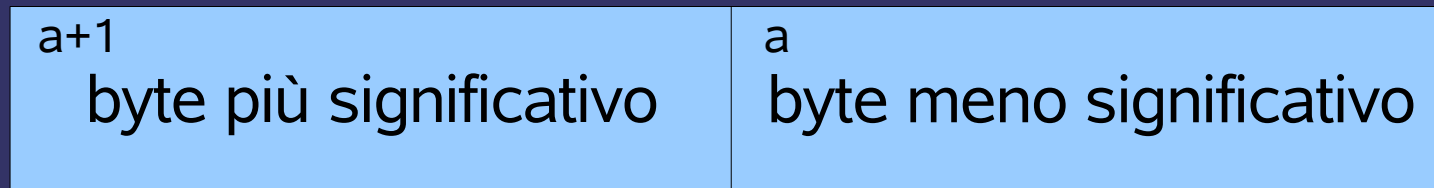
struttura sock_addr generica

```
struct sockaddr {  
    uint8_t sa_len;  
    sa_family_t sa_family;  
    char sa_data[14];  
};
```

- ⇒ questo è un tipo generico, in modo che le funzioni sui socket possano accettare qualsiasi struttura che descriva socket. Questa soluzione è del 1982, prima ancora che si inventasse void* per ANSI C!
- ⇒ esempio:
 - funzione((struct sockaddr*) &addr, sizeof(addr));

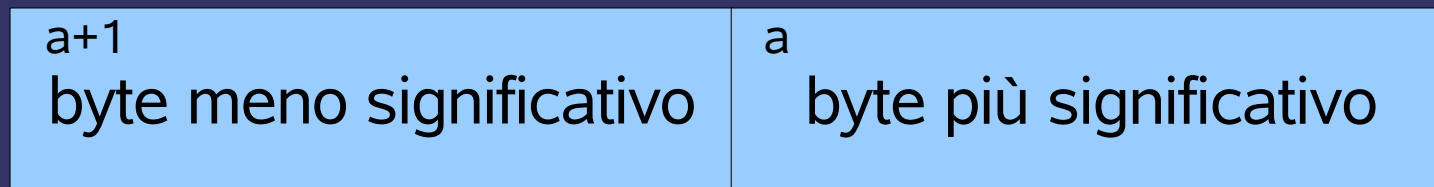
Network byte order

- ➔ dobbiamo rappresentare numeri composti da più di un byte
- ➔ per esempio, un intero a 16 bit può essere rappresentato, nella memoria dell'elaboratore:



little endian

oppure



big endian

Network byte order

- ➔ non c'è uno standard per questo – alcuni sistemi usano uno standard, alcuni l'altro
- ➔ il programma `intro/byteorder.c` ci dice quale dei due modi è adottato dal nostro host
- ➔ la modalità propria del sistema in cui si lavora, viene detta *host byte order*
- ➔ con *network byte order*, ci riferiamo al sistema *big endian*
- ➔ i protocolli di rete, infatti, normalmente utilizzano questo sistema

Network byte order – funzioni di conversione

```
uint32_t htonl( uint32_t hostlong );  
uint16_t htons( uint16_t hostshort );  
uint32_t ntohl( uint32_t netlong );  
uint16_t ntohs( uint16_t netshort );
```

- ➡ in queste funzioni, h vuol dire host, n network
- ➡ ci permettono di effettuare conversioni fra le due modalità
- ➡ nei sistemi che utilizzano il formato big endian, queste funzioni non fanno nulla! Perché big endian == network byte order

conversioni-hton.c

Funzioni per manipolare byte

```
void *memset(void *s, int c, size_t n);  
int memcmp(const void *s1, const void *s2,  
           size_t n);  
void *memcpy(void *dest, const void *src,  
            size_t n);
```

- ➡ con `memset` possiamo scrivere in una zona di memoria una sequenza di byte lunga `n`, tutti uguali a `c`
- ➡ con `memcmp` possiamo confrontare due sequenze di byte
- ➡ con `memcpy` possiamo effettuare la copia di una sequenza di byte
- ➡ lo Stevens usa `bzero`, `bcmp`, `bcpy`, quasi identiche ma *deprecated* perché non standardizzate

Funzioni per manipolare indirizzi IPv4

```
int inet_aton(const char *cp, struct in_addr
    *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

- ➔ `inet_aton` converte una sequenza di caratteri rappresentante un indirizzo IP in una `struct in_addr` (che contiene un `in_addr_t`, cioè un intero a 32 bit)
- ➔ `inet_addr` converte direttamente in un `in_addr_t` – ma non va usata, a causa di alcuni suoi difetti
- ➔ `inet_ntoa` fa l'inverso, converte da una `struct in_addr` a una sequenza di caratteri
- ➔ queste funzioni sono da considerare per motivi storici, meglio usarne altre

Funzioni per manipolare indirizzi IPv4 e IPv6

```
int inet_pton(int af, const char *src,  
              void *dst);  
const char *inet_ntop(int af, const void *src,  
                      char *dst, socklen_t cnt);
```

- ➡ p sta per *presentation*, n per *numeric*
- ➡ family può essere AF_INET o AF_INET6 per IPv4 o IPv6 rispettivamente
- ➡ i char* sono gli argomenti da cui leggere/scrivere le stringhe
- ➡ len specifica la lunghezza a disposizione nel char* - se è meno del necessario, inet_ntop ci restituisce NULL e imposta errno = ENOSPC

Funzioni per manipolare indirizzi: esempio con inet_pton

conversioni-inet_pton.c

- ➔ l'esempio mostra come queste funzioni siano utilizzabili sia con gli indirizzi IPv4 che con Ipv6
- ➔ mostra anche come usare memset, per inizializzare i byte relativi alla struttura per IPv6

Strutture per IPv6

```
struct in6_addr {  
    uint8_t u6_addr8[16] in6_u;  
};
```

```
struct sockaddr_in6 {  
    uint8_t      sin6_len;  
    sa_family_t  sin6_family; //AF_INET6  
    in_port_t    sin6_port;  
    uint32_t      sin6_flowinfo;  
    struct in6_addr sin6_addr;  
}
```

- ➡ del tutto analoga alla struttura per IPv4, con le particolarità di IPv6

Portabilità del codice fra IPv4 e IPv6

- ➔ le funzioni sono flessibili, ma anche `inet_ntop` e `inet_pton` sono diverse nei due casi: bisogna infatti cambiare famiglie e lunghezze
- ➔ usando le strutture generiche, possiamo creare delle funzioni complesse che testano il campo `AF_INET` per costruire di conseguenza le chiamate – e ottenere un codice più portabile
- ➔ lo Stevens crea molte di queste funzioni, le useremo nel seguito, per fare un esempio vediamo:

`sock_ntop_host.c`