Una semplice applicazione client/Server[1]

Il nostro obiettivo

In questa parte del corso implementeremo un'applicazione client/server che usa i socket Internet disponibili nei sistemi Unix/Linux. Nello specifico useremo i `SOCK_STREAM`, che garantiscono una trasmissione dei pacchetti sequenziale e affidabile (grazie all'uso del protocollo TCP). Il sistema sarà molto semplice:

- il **server** si metterà in ascolto su una porta locale, attendendo che i client comunichino con lui; ogni volta che un client lo contatterà gli invierà un messaggio e chiuderà la comunicazione – ovviamente il server dovrà essere in grado di servire *richieste multiple*.
- il **client** contatterà il server e stamperà a video il messaggio ricevuto.

Apposite stampe a video ci informeranno dello stato dei due processi e delle interazioni che stanno avvenendo.
Nelle prossime sezioni andremo a vedere le strutture dati e le system call che il sistema operativo ci mette a disposizione. Attenzione: le dispense non discutono i prototipi delle funzioni, ma sono solo un *vademecum*.

Strutture dati

Un socket è un descrittore di file, ovvero un `int`. Strutture dati più complicate sono invece necessarie per il *set-up* della comunicazione: `addrinfo` prepara le informazioni che serviranno per la definizione dei socket:

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    size_t           ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

[1] Dispense per il corso di Sistemi di Elaborazione Informazione I, Scuola Interfacoltà di Scienze Strategiche, Università di Torino
Docente: Alessia Visconti, http://di.unito.it/~visconti

Alcune strutture dati ignorano la versione di IP, come `sockaddr_storage`:

```
struct sockaddr_storage {
    sa_family_t  ss_family;
    char      __ss_pad1[_SS_PAD1SIZE];
    int64_t   __ss_align;
    char      __ss_pad2[_SS_PAD2SIZE];
};
```

e come `sockaddr`:

```
struct sockaddr {
    unsigned short    sa_family;
    char              sa_data[14];
};
```

Tuttavia, di quest'ultima esistono delle strutture dati specializzate per lavorare con IPv4:

```
struct sockaddr_in {
    short int          sin_family;
    unsigned short int sin_port;
    struct in_addr     sin_addr;
    unsigned char      sin_zero[8];
};

struct in_addr {
    uint32_t s_addr;
};
```

e con IPv6:

```
struct sockaddr_in6 {
    u_int16_t       sin6_family;
    u_int16_t       sin6_port;
    u_int32_t       sin6_flowinfo;
    struct in6_addr sin6_addr;
    u_int32_t       sin6_scope_id;
};

struct in6_addr {
    unsigned char   s6_addr[16];
};
```

`sockaddr` è *compatibile* con entrambe, quindi:

```
struct sockaddr *sa;
struct sockaddr_in sa_ipv4 = (struct sockaddr_in *)sa;
struct sockaddr_in6 sa_ipv6 = (struct sockaddr_in6 *)sa;
```

sono assegnamenti validi.

System Calls

Abbiamo chiuso la sezione precedente parlando si indirizzi IPv4 e IPv6. Ma come si converte un indirizzo codificato in una struttura `sockaddr` nella famigliare notazione puntata? Con questa funzione:

```
#include <arpa/inet.h>
const char *inet_ntop(int af, const void *src,
                char *dst, socklen_t size);
```

le lunghezze degli indirizzi IPv4 e IPv6 sono contenuti in due macro: `INET_ADDRSTRLEN` e `INET6_ADDRSTRLEN`, rispettivamente (in `<netinet/in.h>`).

Per esempio per convertire un indirizzo IPv4 contenuto in una struttura `sockaddr_in sa` e stamparlo a video dovremmo eseguire il seguente codice:

```
...
char ipv4[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &(sa.sin_addr), ipv4, INET_ADDRSTRLEN);
printf("IPv4: %s\n", ipv4);
...
```

Concentriamoci ora sulle sulle system call che ci permetteranno di implementare la nostra applicazione client/server tramite *stream socket*.
Il primo passo che dobbiamo fare è la compilazione delle strutture dati che ci servirà per la connessione al socket, ovvero le `struct addrinfo`. Per farlo sarà necessario inserire solo alcune informazioni salienti e la system call `getaddrinfo` si occuperà di fare il lavoro. La system call `freeaddrinfo()` invece libererà la memoria quando la struttura non sarà più necessaria.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
void freeaddrinfo(struct addrinfo *res);
```

Per esempio se volessimo connetterci sulla porta 4242 del nostro host locale, tramite TCP. lasciando che sia il sistema a scegliere tra IPv4 e IPv6, quello che dovremmo scrivere sarà:

```
...
struct addrinfo hints, *res;
char *node = NULL;
char *service = "4242";
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
int status = getaddrinfo(node, service, &hints, &res);
...
freeaddrinfo(res);
...
```

Dopo aver caricato le informazioni necessarie non ci resta che ottenere il descrittore associato al socket:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

dove i parametri necessari alla funzione sono già codificati nella `struct addrinfo`.
Nel caso di un server devo anche associare il socket a una porta su cui il server stesso sarà in ascolto e cui accetterà le connessioni entranti, attività svolte dalle seguenti system call:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Anche in questo caso tutte le informazioni sono già codificati nella `struct addrinfo`.
Ma come comunicare su una porta associata ad un dato indirizzo? Con la seguente system call:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Ora che la comunicazione è stata stabilita servono due system call per inviare e ricevere messaggi:

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *buf, int len, int flags);
```

Attenzione: queste system call permettono la comunicazione solo su *stream socket*.
L'ultima system call che ci serve è quella che ci permette di chiudere la comunicazione:

```
#include <unistd.h>
int close(int fd);
```

Ora abbiamo tutte le informazioni che ci servono per procedere con l'implementazione!

Letture consigliate

Il materiale presentato in queste dispense segue (abbastanza) fedelmente il libro (in inglese) di Brian "Beej" Jorgenses Hall [B1]. Di questo manuale esiste anche una traduzione in italiano [B2], che però non è aggiornata.
Ricordate sempre che il comando `man` è il modo migliore per avere informazioni puntuali sulle strutture dati e sulle system call utilizzate.

Bibliografia

[B1] Brian "Beej" Jorgenses Hall, *Beej's Guide to Netwroking Programming*, Version 3.0.15, July 2012, http://beej.us/guide/bgnet/

[B2] Brian "Beej" Jorgenses Hall, *Guida di Beej alla Programmazione di Rete*, Version 2.4.5, Agosto 2007, http://linguaggioc.altervista.org/GuidaBeej_it.php