Guida di Beej alla Programmazione di Rete

Usando Socket Internet

Brian "Beej Jorgensen" Hall
beej@beej.us

Versione 2.4.5
5 Agosto 2007

Traduzione di Fabrizio Pani
elfab75@gmail.com

Indice

Indice

1. Introduzione

Ciao! La programmazione Socket ti butta giù? Questa roba è un po' troppo difficile da capire dalle pagine **man**? Vuoi fare della bella programmazione Internet, ma non hai tempo di passare attraverso una moltitudine di `struct` cercando di capire se devi chiamare **`bind()`** prima di **`connect()`**, etc., etc.

Beh, lo sai che? Ho già sbrigato questo sporco affare e muoio dalla voglia di condividere queste informazioni con tutti! Sei nel posto giusto. Questo materiale

dovrebbe dare ad un programmatore C di media competenza, quelle basi di cui ha bisogno per padroneggiare il networking.

1.1. Destinatari

Questo documento è stato scritto come un tutorial, non come una reference. E' probabilmente il migliore quando letto da chi sta appena iniziando con la programmazione socket e cerca un punto d'appoggio. Non è certamente una guida *completa*, ad ogni modo; anche se, si spera, sarà abbastanza perché le pagine man incomincino ad avere senso... `:-)`

1.2. Piattaforma e Compilatore

Il codice contenuto in questo documento è stato compilato su un PC con Linux usando il compilatore Gnu **gcc**. Dovrebbe, in ogni caso, funzionare su ogni piattaforma che usa **gcc**. Naturalmente, questo non si applica se programmi in Windows—vedi la sezione sulla programmazione Windows, qui sotto.

1.3. Sito ufficiale

L'ubicazione ufficiale di questo materiale è `http://beej.us/guide/bgnet/`.

1.4. Nota per i programmatori Solaris/SunOS

Quando compili in Solaris o SunOS, hai bisogno di specificare alcuni switch extra dalla linea di comando per linkare le giuste librerie. Per far ciò, semplicemente aggiungi "`-lnsl -lsocket -lresolv`" alla fine del comando, come:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

Se ottieni degli errori puoi provare ad aggiungere un ulteriore "`-lxnet`" alla fine della linea di comando. Non so cosa fa esattamente, ma alcuni sembrano averne bisogno.

Un altro punto in cui puoi avere dei problemi è nella chiamata a **`setsockopt()`**. Il prototipo differisce da quello della mia Linux box, quindi invece di:

```
int yes=1;
```

Inserisci questo:

```
char yes='1';
```

Poiché non ho una Sun box, non ho testato nessuna delle informazioni di cui sopra—sono solo quelle che alcune persone mi hanno detto via e-mail.

1.5. Nota per i programmatori Windows

A questo punto della guida, storicamente, insulto un po' Windows, semplicemente per il fatto che non mi piace molto, ma dovrei essere veramente imparziale e dirti che

Windows ha un'enorme base d'installazione ed è ovviamente un bellissimo sistema operativo.

Si dice che l'assenza rende il cuore più ardente, e in questo caso, credo che sia vero. Forse è l'età, ma quello che posso dire è che dopo più di una decade in cui non uso sistemi Microsoft per il mio lavoro, sono molto più felice! Per questo, posso sedermi e dirti con sicurezza, "Certo, usa liberamente Windows!" ...Ok si, lo dico a denti stretti.

Per cui t'incoraggio nuovamente a provare Linux, BSD, o qualche sorta di Unix, invece.

Ma alla gente piace quello che piace, e tu popolo di Windows sarai lieto di sapere che questi dati sono in genere applicabili a te ragazzo, con alcuni piccoli cambiamenti.

Una cosa bella che puoi fare è installare Cygwin, che è una raccolta di strumenti Unix per Windows. Ho sentito dire che facendolo si consente a tutti questi programmi di compilare senza modifiche.

Alcuni di voi, però, potrebbero voler fare le cose in Puro Modo Windows. Questo è molto coraggioso da parte tua, e questo è quello che devi fare: corri fuori a procurati Unix immediatamente! No, no—scherzo. Si suppone che sia (più) amico di Windows questi giorni...

Questo è tutto ciò che devi fare (a meno che non installi Cygwin!): primo, ignora del tutto i file che ho menzionato qui. Tutto ciò di cui hai bisogno è includere:

```
#include <winsock.h>
```

Aspetta! Devi fare anche una chiamata a **`WSAStartup()`** prima di compiere qualsiasi cosa con la libreria socket. Il codice per fare questo assomiglia a qualcosa come:

```
#include <winsock.h>

{
    WSADATA wsaData;   // se questo non funziona
    //WSAData wsaData; // allora prova questo

    if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
        fprintf(stderr, "WSAStartup failed.\n");
        exit(1);
    }
```

Devi anche dire al tuo compilatore di linkare la libreria Winsock, di solito chiamata *`wsock32.lib`* o *`winsock32.lib`* o qualcosa di simile. In VC++, questo può essere fatto per mezzo del menu `Project` (`Progetto`), in `Settings...` (`Impostazioni...`). Clicca il tab `Link`, e cerca la casella di testo dal titolo "Object/library modules" ("Oggetto/moduli di libreria") . Aggiungi "wsock32.lib" a quella lista.

O così ho udito.

Infine, hai bisogno di chiamare **`WSACleanup()`** quando hai finito con la libreria socket. Vedi l'help online per i dettagli.

Una volta che lo hai fatto, gli esempi di questo tutorial sono generalmente applicabili, con alcune eccezioni. Per dirne una, non puoi usare **`close()`** per chiudere un socket—necessiti di usare **`closesocket()`**, poi **`select()`** funziona solo con descrittori di socket, non con descrittori di file (come `0` per `stdin`).

C'è anche una classe socket che puoi usare, `CSocket`. Controlla la guida in linea del tuo compilatore per maggiori informazioni.

Per avere ulteriori informazioni sulle Winsock, leggi le FAQ Winsock e parti da lì.

Per finire, ho sentito che Windows non ha una chiamata di sistema **`fork()`**che è, sfortunatamente, usata in alcuni miei esempi. Forse devi collegare una libreria POSIX o qualcosa del genere, o usare invece **`CreateProcess()`**. La funzione **`fork()`**non prende argomenti, mentre **`CreateProcess()`** prende circa *48 miliardi* d'argomenti. Se non sei in grado di usarla, **`CreateThread()`** è un po' più facile da digerire...sfortunatamente una discussione sul multithreading va oltre gli scopi di questo tutorial. Ne posso parlare soltanto molto, lo sai!

1.6. Policy sull'email

Sono generalmente disponibile ad aiutare con domande via e-mail, quindi sentiti libero di scrivermi, ma non posso garantire una risposta. Conduco una vita abbastanza piena e ci sono delle volte in cui non posso proprio rispondere. Quando è il caso, di solito cancello semplicemente il messaggio. Non è niente di personale; non ho proprio il tempo di dare quella risposta dettagliata che chiedi.

Di regola, più complessa è la domanda, meno probabile è che risponda. Se puoi accorciare la tua domanda prima di spedirla e ti assicuri d'includere tutte le informazioni pertinenti (come piattaforma, compilatore, i messaggi d'errore che stai avendo, e qualsiasi altra cosa che pensi potrebbe aiutarmi nel troubleshooting), è molto probabile che avrai una risposta. Per ulteriori indicazioni, leggi il documento ESR, Come fare delle domande in maniera intelligente.

Se non ottieni risposta, lavora su di esso un po' di più, cerca di trovare la risposta, e se ancora ti sfugge, allora riscrivimi con le informazioni che hai trovato e se tutta va bene basterà per aiutarti.

Ora che ti ho seccato su come scrivermi e non scrivermi, vorrei solo farti sapere che apprezzo pienamente tutti gli elogi che questa guida ha ricevuto durante questi anni. E' una vera spinta morale, e mi allieta sentire che è usata bene! :-) Grazie!

1.7. Mirroring

Sei più che benvenuto per fare il mirror del sito, sia pubblicamente che in privato, se fai un mirror pubblico del sito e vuoi che ne faccia un link nella prima pagina, scrivimi due righe a `beej@beej.us`.

1.8. Nota per i Traduttori

Se vuoi tradurre la guida in un altro linguaggio, scrivimi a beej@beej.us e farò un link della tua traduzione nella prima pagina. Sentiti libero di aggiungere nome e contatto alla traduzione.

Per favore nota le restrizioni legali nella sezione Copyright e Distribuzione qui sotto.

Mi spiace, ma per vincoli di spazio, non posso ospitare io stesso la traduzione.

1.9. Copyright e Distribuzione

2. Cos’è un socket?

Senti parlare tutto il tempo di "socket", e forse ti stai chiedendo cosa sia esattamente. Bene, sono questo: un modo di parlare ad altri programmi usando descrittori di file Unix standard.

Cosa?

Ok—puoi aver sentito qualche hacker Unix dichiarare, "Semplice, *ogni cosa* in Unix è un file!" Quello di cui sta parlando è il fatto che quando un programma Unix esegue ogni sorta di I/O, lo fa leggendo o scrivendo un descrittore di file. Un descrittore di file è semplicemente un intero associato ad un file aperto, ma (e qui sta il trucco), quel file può essere una connessione di rete, una FIFO, un pipe, un terminale, un reale file su disco, o qualsiasi altra cosa. *Everything in Unix is a file*! (N.d.T.: *Tutto in Unix è un file*). Quindi quando vuoi comunicare con un altro programma via Internet lo fai attraverso un descrittore di file, devi crederci.

"Dove ottengo questo descrittore di file per la comunicazione di rete, *mister primo della classe*?" è probabilmente l'ultima domanda che ora hai in mente, ti rispondo subito: esegui una chiamata alla routine di sistema **`socket()`**. Questa restituisce un descrittore di socket, e comunichi per mezzo di esso usando le speciali chiamate socket **`send()`** e **`recv()`** (**man send**, **man recv**).

"Ma, ehi!" potresti ora esclamare. "Se è un descrittore di file, perché diavolo non puoi già usare le normali chiamate **`read()`** e **`write()`** per comunicare attraverso il socket?" La risposta breve è, "Puoi!" Quella lunga è, "Puoi, ma **`send()`** e **`recv()`**offrono un controllo molto maggiore nella trasmissione dei dati."

Che altro? Che ne dici di questo: ci sono tutti i tipi di socket. Ci sono indirizzi Internet DARPA (Socket Internet), nomi di percorsi su un nodo locale (Socket Unix), indirizzi CCITT X.25 (Socket X.25 che puoi sicuramente ignorare), e probabilmente molti altri secondo il tipo di Unix che fai girare. Questo tutorial tratta solo il primo: Socket Internet.

2.1. Due Tipi di Socket Internet

Cos'è questo? Ci sono due tipi di socket Internet? Si. Bene, No, bugia. C'è ne sono di più, ma non voglio spaventarti. Qui parlerò solo di due tipi. Eccetto qui, dove accenno ai *Raw Socket* che sono molto potenti e che dovresti controllare.

Bene, quali sono questi due tipi? Uno è lo *Stream Socket* (Lett.: *socket di canale*), l'altro è il *Datagram Socket* (Lett.: socket di datagramma), ai quali puoi fare riferimento d'ora in poi con **`SOCK_STREAM`** e **`SOCK_DGRAM`**, rispettivamente. I datagram socket sono qualche volta chiamati "socket non connessi" (sebbene possano essere connessi se lo vuoi veramente. Vedi **`connect()`** sotto).

I *stream socket* sono dei flussi (*stream* significa flusso) di comunicazione a doppio senso affidabili. Se invii due elementi tramite un socket nell'ordine "1, 2", arriveranno nell'ordine "1, 2" all'altro capo, senza errori. Ne sono così certo, che mi tapperei le orecchie e canterei *la la la la* se qualcuno cerca di sostenere il contrario.

Chi usa i stream socket? Bene, potresti aver sentito dell'applicazione **telnet**, si? Essa usa stream socket Tutti i caratteri arrivano nello stesso ordine in cui li digiti, giusto? Anche i browser che usano il protocollo HTTP utilizzano degli stream socket per caricare le pagine web. Infatti, se esegui telnet su un sito web sulla porta 80, e digiti "`GET / HTTP/1.0`" e premi INVIO (o RETURN) due volte, ti ritorna del codice HTML!

Come raggiungono i stream socket un così altro livello di qualità? Usano un protocollo chiamato *The Transmission Control Protocol* (N.d.T: *Protocollo di Controllo Trasmissione*), altrimenti noto come "TCP" (vedi RFC 793 per informazioni molto dettagliate sul TCP). Questo protocollo assicura che i dati arrivino in sequenza e senza errori. Potresti aver già sentito del "TCP" come della dolce metà del "TCP/IP", dove "IP" sta per "Internet Protocol" (vedi RFC 791). IP tratta principalmente dell'instradamento Internet e non è generalmente responsabile per l'integrità dei dati.

Bello. E i datagram socket? Perché sono chiamati non connessi? Dov'è la convenienza qui, ad ogni modo? Perché non sono affidabili? Bene, ecco alcuni fatti: se invii un datagramma, potrebbe arrivare, ma danneggiato, o arrivare con i dati dentro il pacchetto integri.

Anche i datagram socket usano l'IP per il routing (N.d.T: instradamento), ma non usano TCP; utilizzano lo "User Datagram Protocol" (N.d.T: Protocollo Datagramma Utente), o "UDP" (vedi RFC 768.)

Perché non sono connessi? Bene, fondamentalmente, poiché non hai bisogno di mantenere una connessione aperta come fai con gli stream socket. Semplicemente costruisci un pacchetto, c'inserisci un'intestazione IP con le informazioni sulla destinazione, e lo spedisci. Nessuna connessione è necessaria. Sono generalmente usati quando non è disponibile uno stack TCP o quando pochi pacchetti scartati qui e là non sono la fine del Mondo. Esempi d'applicazioni: **tftp**, **bootp**, videogiochi multiplayer, streaming audio, conferenze video, etc.

"Aspetta un minuto! **tftp** e **bootp** sono usati per trasferire applicazioni binarie da un host ad un altro! I dati non possono andare persi se ti aspetti che l'applicazione funzioni quando arriva! Che diavoleria è questa?"

Bene, amico mio, **tftp** e programmi simili hanno il loro protocollo sopra UDP. Ad esempio, il protocollo TFTP dice che per ogni pacchetto inviato, il destinatario debba spedire indietro un pacchetto che dica, "L'ho preso!" (un pacchetto "ACK"). Se il mittente del pacchetto originale non ottiene risposta diciamo in, cinque secondi, lo rispedirà finché non ottiene un ACK. Questa procedura d'*acknowledgment* (N.d.T: di riconoscimento) è molto importante quando s'implementano applicazioni `SOCK_DGRAM` affidabili.

Per applicazioni non affidabili come giochi, semplicemente ignori i pacchetti persi, o forse cerchi di compensarvi abilmente. (I giocatori di Quake ne manifestano gli effetti con il termine tecnico: *#%$@* lag*.)

2.2. Sciocchezze a Basso Livello e Teoria del Networking

Dato che ho menzionato della stratificazione dei protocolli, è ora di parlare di come funzionano veramente le reti, e mostrare alcuni esempi di come sono formati i pacchetti `SOCK_DGRAM`. In pratica, puoi probabilmente saltare questa sezione. E' un buon background, comunque.

Incapsulamento dei dati.

Ciao, ragazzi, è ora d'imparare *l'Incapsulamento dei Dati*! Questo è molto importante. E' così importante che lo studieresti se fai il corso di reti qui a Chico State ;-). Di base, dice questo: è nato un pacchetto, un pacchetto è avvolto ("incapsulato") da un header (e raramente un footer) dal primo protocollo (diciamo,

quello TFTP), quindi l'intera cosa (compresa l'intestazione TFTP) è di nuovo incapsulata dal successivo protocollo (ad esempio UDP), poi ancora dal prossimo (IP), e ancora dall'ultimo protocollo nel livello hardware (fisico) (diciamo, Ethernet).

Quando un altro computer riceve il pacchetto, l'hardware scarta l'header Ethernet, il kernel estrae le intestazioni IP e UDP, il programma TFTP toglie l'header TFTP, e finalmente ha i dati.

Ora posso finalmente parlare dell'infame *Layered Network Model* (noto anche come "ISO/OSI"). Questo Modello di Network descrive un sistema di funzionalità di rete che ha molti vantaggi rispetto agli altri modelli. Per esempio, puoi scrivere dei programmi con socket che sono esattamente gli stessi senza curarti di come i dati siano trasmessi fisicamente (seriale, thin Ethernet, AUI, qualunque cosa) perché i programmi dei livelli inferiori se ne occupano al tuo posto. La reale rete hardware e la topologia sono trasparenti a chi programma con i socket.

Senza altre chiacchiere, presento tutti gli strati di questo modello. Ricordatene durante gli esami di rete:

- Applicazione
- Presentazione
- Sessione
- Trasporto
- Rete
- Collegamento Dati
- Fisico

Lo strato Fisico è l'hardware (seriale, Ethernet, etc.). Il livello Applicazione è molto lontano da quello Fisico come puoi immaginare—e il luogo in cui gli utenti interagiscono con la rete.

Adesso, questo modello è così generale che potresti probabilmente usarlo come una guida per riparare l'auto se lo volessi veramente. Un modello a strati più conforme a Unix sarebbe:

- Livello Applicazione (*telnet, ftp, etc.*)
- Livello di Trasporto Host-a-Host (*TCP, UDP*)
- Livello Internet (*IP e routing*)
- Livello d'Accesso alla Rete (*Ethernet, ATM, o qualunque altro*)

A questo punto, forse puoi vedere come questi strati corrispondano all'incapsulamento dei dati originali.

Guarda quanto lavoro c'è per costruire un semplice pacchetto? Gesummaria! E devi tu stesso digitare negli header usando "**cat**"! Scherzo. Tutto quello che devi fare per i socket stream è usare `send()` per spedire i dati. Tutto ciò che devi fare per i socket datagram, è incapsulare il pacchetto con un metodo di tua scelta e spedire i dati con `sendto()`. Il kernel fabbrica il *Transport Layer* (Livello di Trasporto) e l'*Internet Layer* (Livello Internet) al tuo posto e l'hardware fa il *Network Access Layer*. Ah, moderna tecnologia.

Così finisce la nostra breve incursione nella teoria delle reti. Oh si, ho dimenticato di dirti tutto quello che volevo sul routing: niente! Esatto, non ne parlerò per niente. Il router estrae l'header IP dal pacchetto, consulta la sua tabella di routing, bla bla bla. Controlla l'RFC IP se te ne importa davvero, se non l'hai mai studiato, beh, sopravviverai.

3. `struct` e Trattamento dei Dati

Ok, eccoci finalmente qui. E' ora di parlare di programmazione. In questa sezione, tratterò dei vari tipi di dati usati dall'interfaccia socket, poiché alcuni di essi sono veramente difficili da capire.

Prima il più facile: un descrittore di socket, che è del seguente tipo:

```
int
```

Solo un standard `int`.

Le cose diventano strane a questo punto, quindi leggi da cima a fondo e abbi pazienza. Sappi questo: ci sono due tipi di ordinamento dei byte: il byte più significativo (qualchevolta chiamato un "otteto") prima, o il byte meno significativo prima. Quello precedente si chiama *Network Byte Order* (Lett.: "Ordine Network dei byte "). Alcune macchine memorizzano internamente i numeri in Network Byte Order, alcune No. Quando dico che qualcosa deve essere in Network Byte Order, devi chiamare una funzione (come **`htons()`**) per convertirla da "Host Byte Order". Se non dico "Network Byte Order", allora devi lasciare il valore in *Host Byte Order* (Lett.: "Ordine Host dei byte").

(Per i curiosi, "Network Byte Order" è anche noto come "Big-Endian Byte Order").

La mia prima struttura è `struct sockaddr`, la quale contiene le informazioni relative all'indirizzo di molti tipi di socket:

```
struct sockaddr {
   unsigned short sa_family;  // famiglia d'indirizzi, AF_xxx
   char          sa_data[14]; //14 byte d'indirizzo protocollo
};
```

`sa_family` può essere diverse cose, ma sarà `AF_INET` per tutto quello che faremo in questo tutorial. *`sa_data`* contiene l'indirizzo di destinazione e il numero di porta del socket. Questo è piuttosto goffo dato che non vuoi impacchettare noiosamente a mano, l'indirizzo in *`sa_data`*.

Per occuparsi della `struct sockaddr`, i programmatori hanno creato una struttura parallela: `struct sockaddr_in` ("in" per "Internet".)

```
struct sockaddr_in {
    short int          sin_family;  // Famiglia indirizzo
    unsigned short int sin_port;    // Numero di porta
    struct in_addr     sin_addr;    // Indirizzo Internet
    unsigned char      sin_zero[8]; // Stessa dimensione di struct
sockaddr
```

```
};
```

Questa struttura facilita il riferimento agli elementi dell'indirizzo socket. Nota che *`sin_zero`* (che viene incluso per eguagliare la lunghezza della struttura a quella di `struct sockaddr`) deve essere tutto azzerato con la funzione **`memset()`**. Poi, e questo è *importante*, si può fare il cast di un puntatore a `struct sockaddr_in` ad un puntatore a `struct sockaddr`, e viceversa. Quindi benché **`connect()`** voglia una `struct sockaddr*`, puoi ancora usare `struct sockaddr_in` e farne il cast all'ultimo momento! Nota anche che *`sin_family`* corrisponde a *`sa_family`* in `struct sockaddr` e deve essere impostata a "`AF_INET`". Infine *`sin_port`* e *`sin_addr`* devono essere in *Network Byte Order*!

"Ma" obietti, "come può l'intera struttura, `struct in_addr sin_addr`, essere in Network Byte Order?" Questa domanda richiede un esame accurato della struttura `struct in_addr`, una delle peggiori union in vita:

```
//Indirizzo Internet (una struttura per motivi storici)
struct in_addr {
    uint32_t s_addr; // questo è un int a 32-bit (4 byte)
};
```

Bene, era solita essere un'unione, ma quei tempi ora sembrano essersene andati. Una liberazione. Per cui se hai dichiarato *`ina`* di tipo `struct sockaddr_in`, allora *`ina.sin_addr.s_addr`* si riferisce ad un indirizzo IP di 4-byte (in Network Byte Order). Nota che perfino se il tuo sistema utilizza ancora la pessima union per `struct in_addr`, puoi sempre riferirti all'indirizzo IP di 4 byte esattamente come sopra (questo è dovuto ai `#define`).

3.1. Converti i Nativi!

Siamo proprio giunti alla prossima sezione. Si è discusso troppo di questa conversione da Network a Host Byte Order—ora è tempo di agire!

Benissimo. Ci sono due tipi che puoi convertire: `short` (due byte) e `long` (quattro byte). Queste funzioni lavorano allo stesso modo per le varianti `unsigned`. Diciamo che vuoi convertire uno `short` da Host Byte Order a Network Byte Order. Inizia con "h" per l'"host", fallo seguire da "to", poi "n" per "network", e "s" per "short": h-to-n-s, or **`htons()`** (da leggere: "Host to Network Short").

E' fin troppo facile...

Puoi usare qualsiasi combinazione di "n", "h", "s", e "l" che vuoi, senza contare quelle veramente stupide. Per esempio, NON c'è una funzione **`stolh()`** ("Short to Long Host"), ma ci sono:

`htons()`	Da **h**ost **a** **n**etwork **s**hort
`htonl()`	Da **h**ost **a** **n**etwork **l**ong

`ntohs()` Da **n**etwork **a** **h**ost **s**hort

`ntohl()` Da **n**etwork **a** **h**ost **l**ong

Ora puoi pensare di essere informato. Potresti pensare, "Che faccio se devo cambiare l'ordine dei byte su un `char`?" Allora penseresti, "Uh, non importa." Potresti anche considerare che poiché la tua macchina 68000 già usa indirizzi in network byte order, non hai bisogno di chiamare **`htonl()`** sui tuoi indirizzi IP. Potresti aver ragione, *MA* se fai il port su una macchina che ha un ordine dei byte inverso, il tuo programma non funziona. Sii portabile! Questo è un mondo Unix! (Per quanto a Bill Gates piacerebbe diversamente). Ricorda: poni i tuoi byte in Network Byte Order prima di metterli in rete.

Un'ultima cosa: perché *`sin_addr`* e *`sin_port`* hanno bisogno di essere in Network Byte Order in una `struct sockaddr_in`, ma *`sin_family`* no? La risposta: *`sin_addr`* e *`sin_port`* sono incapsulati in un pacchetto a livello IP e UDP, rispettivamente. Perciò, devono essere in Network Byte Order. Il campo *`sin_family`* è usato solo dal kernel per determinare che tipo d'indirizzo contiene la struttura, quindi deve essere in Host Byte Order. Inoltre, dato che *`sin_family`* non è inviata in rete, può essere in Host Byte Order.

3.2. Indirizzi IP e come trattarli

Per tua fortuna, ci sono un gruppo di funzioni che ti consentono di manipolare indirizzi IP. Non c'è bisogno di calcolarli a mano e infilarli in una `long` con l'operatore <<.

Primo, diciamo che hai una `struct sockaddr_in ina`, e hai un indirizzo IP "`10.12.110.57`" che vuoi memorizzare. La funzione che vuoi usare, **`inet_addr()`**, converte un indirizzo IP in notazione numeri e punti, in un long senza segno. Si può dunque fare l'assegnazione:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Nota che **`inet_addr()`** restituisce l'indirizzo già in Network Byte Order—non devi chiamare **`htonl()`**. Bellissimo!

Ora, il frammento di codice di cui sopra non è molto robusto perché non c'è un controllo degli errori. Guarda, **`inet_addr()`** restituisce `-1` in caso di errore. Ricordi i numeri binari? Succede che `(unsigned)-1` corrisponde proprio all'indirizzo IP `255.255.255.255`! Questo è l'indirizzo di broadcast! Sbagliato. Ricorda di controllare gli errori regolarmente.

Per la verità, c'è un'interfaccia più chiara che puoi usare invece di **`inet_addr()`**: si chiama **`inet_aton()`** ("aton" significa "ascii to network"):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Ed ecco un semplice uso, mentre s'impacchetta una `struct sockaddr_in` (questo esempio avrà più senso quando andrai alle sezioni **`bind()`** e **`connect()`**.)

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;         // host byte order
my_addr.sin_port = htons(MYPORT);     // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);
```

`inet_aton()`, *a differerenza praticamente di ogni altra funzione legata ai socket*, non restituisce zero in caso di successo, e l'indirizzo è passato a *`inp`*.

Sfortunatamente, non tutte le piattaforme implementano **`inet_aton()`**così, sebbene si preferisce il suo uso, viene usato il più vecchio e comune **`inet_addr()`** in questa guida.

Benissimo, ora puoi convertire stringhe d'indirizzi IP nelle loro rappresentazioni binarie. E il contrario? Cosa accade se hai una `struct in_addr` e vuoi stampare nella notazione numeri e punti? In questo caso, vorrai usare la funzione **`inet_ntoa()`** ("ntoa" significa "network to ascii") come questa:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

Quella stamperà l'indirizzo IP. Nota che **`inet_ntoa()`**prende una `struct in_addr` come argomento, non un `long`. Nota anche che restituisce un puntatore ad un char. Questo punta ad un array statico di caratteri entro **`inet_ntoa()`** cosicché ogni volta che chiami **`inet_ntoa()`** sovrascrive l'ultimo indirizzo IP che hai chiesto. Per esempio:

```
char *a1, *a2;

a1 = inet_ntoa(ina1.sin_addr);  // questo è 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr);  // questo è 10.12.110.57
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

stamperà:

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

Se hai bisogno di salvare l'indirizzo, **`strcpy()`** sul tuo array di caratteri.

Questo è tutto sul soggetto per ora. Più tardi, imparerai a convertire una stringa come "whitehouse.gov" nel suo corrispondente indirizzo IP (guarda DNS, più avanti.)

3.2.1. Reti private (O Disconnesse)

Molti hanno un firewall per nascondere la rete dal resto del mondo per protezione, e spesse volte il firewall traduce gl'indirizzi IP "interni" in indirizzi IP "esterni" (che chiunque può conoscere), usando un procedimento chiamato *Network Address Translation*, o NAT.

Stai diventando già nervoso? "Dove va con questa strana roba?"

Bene, rilassati e comprati un drink, perché come principiante, non ti devi neanche preoccupare di un NAT, poiché viene fatto in modo trasparente. Ma volevo parlarti della rete dietro il firewall, nel caso che incominci a diventare confuso dai numeri di network che vedi.

Ad esempio, a casa ho un firewall. Ho due indirizzi IP statici che mi sono stati assegnati dal provider, e ho inoltre sette computer in rete. Com'è possibile? Due computer non possono condividere lo stesso indirizzo IP, o altrimenti i dati non saprebbero dove andare!

La risposta è: non condividono lo stesso indirizzo IP. Sono in una rete privata con 24 milioni d'indirizzi IP allocati. Tutti per me. Bene, sia per me che per qualsiasi altro. Ecco cosa succede:

Se mi autentico ad un computer remoto, dice che mi sono collegato dall'indirizzo 64.81.52.10 (non il mio vero IP), ma se chiedo al mio computer locale che indirizzo ho, dice 10.0.0.5. Chi traduce l'indirizzo IP dall'uno all'altro? Esatto, il firewall! Sta facendo NAT!

10.*x*.*x*.*x* è uno dei pochi network riservati che devono essere usati solo in reti pienamente disconnesse, o in reti che stanno dietro dei firewall. Dettagli su quali numeri di reti private sono disponibili per essere usati sono descritti nell'RFC 1918, ma alcuni comuni sono 10.*x*.*x*.*x* e 192.168.*x*.*x*, dove *x* è 0-255. Meno comune è 172.*y*.*x*.*x*, dove *y* va da 16 a 31.

Le reti dietro un firewall NAT non hanno bisogno di essere una di queste reti riservate, ma comunemente lo sono.

4. Chiamate di sistema

Questa è la sezione dove entriamo dentro le chiamate di sistema che ci con cui accediamo alle funzionalità di rete di una box Unix. Quando chiami una di queste funzioni, il kernel prende il controllo e fa, per magia, tutto il lavoro al tuo posto.

Il punto in cui la maggior parte delle persone s'impantanano è in quale ordine chiamare tutte queste cose. In ciò le pagine **man** non sono utili, come probabilmente hai scoperto. Bene, per aiutarti in questa orribile situazione, ho cercato di delineare le system call nella sezione seguente *esattamente* nello stesso ordine in cui hai bisogno di chiamarle nel tuo programma.

Tutto questo associato ad alcuni brani di codice d'esempio qua e là, un po' di latte e biscotti (che ho paura devi procurarti tu stesso), e un po' di coraggio, e sarai in grado d'inviare dati in Internet come il figlio di Jon Postel!

4.1. `socket()`—Ottieni il Descrittore di File!

Suppongo che non posso più rimandare—Devo parlare della chiamata di sistema **`socket()`**. Ecco un abbozzo:

```
#include <sys/types.h>
#include <sys/socket.h>

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

Cosa sono questi argomenti? Primo, *`domain`* deve essere settato a "`PF_INET`". Poi l'argomento *`type`* dice al kernel che tipo di socket è: `SOCK_STREAM` o `SOCK_DGRAM`. Infine, imposta *`protocol`* giusto a "`0`" per far scegliere a **`socket()`** il corretto protocollo in base a *`type`*. (Nota: ci sono molti più *`domain [N.d.T: domini]`* di quelli che ho elencato, così come molti più *`type`*. Vedi la man page di **`socket()`**. C'è anche un modo "migliore" di impostare *`protocol`*, ma specificare `0` funziona nel 99.9% dei casi. Guarda la pagina man di **`getprotobyname()`** se sei curioso).

`socket()` ti restituisce semplicemente un descrittore di socket che puoi usare più tardi nelle chiamate di sistema, o `-1` su errore. La variabile globale *`errno`* è impostata al valore dell'errore (Guarda la man page di **`perror()`**).

(Questo `PF_INET` è un parente stretto di `AF_INET` che hai usato quando inizializzavi il campo *`sin_family`* nella tua `struct sockaddr_in`. Di fatto, sono così intimamente correlati che in effetti hanno lo stesso valore, e molti programmatori chiamano **`socket()`**e passano `AF_INET` come primo argomento invece di **`PF_INET`**. Adesso prendi un po' di latte e biscotti, perché è ora di storia. Una volta, molto tempo fa, si pensava che forse una famiglia d'indirizzi (quello che "AF" in "`AF_INET`" sta per) potesse supportare diversi protocolli a cui si faceva riferimento tramite la relativa famiglia di protocolli (quello che "PF" in "`PF_INET`" sta per). Non accadde e ora vivono felicemente fino alla fine. Così la cosa più corretta da fare è usare `AF_INET` nella tua `struct sockaddr_in` e `PF_INET` nella tua call a **`socket()`**).

Bene, bene, bene, ma quanto buono è questo socket? la risposta è che non è realmente buono di per sé, e hai bisogno di leggere ed eseguire più chiamate di sistema perché abbia senso.

4.2. `bind()`—Su che porta sono?

Una volta che hai un socket, potresti voler associare quel socket ad una porta della tua macchina locale. (Questo viene fatto normalmente se userai **`listen()`** per le connessioni in ingresso su una porta specifica—i MUD fanno questo quando ti dicono "telnet to x.y.z port 6969"). Il numero di porta è usato dal kernel per far corrispondere un pacchetto in entrata ad un certo processo di un descrittore di socket; se esegui solo una **`connect()`** questo non è necessario. Leggilo comunque, giusto come spunto.

Ecco Il prototipo della system call **`bind()`**:

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd è il descrittore restituito da **socket()**. *my_addr* è un puntatore a `struct sockaddr` che contiene informazioni sull'indirizzo, cioè, porta e numero IP. *addrlen* può essere settata a `sizeof *my_addr` oppure a `sizeof(struct sockaddr)`.

Okay. C'è un bel po' da acquisire. Facciamo un esempio:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 3490

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(PF_INET, SOCK_STREAM, 0); /* fai un po' di
controllo degli               errori!*/

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte
order
    my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
    memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

    // non dimenticare il controllo degli errori in bind():
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);
    .
    .
    .
```

Qui ci sono alcune cose da notare: *my_addr.sin_port* è in Network Byte Order, e così è *my_addr.sin_addr.s_addr*. Un'altra cosa su cui fare attenzione è che i file header potrebbero differire da sistema a sistema. Per essere sicuro, dovresti controllare le pagine **man**.

Infine, sull'argomento **bind()**, dovrei menzionare che possono essere automatizzati alcuni procedimenti per ottenere il tuo indirizzo IP o porta:

```
my_addr.sin_port = 0; // sceglie una porta a caso non in uso
my_addr.sin_addr.s_addr = INADDR_ANY; // usa il mio indirizzo IP
```

Vedi, impostando *my_addr.sin_port* a zero, stai dicendo a **bind()** di scegliere una porta al posto tuo. Ugualmente, settando *my_addr.sin_addr.s_addr* a `INADDR_ANY`,

gli dici di riempirla automaticamente con l'indirizzo IP della macchina in cui è in esecuzione il processo.

Se sei uno che nota i particolari, potresti aver visto che non ho posto `INADDR_ANY` in Network Byte Order! Che cattivo. Ad ogni modo ho informazioni riservate: `INADDR_ANY` è in realtà zero! Zero è sempre zero perfino se riordini i bit. Comunque, i puristi faranno presente che ci potrebbe essere una dimensione parallela dove `INADDR_ANY` è, diciamo, 12, e lì il mio codice non funzionerà. Questo per me è Ok:

```
my_addr.sin_port = htons(0); // sceglie una porta a caso non in uso
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // usa il mio
indirizzo IP
```

Ora siamo così portabili che forse non ci crederesti. Volevo solo fartelo notare, poiché la maggior parte del codice che incontrerai non darà fastidi con `INADDR_ANY` attraverso **`htonl()`**.

`Anche bind()` ritorna `-1` su errore e setta *`errno`* al valore dell'errore.

Un'altra cosa da cui stare in guardia quando si chiama **`bind()`**: non mettere qualunque numero di porta. Tutte le porte sotto la 1024 sono RISERVATE (a meno che non sei superuser!). Puoi usare qualsiasi numero di porta al di sopra, giusto fino a 65535 (posto che non sia già usato da qualche altro programma).

Talvolta potresti notare che, cercando di riavviare un server, **`bind()`** fallisce lamentando un "Address already in use" (N.d.T: indirizzo già in uso). Cosa significa questo? Bene, un po' del socket che era connesso sta ancora aspettando nel kernel, accaparrandosi la porta. Puoi sia attendere che si liberi (un minuto o più), o aggiungere del codice al tuo programma per consentirgli di riusare la porta, come questo:

```
int yes=1;
//char yes='1'; // Gli utenti di Solaris usino questo

//eviti il fastidioso messaggio di errore "Address already in use"
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int))
== -1) {
    perror("setsockopt");
    exit(1);
}
```

Ancora una piccola nota extra su **`bind()`**: ci sono delle volte in cui non sei tenuto a chiamarlo. Se fai **`connect()`** ad una macchina remota e non t'importa qual'è la tua porta locale (come in **telnet** dove t'importa solo la porta remota), puoi semplicemente chiamare **`connect()`**, che controllerà se il socket è libero, e ne farà il **`bind()`** ad una porta locale non in uso se necessario.

4.3. `connect()`—Ehi, tu!

Facciamo finta per alcuni minuti che sei un'applicazione. Il tuo utente t'impartisce un comando (proprio come nel film *TRON*) per ottenere un descrittore di socket. Tu

obbedisci e chiami **`socket()`**. Dopo l'utente ti dice di connetterti a "`10.12.110.57`" sulla porta "`23`" (la porta telnet standard). Oh mio Dio! Che fai ora?

Per tua fortuna, stai esaminando la sezione **`connect()`**—come connettersi ad un host remoto. Quindi leggi avidamente! Non c'è tempo da perdere!

La call **`connect()`** è come segue:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` è il nostro "vicino" descrittore di socket, come restituito dalla chiamata a **`socket()`**, *`serv_addr`* è una `struct sockaddr` che contiene la porta di destinazione e l'indirizzo IP, e *`addrlen`* può essere impostata a `sizeof *serv_addr` oppure a `sizeof(struct sockaddr)`.

Inizia ad avere più senso tutto ciò? Vediamo un esempio:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEST_IP   "10.12.110.57"
#define DEST_PORT 23

int main(void)
{
    int sockfd;
    struct sockaddr_in dest_addr;   // conterrà l'indirizzo di
destinazione

    sockfd = socket(PF_INET, SOCK_STREAM, 0); /* esegui il
controllo degli errori!*/

    dest_addr.sin_family = AF_INET;          // host byte order
    dest_addr.sin_port = htons(DEST_PORT);   // short, network byte
order
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    memset(dest_addr.sin_zero, '\0', sizeof dest_addr.sin_zero);

    // non dimenticare di controllare connect()!
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof
dest_addr);
    .
    .
    .
```

Di nuovo, assicurati di controllare il valore restituito da **`connect()`**—che ritorna `-1` in caso di errore e setta la variabile *`errno`*.

Inoltre, considera che non abbiamo chiamato **`bind()`**. Fondamentalmente non ce ne importa del nostro numero di porta locale; c'importa solo dove stiamo andando (porta remota). Il kernel sceglierà una porta locale per noi, e il sito a cui ci connettiamo otterrà automaticamente da noi quest'informazione. Nessuna preoccupazione.

4.4. `listen()`—Qualcuno può chiamarmi?

Ok, è ora di cambiare. Cosa succede se non vuoi collegarti ad un host remoto, diciamo solo perché vuoi accettare connessioni entranti e trattarle in qualche modo? Il processo è in due passi: prima usi **`listen()`**, poi **`accept()`** (vedi sotto).

La chiamata Listen è abbastanza semplice, ma richiede qualche spiegazione:

```
int listen(int sockfd, int backlog);
```

`sockfd` è il solito descrittore di socket della system call **`socket()`**. *`backlog`* è il numero di connessioni consentite in coda. Cosa significa questo? Bene, le connessioni in ingresso aspetteranno fino a che le accetti (con **`accept()`**) (vedi più avanti) e questo è il limite di quante possono accodarsi. La maggior parte dei sistemi limitano silenziosamente questo numero a circa 20; puoi probabilmente cavartela con `5` o `10`.

Ancora una volta, come d'abitudine, **`listen()`** restituisce `-1` e imposta *`errno`* in caso di errore.

Bene, come forse puoi immaginare, hai bisogno di chiamare **`bind()`** prima di **`listen()`** o il kernel ci ascolterà su una porta random. Bleah! Quindi, se vuoi stare in ascolto di connessioni entranti, la sequenza delle chiamate di sistema che farai è:

```
socket();
bind();
listen();
/* accept() va qui */
```

Quello lo lascio alla parte dei codici d'esempio, perché si spiega abbastanza da sé. (Il codice nella sezione **`accept()`**, qui sotto, è più completo). La parte veramente difficile di quest'intera sha-bang è la chiamata **`accept()`**.

4.5. `accept()`—"Grazie per aver chiamato la porta 3490."

Preparati—la call **`accept()`** è abbastanza strana! Quello che succederà e questo: qualcuno da molto, molto lontano, cercherà di connettersi (con **`connect()`**)alla tua macchina su una porta in cui sei in ascolto (**`listen()`**). La loro connessione sarà posta in coda aspettando di essere ricevuta (**`accept()`**). Chiami **`accept()`** e gli dici di prendere la connessione in sospeso. Ti restituirà un *nuovissimo descrittore di socket* da usare per questa singola connessione! Esatto, improvvisamente hai *due descrittori di socket* al prezzo di uno! Quello originale sta ancora ascoltando sulla tua

porta e il nuovo creato è finalmente pronto ad inviare (**`send()`**) e a ricevere (**`recv()`**). Ci siamo!

La call è la seguente:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

`sockfd` è il descrittore di **`listen()`**. Abbastanza facile. *`addr`* sarà di solito un puntatore ad una locale `struct sockaddr_in`. Qui è dove andranno le informazioni su una connessione in ingresso (e con esso puoi determinare che host ti chiama e da quale porta). *`addrlen`* è una variabile locale intera che dovrebbe essere impostata a `sizeof(*addr)` o `sizeof(struct sockaddr_in)` prima che il suo indirizzo sia passato ad **`accept()`**. Questa funzione non metterà più di un certo numero di byte in *`addr`*. Se ne mette pochi, modificherà il valore di *`addrlen`* per riflettere ciò.

Indovina? **`accept()`** restituisce `-1` e setta *`errno`* se si verifica un errore. Scommetto che non lo avevi indovinato.

Come prima, c'è molto da capire in un colpo solo, ecco quindi un frammento di codice per i vostri studi:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490    // la porta in cui si connetteranno gli
utenti

#define BACKLOG 10     // quante connessioni in sospeso conterrà la
coda

int main(void)
{
    int sockfd, new_fd;  // ascolta in sock_fd, nuove connesioni in
new_fd
    struct sockaddr_in my_addr;    // informazioni sul mio
indirizzo
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette
    int sin_size;

    sockfd = socket(PF_INET, SOCK_STREAM, 0); // fai il check degli
errori!

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte
order
```

```
    my_addr.sin_addr.s_addr = INADDR_ANY; /* riempio
automaticamente con il mio IP*/
    memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

    // non dimenticare il tuo controllo degli errori per queste
chiamate:
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr);

    listen(sockfd, BACKLOG);

    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
&sin_size);
    .
    .
    .
```

Di nuovo, nota che useremo il descrittore *new_fd* per tutte le call **send()**e **recv()**. Se mai vuoi avere una singola connessione, puoi chiudere (**close()***sockfd* in ascolto, per prevenire ulteriori connessioni entranti sulla stessa porta.

4.6. send() e recv()—Parlami, baby!

Queste due funzioni servono per comunicare su stream socket o datagram socket connessi, se vuoi usare regolari datagram socket, non connessi, hai bisogno di vedere la sezione **sendto() e recvfrom()**, di sotto.

La chiamata **send()**:

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd è il descrittore del socket che vuoi spedisca i dati (sia che sia quello restituito **socket()** o quello avuto da **accept()**). *msg* è un puntatore ai dati che vuoi inviare, e *len* è la lunghezza in byte dei dati. Imposta solo *flags* a 0. (Vedi la man page di **send()**per maggiori informazioni su questi flag).

Un codice d'esempio potrebbe essere:

```
char *msg = "Beej era qui!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

send() restituisce il numero di byte effettivamente inviati—*questo può essere meno del numero che gli hai detto di spedire!* Vedi, talvolta gli dici di inviare una gran

quantità di dati che non può gestire. Invierà quanto più dati possibili, e fidati che spedirà i restanti più tardi. Ricorda, se il valore ritornato da **`send()`** non corrisponde a quello di *`len`*, spetta a te inviare il resto della stringa. La buona notizia è che: se il pacchetto è piccolo (meno di 1K o circa) *probabilmente* riuscirai ad inviarlo tutto in una volta. Come sempre, viene restituito `-1` su errore, ed *`errno`* è impostata al suo valore.

La call **`recv()`** è simile in molti aspetti:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

`sockfd` è il descrittore di socket da cui leggere, *`buf`* è il buffer per leggere le informazioni, *`len`* è la lunghezza massima del buffer, e *`flags`* può essere nuovamente messo `0`. (**`recv()`** man page per info sui flags.)

`recv()` restituisce il numero di byte effettivamente letti nel buffer, o `-1` in caso di errore *`errno`* settata di conseguenza).

Aspetta! **`recv()`** può restituire `0`. Questo può significare solo una cosa: la parte remota ti ha chiuso la connessione! Un valore di ritorno `0` è il modo in cui **`recv()`** ti fa sapere che ciò si è verificato.

E' stato facile, o No? Ora puoi inviare e ricevere dati su stream socket! Fantastico! Sei un Programmatore di rete Unix!

4.7. `sendto()` e `recvfrom()`—parlami, in stile DGRAM

"Questo è tutto bello è grandioso," Ti sento dire, "ma perché questo mi lascia con dei datagram socket non connessi?" No problem, amico. Abbiamo quello di cui c'è bisogno.

Poiché i datagram socket non sono connessi ad un host remoto, indovina quale tipo d'informazione abbiamo bisogno, prima d'inviare un pacchetto? Esatto! L'indirizzo di destinazione! Ecco lo scoop:

```
int sendto(int sockfd, const void *msg, int len, unsigned int
flags,
           const struct sockaddr *to, socklen_t tolen);
```

Come puoi vedere, questa chiamata è di base la stessa di quella **`send()`** con l'aggiunta d'altri due pezzi d'informazioni. *`to`* è un puntatore a `struct sockaddr` (la quale forse hai come una `struct sockaddr_in` e ne fai il cast all'ultimo minuto) che contiene l'indirizzo IP e di destinazione e una porta. *`tolen`*, in fondo un intero `int`, può essere semplicemente impostato a `sizeof(*to)` o a `sizeof(struct sockaddr)`.

Così come **`send()`**, **`sendto()`** restituisce il numero di byte effettivamente inviati (i quali, ancora, potrebbero essere di meno di quelli che gli hai detto di spedire!), o `-1` in caso di errore.

Ugualmente simili sono **recv()** e **recvfrom()**. Il prototipo di **recvfrom()** è:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

Di nuovo, questo è proprio come **recv()** con l'aggiunta di un paio di campi. *from* è un puntatore ad una `struct sockaddr` locale che sarà riempita con l'indirizzo IP e la porta della macchina da cui hanno origine. *fromlen* è un puntatore ad un `int` locale che deve essere inizializzata a `sizeof *from` oppure a `sizeof(struct sockaddr)`. Quando la funzione ritorna, *fromlen* conterrà la lunghezza dell'indirizzo memorizzato proprio in *from*.

recvfrom() restituisce il numero di byte ricevuti, o `-1` su error (con *errno* settata in conformità).

Ricorda, se usi **connect()** con un datagram socket, poi puoi semplicemente utilizzare **send()** e **recv()** for per tutte le tue transazioni. Di per sé il socket è ancora un datagram socket e i pacchetti usano ancora UDP, ma l'interfaccia aggiungerà automaticamente destinazione e sorgente al tuo posto.

4.8. `close()` e `shutdown()`—Vattene!

Okay! Hai inviato (**send()**) e ricevuto (recv()) dati tutto il giorno, e ne hai abbastanza. Sei pronto a chiudere la connessione al tuo descrittore di socket. Questo è facile. Puoi usare proprio la funzione close() per i normali descrittori Unix:

```
close(sockfd);
```

Ciò impedirà successive letture e scritture sul socket. Chiunque proverà a leggere e a scrivere con il socket all'altro capo, riceverà un errore.

Giusto nel caso voglia avere un po' più di controllo nel modo in cui chiudere un socket, puoi usare la funzione **shutdown()**; che ti consente di tagliare la comunicazione in una certa direzione, o in entrambe (proprio come fa **close()**) . Prototipo:

```
int shutdown(int sockfd, int how);
```

sockfd è il descrittore del socket che desideri chiudere, e *how* è uno dei seguenti:

`0`	Non sono consentite nuove ricezioni
`1`	Non sono consentiti altri invii
`2`	Non sono permessi ulteriori invii e ricezioni (come **close()**)

shutdown() restituisce `0` nel caso di successo, e `-1` su errore (con *errno* settata di conseguenza.)

Se ti degni di usare **`shutdown()`** su datagram socket non connessi, semplicemente renderà il socket indisposto ad ulteriori chiamate a **`send()`** e **`recv()`** (ricorda che le puoi usare se connetti (**`connect()`**) il tuo datagram socket).

E' importante notare che **`shutdown()`** effettivamente non chiude il descrittore di file—ne modifica solo la fruibilità. Per liberare un descrittore di socket hai bisogno di usare **`close()`**.

Non ci vuole niente.

(Eccetto di ricordarti che sei stai usando Windows e le Winsock, devi chiamare **`closesocket()`** invece di **`close()`**).

4.9. `getpeername()`—Chi sei tu?

Questa funzione è così facile.

Così facile, che stavo per non dargli la sua sezione, ma ad ogni modo eccola.

La funzione **`getpeername()`** ti dirà chi c'è dall'altro capo di un socket stream connesso. Il prototipo:

```
#include <sys/socket.h>

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

`sockfd` è il descrittore dello stream socket connesso, *`addr`* è puntatore a `struct sockaddr` (o a `struct sockaddr_in`) che conterrà le informazioni della parte remota della connessione, e *`addrlen`* è un puntatore ad un `int`, che dovrebbe essere inizializzato a `sizeof *addr` or `sizeof(struct sockaddr)`.

La funzione ritorna `-1` su error e imposta *`errno`* in conformità.

Una volta che hai il loro indirizzo, puoi usare **`inet_ntoa()`** o **`gethostbyaddr()`** per stampare od ottenere più informazioni. No, non puoi ottenere il loro nome di login. (Ok, ok. Se l'altro computer ha in esecuzione un daemon ident, questo è possibile. Ciò, ad ogni modo, è al di là degli scopi di questo documento. Controlla RFC 1413 per maggiori info).

4.10. `gethostname()`—Chi sono?

Persino più facile di **`getpeername()`** è la funzione **`gethostname()`**. Essa restituisce il nome del computer in cui è in esecuzione il tuo programma. Il nome può essere utilizzato da **`gethostbyname()`**, qui sotto, per determinare l'indirizzo IP della tua macchina locale.

Cosa c'è di più divertente? Potrei pensare a poche cose, le quali non sono pertinenti con la programmazione socket. Comunque, ecco un abbozzo:

```
#include <unistd.h>
```

```
int gethostname(char *hostname, size_t size);
```

Gli argomenti sono semplici: *`hostname`* è un puntatore ad un array di caratteri che contiene l'hostname che la funziona restituisce, e *`size`* è la lunghezza in byte dell'array *`hostname`*.

La funzione ritorna `0` se è completata con successo, e `-1` su errore, impostando *`errno`* come al solito.

4.11. DNS—Tu dici "whitehouse.gov", io dico "63.161.169.137"

Nel caso che tu non sappia cosa sia un DNS, esso sta per "Domain Name Service". In poche parole, tu gli dici un indirizzo leggibile da un essere umano, e lui ti dà l'indirizzo IP (così lo puoi usare con **`bind()`**, **`connect()`**, **`sendto()`**, o altre funzioni). In questo modo, quando qualcuno inserisce:

```
$ telnet whitehouse.gov
```

telnet può trovare che il server a cui connettersi (con **`connect()`**) è "63.161.169.137".

Come funziona? Userai la funzione **`gethostbyname()`**:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Come vedi, restituisce un puntatore a `struct hostent`, la cui struttura è la seguente:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
```

Ecco qui la descrizione dei campi di `struct hostent`:

`h_name`	Nome canonico dell'host.
`h_aliases`	Un array zero terminato di nomi alternativi per l'host.
`h_addrtype`	Il tipo d'indirizzo restituito; solitamente *`AF_INET`*.
`h_length`	Lunghezza dell'indirizzo in byte.

h_addr_list	Un array zero terminato d'indirizzi di rete per l'host. Gli indirizzo host sono in Network Byte Order.
h_addr	Il primo indirizzo in *h_addr_list*.

gethostbyname() ritorna un puntatore alla `struct hostent`, o NULL su errore. (*errno* non è impostato —bensì *h_errno*. Vedi **herror()**, sotto).

Come si usa? Talvolta (come si trova leggendo nei manuali di computer), esporre solo le informazioni al lettore non è abbastanza. Questa funzione è senz'altro più semplice da usare di quanto sembri.

Ecco un programma d'esempio:

```
/*
** getip.c – demo lookup di un hostname
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    struct hostent *h;

    if (argc != 2) {  // errore controlla la linea di comando
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) {  // ottiene le
informazioni sull'host
        herror("gethostbyname");
        exit(1);
    }

    printf("Host name  : %s\n", h->h_name);
    printf("Indirizzo IP : %s\n", inet_ntoa(*((struct in_addr *)h-
>h_addr)));

   return 0;
}
```

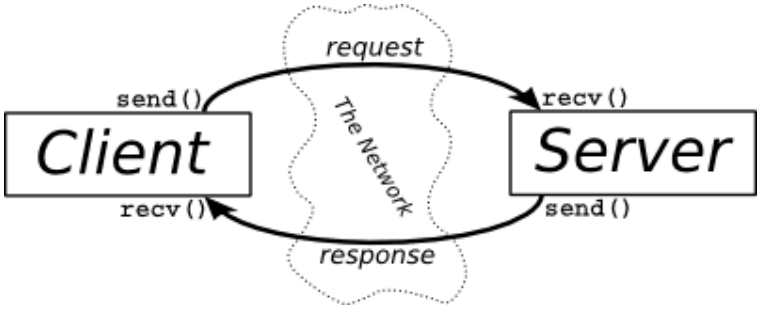
Con **gethostbyname()**, non puoi usare **perror()** per stampare i messaggi d'errore (dato che non è usato *errno*). Invece, chiama **herror()**.

E’ piuttosto diretto. Semplicemente passa la stringa che contiene il nome della macchina ("whitehouse.gov") a **`gethostbyname()`**, e poi prendi le informazioni da `struct hostent`.

L’unica possibile stranezza potrebbe esserci nello stampare l’indirizzo IP, come sopra. *`h->h_addr`* è un `char*`, ma **`inet_ntoa()`** vuole che gli sia passata una `struct in_addr`. Quindi faccio il cast di *`h->h_addr`* a `struct in_addr*`, poi lo dereferenzio per avere i dati.

5. Background Client-Server

E’ un mondo client-server, baby. Quasi tutto quello che c’è in rete ha a che fare con processi client che parlano a processi server e viceversa. Prendi **telnet**, per esempio. Quando ti connetti ad un host remoto sulla porta 23 con telnet (il client) un programma su quell’host (chiamato **telnetd**, il server) ritorna in vita. Gestisce la connessione telnet in ingresso, t’imposta un prompt di login, etc.



Interazione Client-Server.

Lo scambio d’informazioni client-server è riassunto nel diagramma di sopra.

Nota che la coppia client-server può parlare `SOCK_STREAM`, `SOCK_DGRAM`, o qualsiasi altro (fintanto che stanno dicendo la stessa cosa). Alcuni validi esempi di coppie di client-server sono **telnet**/**telnetd**, **ftp**/**ftpd**, oppure **bootp**/**bootpd**. Ogni volta che usi **ftp**, c’è un programma remoto, **ftpd**, che ti serve.

Spesso, ci sarà un solo server su quella macchina, e quel server maneggerà client multipli usando **`fork()`**. La routine di base è: il server aspetta una connessione, l’accetta (**`accept()`**), ed esegue una **`fork()`** con un processo figlio per gestirla. Questo è ciò che il nostro esempio di server fa nella prossima sezione.

5.1. Un semplice Stream Server

Tutto quello che fa questo server è inviare la stringa "`Hello, World!\n`" (“Ciao, Mondo!\n”) su una connessione stream. Hai bisogno solo di testarlo su una finestra, e collegarti via telnet da un’altra con:

```
$ telnet remotehostname 3490
```

dove `remotehostname` è il nome della macchina in cui lo stai eseguendo.

Il codice del server: (Nota: una barra rovesciata a fine riga significa che si continua nella prossima.)

```
/*
** server.c – demo di un socket stream server
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPORT 3490    // la porta in cui si connetteranno gli
utenti
#define BACKLOG 10     // quante connessioni pendenti terrà in coda

void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(void)
{
    int sockfd, new_fd;  // ascolta su sock_fd, nuova connessione
su new_fd
    struct sockaddr_in my_addr;    // informazioni sul mio
indirizzo
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         // host byte order
```

```
    my_addr.sin_port = htons(MYPORT);     // short, network byte
order
    my_addr.sin_addr.s_addr = INADDR_ANY; // lo riempie
automaticamente con il mio IP
    memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr)
== -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    sa.sa_handler = sigchld_handler; // elimina tutti i processi
morti
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        if ((new_fd = accept(sockfd, (struct sockaddr
*)&their_addr, \
                &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: ho una connessione da %s\n", \
            inet_ntoa(their_addr.sin_addr));
        if (!fork()) { // this is the child process
            close(sockfd); // il figlio non ha bisogno del'listener
            if (send(new_fd, "Ciao, Mondo!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // il genitore non ne ha bisogno
    }

    return 0;
}
```

Se sei curioso, ho il codice in un'unica grande funzione **`main()`**per chiarezza (penso) sintattica. Sentiti libero di suddividerla in piccole funzioni se ti fa sentire meglio.

(Poi, l'intera **`sigaction()`** potrebbe esserti nuova—questo è ok. Il codice che c'è là è responsabile dell'eliminazione dei processi zombie che restano quando termina il processo sottoposto a **`fork()`**. Se crei molti zombie e non li tagli il tuo amministratore di sistema di agiterà).

Puoi ottenere i dati da questo server usando il client elencato nella sezione seguente.

5.2. Un semplice Stream Client

Ragazzi, questo è perfino più semplice del server. Tutto quello cha fa il client è connettersi all'host specificato sulla linea di comando, porta 3490, ottenendo la stringa che il server invia.

Sorgente del client:

```
/*
** client.c – demo di un client stream socket
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490 // la porta in cui si connetterà

#define MAXDATASIZE 100 // il numero massimo di byte che otteniamo
in una volta

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }
    if ((he=gethostbyname(argv[1])) == NULL) {  // ottiene le
informazioni sull'host
        herror("gethostbyname");
        exit(1);
    }
```

```
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;    // host byte order
    their_addr.sin_port = htons(PORT);  // short, network byte
order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if (connect(sockfd, (struct sockaddr *)&their_addr,
                                          sizeof their_addr) == -1)
{
        perror("connect");
        exit(1);
    }

    if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("Ricevuti: %s",buf);

    close(sockfd);

    return 0;
}
```

Nota che se non esegui il server prima del client, **`connect()`** ritorna "Connection refused" ("Connessione rifiutata"). Molto utile.

5.3. Socket Datagram

Non ho veramente molto di cui parlare qui, così presenterò un paio di esempi: *`talker.c`* e *`listener.c`*.

listener si trova su una macchina aspettando un pacchetto entrante sulla porta 4950. **talker** invia un pacchetto a quella porta, sulla macchina specificata, che contiene qualsiasi cosa l'utente digita.

Ecco il *`sorgente di listener.c`*:

```
/*
** listener.c – demo di un server datagram socket
*/

#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 4950    // la porta a cui si collegheranno gli
utenti

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;    // informazioni sul mio
indirizzo
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette
    socklen_t addr_len;
    int numbytes;
    char buf[MAXBUFLEN];

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte
order
    my_addr.sin_addr.s_addr = INADDR_ANY; /* automaticamente
riempito con il mio IP*/
    memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof my_addr)
== -1) {
        perror("bind");
        exit(1);
    }

    addr_len = sizeof their_addr;
    if ((numbytes = recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
    printf("Il pacchetto è lungo %d byte \n",numbytes);
    buf[numbytes] = '\0';
```

```
    printf("Il pacchetto contiene \"%s\"\n",buf);

    close(sockfd);

    return 0;
}
```

Nota che nella nostra chiamata a socket**()** stiamo finalmente usando SOCK_DGRAM. Nota inoltre che non c'è bisogno di usare **listen()** o **accept()**. Questo è uno dei benefici usando datagram socket non connessi!

Ora viene il *sorgente di talker.c*:

```
/*
** talker.c – demo di una datagram "client"
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950    // la porta in cui si connetteranno gli
utenti

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // ottiene
informazioni sull'host
        herror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

```
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte
order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof their_addr)) ==
-1) {
        perror("sendto");
        exit(1);
    }

    printf("Inviati %d byte a %s\n", numbytes,
inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

Questo è quanto! Avvia **listener** su una macchina, poi esegui **talker** su un'altra. Guardali in un'eccitante comunicazione G-rated per l'intero nucleo familiare!

In passato ho menzionato molte volte un minuscolo dettaglio: datagram socket connessi. Ho bisogno di parlarne qui, poiché siamo nella sezione datagram di questo documento. Diciamo che **talker** chiama **`connect()`** e specifica l'indirizzo del **listener**. D'ora in poi, **talker** può inviare e ricevere solo da quell'indirizzo tramite **`connect()`**. Per questa ragione non necessita di usare **`sendto()`** e **`recvfrom()`**; puoi semplicemente usare **`send()`** e **`recv()`**.

6. Tecniche leggermente Avanzate

Non sono *veramente* avanzate, ma vanno oltre i livelli di base che abbiano già trattato. In effetti, se sei arrivato così lontano, dovresti considerarti abbastanza esperto sulle basi della programmazione di rete Unix! Congratulazioni!

Quindi eccoci qua nel mondo di alcune delle più esoteriche cose che potresti voler imparare sui socket. All'attacco!

6.1. Bloccaggio

Bloccaggio. Ne avrai sentito parlare—che diavolo è? In poche parole, "block" ("blocca") è gergo tecnico per "sleep" (Let.: dorme). Avrai probabilmente notato che quando esegui **listener**, sta lì finché arriva un pacchetto. Quello che succede è che viene chiamato `recvfrom()`, non c'erano dati, e così si dice che recvfrom() "si blocca" (cioè, è dormiente) fino a che arrivano i dati.

La maggior parte delle funzioni bloccano. **`accept()`** blocca. Tutte le funzioni **`recv()`** bloccano. Possono farlo perché gli è permesso. Quando in primo luogo crei il descrittore con **`socket()`**, il kernel lo imposta come bloccante. Se non vuoi che un socket sia bloccante, devi fare una chiamata a **`fcntl()`**:

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

Impostando un socket come non-bloccante, puoi in effetti "sondare" il socket per avere delle informazioni. Se cerchi di leggere da un socket non bloccante e non ci sono dati, se non gli è consentito di bloccarsi—restituirà `-1` ed *`errno`* sarà impostata a `EWOULDBLOCK`.

Parlando in generale, comunque, questo tipo di polling è una cattiva idea. Se poni il tuo programma in busy-wait (Lett.: attesa impegnata) cercando dati nel socket, succhierai tempo alla CPU inutilmente. Una soluzione più elegante per controllare se ci sono dati da leggere giunge nella sezione che segue su **`select()`**.

6.2. `select()`—Multiplexing Sincrono I/O

Questa funzione è alquanto strana, ma è molto utile. Considera la seguente situazione: sei un server e vuoi restare in ascolto di connessioni in ingresso così come continuare a leggere da connessioni che hai già.

Non c'è problema, tu dici, solo un **`accept()`** e un paio di **`recv()`**. Non così in fretta bellezza! Cosa succede se blocchi su una call **`accept()`**? Come riceverai (**`recv()`**) i dati allo stesso tempo? "Usa socket non bloccanti!" Non se ne parla! Non vuoi essere un divoratore di risorse della CPU. Cosa allora?

`select()` ti dà il potere di monitorare diversi socket allo stesso tempo. Ti dirà quali sono pronti per essere letti, quali per essere scritti, e quali hanno sollevato delle eccezioni, se vuoi veramente saperlo.

Senza ulteriori chiacchiere, ti offro il prototipo di **`select()`**:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int numfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

La funzione monitora “set” ("insiemi") di descrittori di file; in particolare *readfds*, *writefds*, ed *exceptfds*. Se vuoi vedere se puoi leggere dallo standard input e da qualche descrittore di socket, *sockfd*, aggiungi giusto il descrittore di file `0` e *sockfd* al set *readfds*. Il parametro *numfds* deve essere impostato al più alto valore del descrittore di file + uno. In questo esempio, deve essere impostato a *sockfd+1*, dato che è sicuramente più alto dello standard input (`0`).

Quando **`select()`** ritorna, *readfds* sarà modificato per riflettere quale dei descrittori che hai selezionato è pronto in lettura. Li puoi testare con la macro **`FD_ISSET()`**, di sotto.

Prima di andare avanti, ti parlerò di come manipolare questi insiemi. Ogni set è del tipo `fd_set`. Le macro che seguono operano su questo tipo:

`FD_SET(int fd, fd_set *set);`	Aggiunge il *fd* al *set*.
`FD_CLR(int fd, fd_set *set);`	Rimuove il *fd* dal *set*.
`FD_ISSET(int fd, fd_set *set);`	Restituisce vero se il *fd* è nel *set*.
`FD_ZERO(fd_set *set);`	Elimina tutti le voci dal *set*.

Infine, cos’è questa strana `struct timeval`? bene, qualche volta non vuoi aspettare per sempre che qualcuno t’invii dei dati. Forse ogni 96 secondi vuoi stampare "In corso..." al terminale anche se non è successo niente. Questa struttura *time* ti permette di specificare un timeout. Se il tempo è scaduto e **`select()`** non ha ancora trovato un descrittore di file in sola lettura, ritornerà cosicché potrai continuare le operazioni.

`struct timeval` ha i seguenti campi:

```
struct timeval {
    int tv_sec;     // secondi
    int tv_usec;    // microsecondi
};
```

Solo imposta *tv_sec* al numero di secondi d’attesa, e *tv_usec* al numero di microsecondi. Si, questi sono *micro*secondi, non millisecondi. Ci sono 1.000 microsecondi in un millisecondo, e 1.000 millisecondi in un secondo. Perciò, ci sono 1.000.000 di microsecondi in un secondo. Perché è "usec"? La "u" si suppone che assomigli alla lettera greca μ (Mu) che usiamo per "micro". Inoltre, quando la funzione ritorna, *timeout può* essere aggiornata per mostrare il tempo che ancora rimane. Questo dipende dal tipo di Unix che stai eseguendo.

Sì!! Abbiamo un timer con una risoluzione al microsecondo! beh, non contarci. Probabilmente dovrai attendere un po’ della tua *timeslice* Unix non importa quanto piccola è la tua `struct timeval`.

Altre cose d’interesse: se setti i campi in `struct timeval` a `0`, **`select()`** andrà immediatamente in timeout, eseguendo effettivamente il *polling* di tutti i descrittori del tuo set. Se imposti il parametro *timeout* a NULL, non andrà mai in timeout, e

aspetterai finché il primo descrittore di file è pronto. Infine, se non t'importa di attendere per un certo insieme, puoi impostarlo giusto a NULL nella call a **`select()`**.

Nel seguente frammento di codice si aspettano 2,5 secondi perché qualcosa appaia sullo standard input:

```
/*
** select.c – un demo di select()
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // descrittore di file per lo standard input

int main(void)
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    // non c'importa di writefds e exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("E' stato premuto un tasto!\n");
    else
        printf("Tempo scaduto.\n");

    return 0;
}
```

Se sei su un terminale bufferizzato, il tasto che premi deve essere seguito da INVIO (RETURN) o andrà comunque in timeout.

Ora, qualcuno di voi potrebbe pensare che questo è un bel modo di attendere dati su un datagram socket—e hai ragione: *potrebbe* esserlo. Alcuni Unix possono usare **`select()`** in questo modo, alcuni non possono. Dovesti vedere cosa dice la tua pagina man locale sulla questione.

Alcuni Unix aggiornano il tempo in `struct timeval` per rispecchiare l'ammontare di tempo che ancora rimane, prima di un timeout, ma altri No. Non contarci se vuoi essere portabile. (Usa **`gettimeofday()`** se vuoi tenere traccia del tempo trascorso. E' una disdetta, lo so, ma è così.)

Cosa succede se un socket nel set in lettura chiude la connessione? Beh, in quel caso, **`select()`** ritorna con tale descrittore di socket impostato come "pronto da leggere". Quando esegui effettivamente **`recv()`**, **`esso ritorna`** `0`. Questo è come sai che il socket ha chiuso la connessione.

Un'altra nota d'interesse in più su **`select()`**: se hai un socket in ascolto, puoi controllare per vedere se c'è una nuova connessione ponendo quel descrittore di socket nel set *`readfds`* set.

E questo, amici miei, è una veloce visione d'insieme della grande funzione **`select()`**.

Come da voi richiesto, ecco un esempio approfondito. Sfortunatamente, la differenza tra il banale esempio di sopra, e questo qua è significativa. Dai comunque un'occhiata, e leggi la descrizione che segue.

Questo programma agisce come un semplice chat server multi-utente. Fallo partire da una finestra, poi collegati via **telnet** ("**telnet hostname 9034**") da altre finestre multiple. Quando scrivi qualcosa in una sessione **telnet**, dovrebbe apparire in tutte le altre.

```
/*
** selectserver.c – un chat server multi-persona di basso livello
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 9034   // la porta su cui ascoltiamo

int main(void)
{
    fd_set master;   //descrittore della lista master
    fd_set read_fds; // descrittore della lista temp per select()
    struct sockaddr_in myaddr;     // indirizzo del server
    struct sockaddr_in remoteaddr; // indirizzo del client
    int fdmax;        // numero massimo di descrittori di file
    int listener;     // descrittore del socket in ascolto
    int newfd;        // nuovo socket descriptor accettato
    char buf[256];    // buffer per i dati del client
    int nbytes;
    int yes=1;        // per setsockopt() SO_REUSEADDR
    socklen_t addrlen;
    int i, j;
```

```
    FD_ZERO(&master);    // libera i set master e temp
    FD_ZERO(&read_fds);

    // get the listener
    if ((listener = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    /* evitiamo il fastidioso messaggio di errore "address already
in use" (indirizzo già in uso)*/
    if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, \
        sizeof(int)) == -1) {

        perror("setsockopt");
        exit(1);
    }

    // bind
    myaddr.sin_family = AF_INET;
    myaddr.sin_addr.s_addr = INADDR_ANY;
    myaddr.sin_port = htons(PORT);
    memset(myaddr.sin_zero, '\0', sizeof myaddr.sin_zero);
    if (bind(listener, (struct sockaddr *)&myaddr, sizeof myaddr)
== -1) {
        perror("bind");
        exit(1);
    }

    // ascolta
    if (listen(listener, 10) == -1) {
        perror("listen");
        exit(1);
    }

    // aggiunge il listener al set master
    FD_SET(listener, &master);

    // tiene traccia del più alto descrittore di file
    fdmax = listener; // so far, it's this one

    // loop principale
    for(;;) {
        read_fds = master; // lo copia
        if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }

        // cicla attraverso le connessioni esistenti cercando dati
da leggere
        for(i = 0; i <= fdmax; i++) {
```

```
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener) {
                    // gestisce nuove connessioni
                    addrlen = sizeof remoteaddr;
                    if ((newfd = accept(listener, \
                        (struct sockaddr *)&remoteaddr, &addrlen))
== -1) {

                        perror("accept");
                    } else {
                        FD_SET(newfd, &master); // aggiungi al set
master
                        if (newfd > fdmax) {    // tiene traccia
del massimo
                            fdmax = newfd;
                        }
                        printf("selectserver: nuova connessione da
%s sul " \
                            "socket %d\n", \
                            inet_ntoa(remoteaddr.sin_addr), newfd);
                    }
                } else {
                    // gestisce nuovi dati dal client
                    if ((nbytes = recv(i, buf, sizeof buf, 0)) <=
0) {
                        // abbiamo un errore o connessione chiusa
dal client
                        if (nbytes == 0) {
                            // connessione chiusa
                            printf("selectserver: il socket %d ha
interrotto la comunicazione\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // ciao!
                        FD_CLR(i, &master); // rimuovi dal master
set
                    } else {
                        // otteniamo qualche dato dal client
                        for(j = 0; j <= fdmax; j++) {
                            // inviamo a chiunque!
                            if (FD_ISSET(j, &master)) {
                                // eccetto il listener e noi stessi
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) ==
-1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // è così BRUTTO!
```

```
            }
        }
    }

    return 0;
}
```

Nota che ho impostato due descrittori di file nel codice: *`master`* e *`read_fds`*. Il primo, *`master`*, contiene tutti i descrittori che sono connessi in questo momento, così come il descrittore di socket che è in ascolto di nuove connessioni.

Il motivo per il quale ho settato *`master`* è che **`select(),`** in effetti, *modifica* il set che gli passi per rispecchiare i socket pronti da leggere. Poiché devo tenere traccia delle connessioni da una chiamata di **`select()`** alla successiva, devo memorizzarli in modo sicuro da qualche parte. All'ultimo minuto copio il *`master`* in *`read_fds`*, e poi chiamo **`select()`**.

Questo non significa che ogni volta ho una nuova connessione devo aggiungerla al set master? Si! E che ogni volta che si chiude una connessione, devo rimuoverla dal set *`master`*? Si.

Nota che controllo quando il socket di *`listener`* è pronto in lettura. Quando lo è, significa che ho una nuova connessione in sospeso, e la accetto (**`accept()`**) e la aggiungo all'insieme *`master`*. Similarmente, quando una connessione client è pronta per essere letta, e **`recv()`** restituisce `0`, so che il client ha chiuso la connessione, e devo rimuoverlo dal set *`master`*.

Nel caso che la **`recv()`**del client non ritorni zero, so che sono stati ricevuti dei dati. Quindi li prendo, e poi vado attraverso la lista *`master`* e invio i dati ai restanti dei client connessi.

Quella, amici miei, è una descrizione tutto fuorché semplice della straordinaria funzione **`select()`**.

In aggiunta, ecco un'altra informazione: c'è un'altra funzione di nome **`poll()`** che si comporta più o meno allo stesso modo di **`select()`**, ma con un sistema differente per maneggiare gli insiemi di descrittori. Testala!

6.3. Maneggiare `send()` parziali

Ti ricordi che nella **`sezione di send()`** quando ho detto che **`send()`** potrebbe non inviare tutti i dati che gli hai chiesto? In altre parole, vuoi che spedisca 512 byte, ma ne restituisce 412. Cosa è successo ai rimanenti 100 byte?

Beh, sono ancora nel tuo piccolo buffer aspettando di essere inviati. A causa di circostanze fuori del nostro controllo, il kernel ha deciso di non inviare tutti i dati in un colpo solo, e ora, amico mio, dipende da te far uscire i dati.

Puoi anche scrivere una funziona come questa per farlo:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // quanti byte abbiamo inviato
    int bytesleft = *len; // quanti ne rimangono da inviare
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // ritorna il numero effettivamente inviato

    return n==-1?-1:0; // restituisce -1 in caso di fallimento, 0
in caso di successo
}
```

In questo esempio, *`s`* è il socket a cui vuoi spedire i dati, *`buf`* è il buffer che contiene i dati, e *`len`* è un puntatore ad un `int` che contiene il numero di byte nel buffer.

La funzione restituisce `-1` su errore (ed *`errno`* è sempre impostata dopo la chiamata a **`send()`**). Inoltre, il numero di byte effettivamente inviati è in *`len`*. Questo sarà l'identico numero di byte che vuoi che spedisca, a meno che non c'è un errore. **`sendall()`** farà del suo meglio, soffiando e sbuffando, per inviare i dati, ma se c'è un errore ti riporta subito indietro.

Per completezza, ecco una semplice call alla funzione:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("Abbiamo inviato solo %d a causa dell'errore!\n", len);
}
```

Cosa accade all'altro capo quando arriva parte del pacchetto? Se i pacchetti sono di dimensioni variabili, come sa il destinatario quando finisce un pacchetto e ne inizia un altro? Si, il mondo reale è una vera e propria rottura. Forse devi *incapsulare* (ti ricordi di questo tempo addietro all'inizio nella sezione incapsulamento dati). Leggi là per i dettagli!

6.4. Serializzazione—Come impachettare i dati

E’ abbastanza semplice inviare dati testuali attraverso la rete, lo stai vedendo, ma cosa accade se tu vuoi inviare qualche dato "binario" come `int` o `float`? Ti risulta che hai poche opzioni.

1. Convertire il numero in testo con una funzione come **`sprintf()`**, quindi inviare il testo. Il ricevente riporterà il testo in numero usando una funzione come **`strtol()`**.
2. Invia solo i dati grezzi, passando un puntatore ai dati a **`send()`**.
3. Codificare il numero in una forma binaria portabile. Il ricevente lo decodificherà.

Sguardo furtivo! Solo stanotte!

[*Si alza il sipario*]

Beej dice, "preferisco il Metodo Tre!"

[*FINE*]

In effetti, hanno tutti i loro pro e contro, ma, come ho detto, in generale, preferisco il terzo metodo; ma parliamo prima di alcuni vantaggi e svantaggi degli altri due.

Il primo metodo, codificare i numeri in testo prima d’inviarli, ha il vantaggio che puoi facilmente inviare e leggere i dati in ingresso. Talvolta un protocollo leggibile dagli umani è eccellente da usare in una situazione di consumo non intensivo di banda, come ad esempio Internet Relay Chat (IRC). Ad ogni modo, ha lo svantaggio che è lenta da convertire, e risulta quasi sempre nell’occupazione di più spazio rispetto al numero originale!

Metodo due: inviare i dati grezzi. Questo è piuttosto semplice (ma pericoloso!): prendi giusto un puntatore ai dati da spedire, e chiama send() con esso.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* PERICOLO--non-portabile! */
```

Il ricevente ottiene qualcosa come:

```
double d;

recv(s, &d, sizeof d, 0);  /* PERICOLO--non-portabile! */
```

Veloce, semplice—cosa non va? Beh, risulta che non tutte le architetture rappresentano un `double` (o un `int`) allo stesso modo o persino nello stesso ordine di byte! Il codice è decisamente non portabile. (Ehi—forse non hai bisogno della portabilità, in questo caso è bello e veloce).

Quando impacchetti tipi interi, abbiamo già visto come la classe di funzioni **`htons()`**può aiutare a mantenere le cose portabili trasformando i numeri in Network

Byte Order, e questa è cosa la giusta da fare. Sfortunatamente, non ci sono simili funzioni per i tipi `float`. Ogni speranza è persa?

Temo di No! (Per un attimo hai avuto paura? No? Neanche un po'?). C'è qualcosa che possiamo fare: impacchettare (o eseguire il "marshaling", o la "serializzazione" dei dati, o uno tra i cento milioni di altri nomi) i dati in un formato binario noto che il ricevente può spacchettare dalla parte remota.

Cosa intendo per "noto formato binario"? Bene, abbiamo già visto l'esempio **`htons()`**, giusto? Cambia (o "codifica", se vuoi pensarla in quel modo) da qualsiasi formato l'host sia, in Network Byte Order. Per riconvertire (*unencode*) a numero, il ricevente chiama **`ntohs()`**.

Non avevo già finito dicendo che non esiste una funzione per numeri non interi? Sì. L'ho detto, e poiché non c'è un modo standard in C per fare questo, è un po' un pasticcio (che gioco di parole gratuita per te fan di Python).

La cosa da fare è impacchettare i dati in un formato noto e spedirli in rete per la decodifica. Per esempio, per impacchettare dei `float`, ecco qualcosa di veloce e sporco con ampio margine di miglioramento:

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // parte intera
e segno
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // frazione

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // parte intera
    f += (p&0xffff) / 65536.0f; // frazione

    if (((p>>31)&0x1) == 0x1) { f = -f; } // bit di segno impostato

    return f;
}
```

Il codice di cui sopra è una specie di ingenua implementazione che memorizza un `float` in un numero a 32 bit. Il bit alto (31) è usato per memorizzare il segno del numero ("1" significa negativo), e i successivi 7 bit (30-16) sono usati per

memorizzare la parte intera del `float`. Infine, i rimanenti bit (15-0) sono usati per memorizzare la parte frazionaria del numero.

L'uso è alquanto diretto:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // converte nella forma "network"
    f2 = ntohf(netf); // riconverte a test

    printf("Original: %f\n", f);        // 3.141593
    printf(" Network: 0x%08X\n", netf); // 0x0003243F
    printf("Unpacked: %f\n", f2);       // 3.141586

    return 0;
}
```

Dal lato positivo, è piccolo, semplice, e veloce. Dal lato negativo, non è un modo efficiente di usare lo spazio e il range è pesantemente ristretto—prova a memorizzare un numero più grande di 32767 e non ne sarà molto felice! Nell'esempio di cui sopra puoi anche vedere che l'ultime due cifre decimali non sono correttamente preservate.

Cosa possiamo fare invece? Beh, *lo* Standard per memorizzare numeri in virgola mobile è noto come IEEE-754. La maggior parte dei computer usano questo formato internamente per la matematica con i float, quindi in questi casi, strettamente parlando, non ci sarebbe bisogno di una conversione, ma se vuoi che il tuo codice sorgente sia portabile questa è una supposizione che di conseguenza non puoi fare.

Ecco del codice di codifica float e double nel formato IEEE-754 (Per lo più—non codifica NaN o Infinito, ma può essere modificato per farlo).

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

long long pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;
    unsigned significandbits = bits - expbits - 1; // -1 per il bit
di segno

    if (f == 0.0) return 0; // levati di mezzo questo caso speciale
```

```
    // controlla il segno e inizia la normalizzazione
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // Ottiene la forma normalizzata di f e traccia l'esponente
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // calcola la forma binaria (non float) dei dati significand
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // ottiene l'esponente con bias
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // restituisce il risultato finale
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) |
significand;
}

long double unpack754(long long i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1; // -1 per il bit
di segno

    if (i == 0) return 0.0;

    // estrae il significand
    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // riconverte a float
    result += 1.0f; // riaggiunge 1

    // si occupa dell'esponente
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // gli dà il segno
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

Ho messo qualche macro a mano là sopra per impacchettare e spacchettare numeri a 32-bit (probabilmente un `float`) e a 64-bit (probabilmente un `double`), ma la funzione **`pack754()`** può essere chiamata direttamente per dirgli di codificare dati del

valore di *bits* (*expbits* i quali sono riservati per gli esponenti dei numeri normalizzati.)

Ecco un codice d'esempio:

```
#include <stdio.h>
#include <stdint.h> // definisce tipi uintN_t

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);

    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float prima : %.7f\n", f);
    printf("float codificato: 0x%08X\n", fi);
    printf("float dopo  : %.7f\n\n", f2);

    printf("double prima : %.20lf\n", d);
    printf("double codificato: 0x%016llX\n", di);
    printf("double dopo  : %.20lf\n", d2);

    return 0;
}
```

Il codice di sopra produce il seguente output:

```
float prima : 3.1415925
float codificato: 0x40490FDA
float dopo  : 3.1415925

double prima : 3.14159265358979311600
double codificato: 0x400921FB54442D18
double dopo  : 3.14159265358979311600
```

Un'altra domanda che potresti avere è come impacchetti `struct`? Sfortunatamente per te, il compilatore è libero di aggiungere della roba dappertutto nella `struct`, e questo vuol dire che non puoi inviare in modo portabile l'intera cosa in una volta sola. (Non sei stufo di sentire "non posso fare questo", "non posso fare quello"? Mi spiace! Per citare un amico, "Ogni qualvolta qualcosa va male, incolpo sempre Microsoft." Questo potrebbe non essere colpa di Microsoft, esplicitamente, ma l'affermazione del mio amico è verissima).

Torniamo al punto: il modo migliore di spedire una `struct` in rete è impacchettare ogni campo in modo indipendente e poi spacchettarli in una `struct` quando arrivano dall'altro parte.

E' un sacco di lavoro, è quello che stai pensando. Sì, lo è. Una cosa che puoi fare è scrivere una funzione che ti aiuti ad impacchettare i dati al tuo posto. Sarà uno spasso! Davvero!

Nel libro "The Practice of Programming" di Kernighan e Pike, s'implementano funzioni simili a **`printf()`**chiamate **`pack()`** e **`unpack()`**che fanno esattamente questo. Ne farò il link, ma apparentemente queste funzioni non sono in linea con il resto dei sorgenti del libro.

(*The Practice of Programming* è un'ottima lettura. Zeus salva un gattino ogni volta che lo raccomando).

A questo punto, scriverò qualcosa sulle API C Typed Parameter Language con licenza BSD, che non ho mai usato, ma che sembrano di tutto rispetto. Programmatori Python e Perl vorranno controllare le loro funzioni **`pack()`** e **`unpack()`** per ottenere la stessa cosa, e Java ha la cara vecchia interfaccia *Serializable* che può essere usata in modo simile.

Ma se vuoi scrivere la tua utilità d'impacchettamento in C, il trucco di K&P's è quello di usare una lista ad argomenti variabili per sviluppare funzioni come **`printf()`**per costruire dei pacchetti. Ecco una versione che ho inventato io stesso che spero basti per darti un'idea di come può funzionare una tale cosa.

(Questo codice fa riferimento alla funzione **`pack754()`** poc'anzi. Le funzioni **`packi*()`** lavorano come la familiare **`htons()`**, eccetto che impacchettano array di char invece che altri interi).

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>

/*
** packi16() – memorizzano un int a 16 bit in un buffer char (come
htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
    *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() – memorizza un int a 32 bit in un buffer char (come
htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
    *buf++ = i>>24; *buf++ = i>>16;
```

```
    *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() – spacchetta un int a 16 bit da un buffer char (come
ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
    return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() – spacchetta un int a 32 bit da un buffer char (come
ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
    return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}

/*
** pack() -- immagazzina i dati imposti dal formato stringa nel
buffer
**
**  h - 16-bit              l - 32-bit
**  c - 8-bit char          f - float, 32-bit
**  s - string (Viene automaticamente aggiunta una lunghezza di 16-
bit)
*/
size_t pack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int h;
    int l;
    char c;
    float f;
    char *s;
    size_t size = 0, len;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            size += 2;
            h = va_arg(ap, int); // promosso
            packi16(buf, h);
            buf += 2;
            break;

        case 'l': // 32-bit
```

```
            size += 4;
            l = va_arg(ap, int);
            packi32(buf, l);
            buf += 4;
            break;

        case 'c': // 8-bit
            size += 1;
            c = va_arg(ap, int); // promosso
            *buf++ = (c>>0)&0xff;
            break;

        case 'f': // float
            size += 4;
            f = va_arg(ap, double); // promosso
            l = pack754_32(f); // converti a IEEE 754
            packi32(buf, l);
            buf += 4;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }

    va_end(ap);

    return size;
}

/*
** unpack() – spacchetta i dati imposti dal formato stringa nel
buffer
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    short *h;
    int *l;
    int pf;
    char *c;
    float *f;
    char *s;
    size_t len, count, maxstrlen=0;
```

```
    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            h = va_arg(ap, short*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'l': // 32-bit
            l = va_arg(ap, int*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'c': // 8-bit
            c = va_arg(ap, char*);
            *c = *buf++;
            break;

        case 'f': // float
            f = va_arg(ap, float*);
            pf = unpacki32(buf);
            buf += 4;
            *f = unpack754_32(pf);
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = unpacki16(buf);
            buf += 2;
            if (maxstrlen > 0 && len > maxstrlen) count = maxstrlen
- 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // traccia max lung str
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

Ecco un programma dimostrativo del codice di sopra che impacchetta dei dati in *buf* e gli spacchetta in variabili. Nota che quando chiami **unpack()** con un argomento stringa (specificatore di formato "s"), è saggio indicare una lunghezza massima per evitare un *buffer overrun*, es. "96s". Sii guardingo quando spacchetti dati che ottieni dalla rete—un utente malizioso potrebbe inviare pacchetti malformati nello sforzo di attaccare il tuo sistema!

```
#include <stdio.h>

int main(void)
{
    unsigned char buf[1024];
    char magic;
    short monkeycount;
    long altitude;
    float absurdityfactor;
    char *s = "Grande forte Zot!  Hai trovato la Runa Magica!";
    char s2[96];
    size_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", 'B', 0, 37, -5, s, -
3490.6677);
    packi16(buf+1, packetsize); // memorizza la dimensione del
pacchetto nel pacchetto per divertimento

    printf("Pacchetto ha %d byte\n", packetsize);

    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude,
s2,
        &absurdityfactor);

    printf("'%c' %d %d %ld \"%s\" %f\n", magic, ps2, monkeycount,
altitude,
        s2, absurdityfactor);

    return 0;
}
```

Sia che fai girare il tuo codice che quello di qualcun altro, è buona idea avere un insieme generale di routine per l'impacchettamento dei dati per mantenere i bug sotto controllo, piuttosto che impacchettare ogni bit a mano ogni volta.

Quando impacchetti i dati, quale è un buon formato da usare? Ottima domanda. Per fortuna, RFC 4506, lo standard per la Rappresentazione Esterna dei Dati, definisce già dei formati binari per differenti tipi, come numeri in virgola mobile, interi, array, dati grezzi, etc. Ti consiglio di conformarti a questi se li userai tu stesso. Ma non sei obbligato. La "Polizia del Pacchetto" non è proprio fuori dalla tua porta. Almeno, non *penso* lo sia.

In ogni caso, codificare i dati in un modo o nell'altro prima d'inviarli è la cosa giusta da fare!

6.5. Figlio dell'incapsulamento dei Dati

Cosa veramente significa incapsulare i dati, ad ogni modo? Nel più semplice dei casi, significa che tu c'infili un header anche assieme a qualche informazione identificativa o alla lunghezza del pacchetto, o entrambe.

Come dovrebbe apparire un'intestazione? Beh, è giusto un qualche dato binario che rappresenta qualunque cosa tu ritieni necessaria per completare il tuo progetto.

Wow. Questo è vago.

Okay. Ad esempio, diciamo che hai un programma di chat multiutente che usa dei `SOCK_STREAM`. Quando un utente digita ("dice") qualcosa, due pezzi d'informazione hanno bisogno di essere trasmessi al server: cosa è stato detto e chi l'ha detto.

Fin qui ok? "Qual'è il problema?" ti chiedi.

Il problema è che i messaggi possono essere di lunghezza variabile. Una persona di nome "tom" potrebbe dire, "Hi" ("Ciao"), e un'altra persona chiamata "Benjamin" potrebbe dire, "Hey guys what is up?" ("Ciao ragazzi che succede").

Così tu invii (**`send()`**)tutta questa roba ai client come ti arriva. Il tuo flusso di dati in uscita appare come esegue:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

E così via. Come sa il client quando messaggio inizia e un altro termina? Potresti, se lo volessi, formare dei messaggi tutti della stessa lunghezza e chiamare proprio **`sendall()`** che abbiamo implementato prima, ma che spreco di banda! Non vogliamo inviare 1024 byte affinché "tom" possa dire "Hi".

Così *incapsuliamo* i dati in un piccolo header e struttura di pacchetto. Sia il client che il server sanno come fare il packing e l'unpacking di questi dati (talvolta ci si riferisce a essi con "marshal" e "unmarshal"). Ora non lo vedi, ma stiamo iniziando a definire un protocollo che descrive come comunicano un server ed un client!

In questo caso, supponiamo che il nome utente sia di una lunghezza fissa di 8 caratteri, che termina con `'\0'`, e che la lunghezza dei dati vari, fino ad un massimo di 128 caratteri. Diamo un'occhiata ad un esempio di struttura di pacchetto che potremo usare in questa situazione:

1. `len` (1 byte, unsigned)—la lunghezza totale del pacchetto, che include gli 8-byte dello user name e dei dati della chat.
2. `name` (8 bytes)—Il nome utente, termianto da NULL se necessario.
3. `chatdata` (*n*-byte)—Il dato in se stesso, non più di 128 byte. La lunghezza del pacchetto va calcolata come la lunghezza di questi dati più 8 (la dimensione del campo nome di sopra).

Perché ho scelto i limiti 8-byte e 128-byte per i campi? Li ho tirati fuori considerando che siano abbastanza lunghi, forse, sebbene 8 byte sia troppo restrittivo per le tue

necessità, e potresti avere un campo nome di 30 byte, o qualunque altro. La scelta è tua.

Usando la definizione di pacchetto di sopra, il primo pacchetto sarà composto dalle seguenti informazioni (in hex e ASCII):

```
   0A     74 6F 6D 00 00 00 00 00      48 69
(lunghezza)  T  o  m    (padding)         H  i
```

E il secondo è simile:

```
   18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20
77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w
...
```

(la lunghezza è memorizzata in Network Byte Order, naturalmente. In questo caso è solo un byte quindi non ha importanza, ma parlando in generale vorrai che tutti i tuoi interi binari siano memorizzati in Network Byte Order).

Quando invii questi dati, dovresti essere sicuro e usare un comando simile a **`sendall()`**, così sai che tutti i dati sono inviati, persino se ci vogliono chiamate multiple a **`send()`** per farli uscire tutti.

Ugualmente quando stai ricevendo questi dati, devi fare un po' di lavoro extra. Per essere sicuro dovresti aspettarti di ricevere un pacchetto parziale (come forse riceviamo "`18 42 65 6E 6A`" da Benjamin, ma è tutto quello che otteniamo in questa chiamata a **`recv()`**). Abbiamo necessità di chiamare **`recv()`** ancora e ancora di nuovo, fino a che il pacchetto è completamente ricevuto.

Ma in che modo? Beh, consociamo il numero di byte totali che abbiamo bisogno di ricevere perché il pacchetto sia completo, dato che quel numero è contrassegnato nel pacchetto stesso. Sappiamo anche che la massima dimensione del pacchetto è 1+8+128, o 137 byte (perché è così che lo abbiamo definito).

Ci sono in effetti un paio di cose che puoi fare qui. Poiché sai che ogni pacchetto inizia con la lunghezza, puoi fare una chiamata a **`recv()`** giusto per conoscerne il valore. Dopo di che puoi richiamarlo specificando esattamente la lunghezza rimanente del pacchetto (possibilmente ripetutamente per ottenere tutti i dati), finché non hai il pacchetto completo. Il vantaggio di questo metodo è che necessiti solo di un buffer grande abbastanza per un pacchetto, mentre lo svantaggio è che hai bisogno di chiamare **`recv()`** almeno due volte per avere tutti i dati.

Un'altra opzione è quella di chiamare solo **`recv()`** e dire che l'ammontare che sei disposto a ricevere è il massimo numero di byte in un pacchetto. Poi qualunque cosa hai, cacciala nel buffer, e finalmente controlla se è completo. Ovviamente, potresti ottenere qualcosa del prossimo pacchetto per cui avrai bisogno di spazio per quello.

Ciò che puoi fare è dichiarare un array grande abbastanza per due pacchetti. Questo è il tuo array di lavoro dove ricostruisci i pacchetti come arrivano.

Ogni volta che ricevi dati, gli aggiungerai al buffer di lavoro e controllerai che il buffer sia completo. Cioè che il numero di byte in esso sia più grande o uguale alla lunghezza specificata nell'intestazione (+1, perché la lunghezza nell'header non include il byte della lunghezza in se stessa). Se il numero di byte nel buffer è inferiore a 1, il pacchetto non è completo, ovviamente. Devi creare un caso speciale per questo, anche se, dato che il primo è garbage (lett.: spazzatura) e non puoi fidarti di esso per la corretta lunghezza del pacchetto.

Una volta che il pacchetto è completo, puoi farci quello che vuoi. Usalo, e rimuovilo dal buffer di lavoro.

Okay! Ci stai ancora giocherellando nella tua testa? Bene, ecco la seconda della battuta uno-due: potresti aver letto oltre la fine di un pacchetto e nel successivo pacchetto in una singola call a **`recv()`**. Vale a dire, hai un buffer di lavoro con un pacchetto completo, e una parte incompleta del prossimo pacchetto! Che diamine. (Ma questo è perché hai reso il tuo buffer di lavoro grande abbastanza per contenere due pacchetti—nel caso questo succedesse!)

Poiché conosci la lunghezza del pacchetto, dall'intestazione, e hai tenuto traccia del numero di byte nel tuo buffer, puoi sottrarre e calcolare quanti byte nel buffer di lavoro appartengono al secondo (incompleto) pacchetto. Quando hai maneggiato il primo, puoi eliminarlo dal buffer e muovere il secondo pacchetto parziale all'inizio del buffer, così è tutto pronto per la prossima **`recv()`**.

(Alcuni di voi lettori noteranno che effettivamente spostare il pacchetto parziale al principio del buffer di lavoro prende tempo, e il programma può essere scritto per non richiederlo usando un buffer circolare. Sfortunatamente per il resto di voi, una discussione sui buffer circolari è oltre gli scopi di questo articolo. Se sei ancora curioso, afferra un libro sulle strutture dati e inizia da lì).

Non ho mai detto che era facile. Ok, ho detto che era facile. E lo è; hai bisogno solo di pratica e quasi subito ti sarà naturale. Lo giuro su Excalibur!

6.6. Pacchetti Broadcast—Ciao, Mondo!

Finora, questa guida ha parlato di spedire dati da un host ad un altro host, ma è possibile, insisto, è possibile, avendone la facoltà, inviare dati a host multipli *allo stesso tempo*!

Con UDP (solo UDP, non TCP) e IPv4 standard, questo viene fatto attraverso un meccanismo noto come *broadcasting*. Con IPv6 (che non appare ancora in questa guida…), il broadcasting non è supportato, e devi fare ricorso alla tecnica, spesso superiore, del *multicasting*; ma è un futuro abbastanza irrealistico—siamo fermi in un presente a 32-bit.

Ma aspetta! Non puoi uscire di corsa e iniziare a fare del broadcasting a caso; devi impostare l'opzione socket **`SO_BROADCAST`** prima che tu possa spedire un pacchetto di broadcast in rete. E' come una di questi piccoli mantelli di plastica che mettono sopra l'interruttore di lancio del missile! Questo è giusto quanto potere avete nelle vostre mani!

Ma seriamente, sebbene ci sia del pericolo nell'usare pacchetti di braodcast, cioè: ogni sistema che riceve un pacchetto di broadcast deve disfare tutti i layer (N.d.T: strati) a buccia di cipolla (d'incapsulamento dati) fino a che non trova a quale porta sono destinati i dati. Poi o li consegna o li scarta. In entrambi i casi, è un mucchio di lavoro per ogni macchina che riceve il pacchetto di broadcast, e dato che è lo stesso per tutte le macchine della rete locale, ci possono essere un sacco di computer che fanno del lavoro non necessario. Quando il gioco Doom usci per primo, questa era il reclamo riguardo al codice di rete.

Sì, ho detto la rete locale. C'è più di un modo di spellare un gatto... aspetta un minuto. C'è veramente più di un modo di spellare un gatto? Che razza di espressione è questa? Uh, e ugualmente, c'è più di un modo per inviare un pacchetto di broadcast, ma tali pacchetti saranno di solito limitati alla tua rete locale non importa come lo invii.

Perciò adesso veniamo alle basi dell'intera cosa: come specifichi l'indirizzo di destinazione per un messaggio broadcast? Ci sono due comuni metodi.

1. Invia i dati al tuo indirizzo di broadcast. Questo è il tuo indirizzo di rete con tutti i bit settati a 1 per la parte host dell'indirizzo. Ad esempio, a casa la mia rete è 192.168.1.0, la mia netmask (maschera di rete) è 255.255.255.0, quindi l'ultimo byte dell'indirizzo è il numero di host (poiché i primi tre byte, secondo la netmask, sono il numero di rete). Per cui il mio indirizzo di broadcast è 192.168.1.255. Sotto Unix, il comando **ifconfig** ti darà effettivamente tutti questi dati. (Se sei curioso, la logica bit a bit per avere il tuo indirizzo di broadcast è *`numero_di_rete`* OR (NOT *`maschera_di_rete`*)).
2. Invia i dati all'indirizzo "globale" di broadcast. Che è 255.255.255.255, anche noto come `INADDR_BROADCAST`. Molte macchine automaticamente faranno un AND bit a bit con il tuo numero di rete per convertirlo in un indirizzo di rete di broadcast, ma alcune No. Questo varia.

Quindi cosa succede se cerchi d'inviare dei dati ad un indirizzo di broadcast senza prima settare l'opzione socket `SO_BROADCAST`? Bene, accendiamo i cari vecchi **talker e listener** e vediamo che accade.

```
$ talker 192.168.1.2 foo
inviati 3 byte a 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied (N.d.T: permesso negato)
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Sì, non è per niente bello...perché non abbiamo impostato l'opzione `SO_BROADCAST`. Fallo, e ora puoi inviare **`sendto()`** dovunque vuoi!

Di fatto, questa è *l'unica differenza* tra un'applicazione UDP che può fare il broadcast e una che non può. Per cui prendiamo la vecchia applicazione **talker** e aggiungiamo una sezione che setta l'opzione socket `SO_BROADCAST`. Chiameremo questo programma *`broadcaster.c`*:

```
/*
** broadcaster.c -- un "client" datagram come talker.c, eccetto che
**                  questo può fare broadcasting
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950    // la porta in cui si connetteranno gli
utenti

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // informazioni sull'indirizzo
di chi si connette
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // se ciò non funziona prova questo

    if (argc != 3) {
        fprintf(stderr,"usage: <hostname di broadcast>
<messaggio>\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // ottieni le info
dell'host
        herror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // questa chiamata fa la differenza tra questo programma e
talker.c:
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
```

```
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte
order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof their_addr)) ==
-1) {
        perror("sendto");
        exit(1);
    }

    printf("Inviati %d byte a %s\n", numbytes,
inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

Qual'è la differenza tra questo e una "normale" situazione UDP client/server? Nessuna! (Ad eccezione del client a cui è consentito d'inviare pacchetti di broadcast in questo caso). Vai avanti ed esegui il vecchio programma UDP **listener** in una finestra, e **broadcaster** in un'altra. Dovresti ora essere in grado di effettuare tutti questi invii che erano falliti.

```
$ talker 192.168.1.2 foo
inviati 3 byte a 192.168.1.2
$ talker 192.168.1.255 foo
inviati 3 byte a 192.168.1.255
$ talker 255.255.255.255 foo
inviati 3 byte a 255.255.255.255
```

E dovresti vedere **listener** rispondere che riceve i pacchetti.

Bene, questo è eccitante, ma ora accendiamo **listener** su un'altra macchina vicina a te nella stessa rete cosicché hai due copie in esecuzione, una su ogni macchina, ed avvia di nuovo **broadcaster** con il tuo indirizzo di broadcast... Ehi! Entrambi i **listener** ricevono i pacchetti sebbene tu hai chiamato **`sendto()`** una volta sola! Figo!

Se il **listener** riceve i dati che gli hai inviato direttamente, ma non i dati dell'indirizzo di broadcast, potrebbe essere che il firewall della tua macchina locale sta bloccando i pacchetti. (Sì, Pat e Bapper, grazie per esservene accorti prima di me del perché il mio codice d'esempio non funzionava. Ti avevo detto che vi menzionavo nella guida, ed eccovi qui. Quindi *nyah*.)

Ancora, sii prudente con i pacchetti di broadcast. Poiché ogni macchina della LAN che sarà forzata ad occuparsene sia con **`recvfrom()`** che No, sarà un carico per l'intera rete. Sono definitivamente da usare in modo appropriato e con moderazione.

7. Domande Comuni

Dove trovo questi file header?

Se non li hai già nel tuo sistema, probabilmente non ne hai bisogno. Controlla il manuale relativo alla tua piattaforma. Se stai sviluppando per Windows, hai solo bisogno di `#include <winsock.h>`.

Cosa faccio quando `bind()` riporta "Indirizzo già in uso"?

Devi usare **`setsockopt()`** con l'opzione **`SO_REUSEADDR`** sul socket in ascolto. Controlla la sezione su **`bind()`** e la su **`select()`** per un esempio.

Come ottengo una lista dei socket aperti nel sistema?

Usa **netstat**. Controlla le pagine **man** per i dettagli completi, ma dovresti ottenere un buon output solo digitando:

```
$ netstat
```

L'unico trucco è determinare quale socket è associato a quel programma. `:-)`

Come posso visualizzare la tabella d'instradamento?

Esegui il comando **route** (in *`/sbin`* nella maggior parte dei Linux) o il comando **netstat -r**.

Come posso eseguire i programmi client e server se ho solo un computer? Non ho bisogno di una rete per scrivere programmi di rete?

Fortunatamente per te, quasi tutte le macchine implementano un "dispositivo di loopback " che si trova nel kernel e fa finta di essere una scheda di rete. (Questa è l'interfaccia che viene elencata come "`lo`" nella tabella di routing).

Facciamo finta che ti sei collegato ad una macchina di nome "`capra`". Esegui il client su una finestra e il server in un altro, o fai partire il server in background ("**server &**") e avvia il client nella stessa finestra. Il risultato del dispositivo di loopback è che puoi sia fare **client capra** oppure **client localhost** (dato che "`localhost`" è probabilmente definito nel tuo file *`/etc/hosts`*) e avrai un client che parla al server senza un network!

In breve, non sono necessari cambiamenti al codice per farli girare in una singola macchina senza rete! Hurrah!

Come posso dire che la parte remota ha chiuso la connessione?

Puoi dirlo perché **`recv()`** restituirà `0`.

Come implemento una utility "ping"? Cos'è ICMP? Come posso scoprire di più sulle raw socket e `SOCK_RAW`?

Tutte le tue domande sulle raw socket avranno una risposta nel libro di programmazione di W. Richard Stevens' *UNIX Network*. Vedi la sezione libri di questa guida.

Come sviluppo in Windows?

Primo, cancella Windows e installa Linux o BSD. `};-)`. No, in effetti, guarda solo la sezione per sviluppare in Windows nell'introduzione.

Come sviluppo in Solaris/SunOS? Continuo ad avere errori del linker quando cerco di compilare!

Gli errori del linker si verificano perché i box Sun non compilando automaticamente le librerie socket. Vedi la sezione sullo sviluppo in Solaris/SunOS all'inizio.

Perché `select()` continua a cadere su un segnale?

I segnali tendono a far si che chiamate di sistema bloccate ritornino `-1` con *`errno`* settato a `EINTR`. Quando imposti un handler di un segnale con **`sigaction()`**, puoi settare il flag `SA_RESTART`, che si suppone riavvi la system call dopo che è stata interrotta.

Naturalmente, questo non sempre funziona.

La mia soluzione preferita a questo comporta un'istruzione di `goto`. Sai che questo irrita i professori come non mai, quindi andiamo!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -
1) {
    if (errno == EINTR) {
        // qualche segnale ci ha proprio interrotti, allora
riavvia
        goto select_restart;
    }
    // maneggia il vero errore qui:
    perror("select");
}
```

Certo, non hai *bisogno* di usare `goto` in questo caso; puoi usare altre strutture, ma penso che l'istruzione `goto` sia effettivamente più chiara.

Come posso implementare un timeout su una call a `recv()`?

Usa **`select()`**! Ti consente di specificare un parametro di timeout per i descrittori di socket che stai per leggere, o potresti racchiudere l'intera funzionalità in una singola funzione, come questa:

```
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;

    // imposta il set del descrittore
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // imposta il valore di timeout nella struct
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // aspetta fino al timeout o alla ricezione dei dati
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
    if (n == -1) return -1; // error

    // i dati devono essere qui, quindi fai una normale
recv()
    return recv(s, buf, len, 0);
}
.
.
.
// Semplice chiamata a recvtimeout():
n = recvtimeout(s, buf, sizeof buf, 10); // 10 second timeout

if (n == -1) {
    // si è verificato un errore
    perror("recvtimeout");
}
else if (n == -2) {
    // si è verificato il timeout
} else {
    // ho ottenuto qualche dato nel buffer
}
.
.
.
```

Nota che **`recvtimeout()`** ritorna `-2` in caso di un timeout. Perché non restituisce `0`? Beh, se ti ricordi, un valore di ritorno di `0` su una chiamata a **`recv()`** signfica che la parte remota ha chiuso la connessione. Cosìcchè il

valore parla già di per sé, e `-1` significa "errore", quindi scelgo `-2` come mio indicatore di timeout.

Come cripto o comprimo i dati prima che siano spediti attraverso il socket?

Una maniera semplice per crittografare è usare SSL (secure sockets layer), ma questo è al dilà degli scopi di questa guida. (Controlla Progetto OpenSSL per maggiori informazioni).

Ipotizzando però che tu voglia inserire o implementare il tuo sistema di crittografia o di archiviazione, è solo un problema di pensare che i tuoi dati viaggiano attraverso due estremità, in una sequenza di passi. Ciascun gradino modifica i dati in qualche modo.

1. il server legge i dati da un file (o da qualsiasi parte)
2. il server cripta/comprime i dati (tu aggiungi questa parte)
3. il server `invia` i dati criptati

Ora nel modo opposto:

1. il client riceve i dati crittografati
2. il client decripta/decomprime i dati (tu aggiungi questa parte)
3. il client scrive i dati nel file (o da qualsiasi parte)

Se comprimerai e crittograferai, ricordati solo di comprimere prima. `:-)`

Fintanto che il client disfa quello fa il server, alla fine i dati saranno a posto non importa quanti passi intermedi aggiungi.

Quindi tutto quello che hai bisogno di fare per usare il mio codice è trovare lo spazio tra i dati che sono letti e i dati che sono inviati in rete, e metterrci un po' di codice là che crittografi.

Cos'è questo "`PF_INET`" che continuo a vedere? E' in relazione a `AF_INET`?

Sì, lo è. Vedi **`la sezione sui socket`** per i dettagli.

Come posso scrivere un server che accetta comandi shell da un client e gli esegua?

Per semplicità, diciamo che il client esegue delle **`connect()`**, **`send()`**, e delle **`close()`** (cioè, non ci sono successive chiamate di sistema senza che il client si connetta di nuovo).

Il procedimento che il client esegue è questo:

1. si connette (**`connect()`**) ad un server
2. invia i dati (**`send("/sbin/ls > /tmp/client.out")`**)
3. chiude (**`close()`**) la connessione

Nel frattempo, il server maneggia i dati e gli esegue:

1. **`accept()`**la connessione dal client
2. **`recv(str)`**la stringa di comando
3. **`close()`** chiude la connessione
4. **`system(str)`** esegue il comando

Attento! Avere il server che esegue quello che gli dice il client è come dare accesso ad una shell remota e le persone possono fare delle cose con il tuo account quando si connettono al server. Ad esempio, cosa succede se il client invia "**rm -rf ~**"? Cancella ogni cosa del tuo account, questo è quanto!

Quindi sii prudente, e impedisci ai client di usare tutto eccetto un paio di utility che sai che sono sicure, come **foobar**:

```
if (!strncmp(str, "foobar", 6)) {
    sprintf(sysstr, "%s > /tmp/server.out", str);
    system(sysstr);
}
```

ma sei ancora non sicuro, sfortunatamente: cosa accade se il client inserisce "**foobar; rm -rf ~**"? La cosa più sicura è scrivere una piccola routine che inserisce un carattere di escape ("\") davanti a tutti i caratteri non alfanumerici (inclusi gli spazi) negli argomenti del comando.

Come puoi vedere, la sicurezza è un problema abbastanza grande quando inizi ad eseguire cose che il client invia.

Sto inviando una gran quantità di dati, ma quando eseguo `recv()`, riceve solo 536 o 1460 byte alla volta. Ma se è in esecuzione nella mia macchina locale, riceve tutti i dati in una volta sola. Cosa sta succedendo?

Stai raggiungendo l'MTU—la dimensione massima che il dispositivo fisico può gestire. Sulla macchina locale stai usando il dispositivo di loopback che può occuparsi di 8K o più senza problemi. Ma su Ethernet, che può solo gestire 1500 byte con un header, hai toccato il limite. Su un modem con 576 MTU (di nuovo, con l'intestazione), raggiungi persino il limite inferiore.

Ti devi assicurare che tutti i dati siano spediti, prima di tutto. (Vedi l'implementazione della funzione **`sendall()`** per i dettagli). Una volta che sei sicuro di quello, allora hai bisogno di chiamare **`recv()`** ciclicamente fino che non sono ricevuti tutti i dati.

Leggi la sezione Figlio dell'Incapsulamento dei Dati per dettagli sul ricevere pacchetti completi usando call multiple a **`recv()`**.

Sono su un box Windows e non ho la system call `fork()` o nessun tipo di struct `sigaction`. Che fare?

Sono dovunque, saranno in librerie POSIX ricevute assieme al tuo compilatore. Poiché non ho una Windows box, non posso verametne dirti la risposta, ma mi sembra di ricordarmi che Microsoft ha uno strato di

compatibilità POSIX e questo dove sarebbe **`fork()`**. (E forse anche `sigaction`).

Cerca nell'help di VC++ per "fork" o "POSIX" e guarda se ti dà qualche indizio.

Se questo non funziona per niente, molla **`fork()`**/`sigaction` e sostituiscila con l'equivalente Win32: **`CreateProcess()`**. Non so come si usa **`CreateProcess()`**—prende un *bazilione* di argomenti, ma dovrebbe essere trattata nei documenti che sono assieme a VC++.

Sono dietro un firewall—come consento alle persone all'esterno del firewall di conoscere il mio indirizzo IP cosicché possano connettersi alla mia macchina?

Per sfortuna, lo scopo di un firewall è impedire alle persone all'esterno del firewall di connettersi alla macchina dentro il firewall, quindi permettergli di fare questo è considerato fondamentalmente una falla di sicurezza.

Non è detto che tutto sia perso. Per una cosa, puoi sempre usare **`connect()`** attraverso il firewall se esso fa un qualche tipo do masquerading o NAT o qualcosa di simile. Progetta i tuoi programmi in modo che sei sempre il primo che inizia la connessione, e sarai a posto.

Se questo non è soddisfacente, puoi chiedere al tuo sysadmin di lasciare un buco nel firewall in modo che le persone possano connettersi a te. Il firewall puoi inoltrarli a te sia attraverso il suo software NAT, oppure per mezzo di un proxy o qualcosa di simile.

Sii consapevole che un buco nel firewall non è una cosa da prendere alla leggera. Devi assicurarti di non dare accesso alla rete interna a malintenzionati; se sei un principiante, è molto più dura di quanto immagini creare un software sicuro.

Non far diventare il tuo sysadmin pazzo di me. `;-)`

Come scrivo uno sniffer di pacchetto? Come metto la mia interfaccia Ethernet in modalità promiscua?

Per quelli che non lo sanno, quando una scheda di rete è in "modalità promiscua" inoltrerà TUTTI i pacchetti al sistema operativo, non sono quelli che erano indirizzati a questa particolare macchina. (Stiamo parlando d'indirizzi a livello Ethernet qui, non d'indirizzi IP –ma dato che ethernet è un livello inferiore d'IP, tutti gl'indirizzi IP sono allo stesso modo inoltrati. Vedi la sezione Assurdità a Basso Livello e Teoria del Networking per maggiori informazioni).

Queste sono le basi di come funziona uno sniffer di pacchetti. Pone l'interfaccia in modalità promiscua, poi il SO prende ogni singolo pacchetto che passa. Avrai un socket di qualche genere dsa cui puoi leggere i dati.

Sfortunatamente, la risposta alla domanda varia a seconda della piattaforma, ma se usi Google, ad esempio, "windows promiscuo ioctl" probabilmente andrai da qualche parte. Questo è quello che sembra un articolo decente, in Linux Journal.

Come posso impostare un valore di timeout custom per un socket TCP o UDP?

Dipende dal sistema. Potresti cercare in rete per `SO_RCVTIMEO` e `SO_SNDTIMEO` (da usare con **`setsockopt()`**) per vedere se il tuo sistema supporta tale funzionalità.

Le pagine man di Linux suggeriscono di usare **`alarm()`** o **`setitimer()`** come sostituto.

Come posso sapere quali porte sono disponibili all'uso? C'è una lista di numeri di porta "ufficiali"?

Di solito questo non è un problema. Se stai scrivendo, diciamo, un server web, allora è una buona idea la ben nota porta 80. Se stai scrivendo solo il tuo server specializzato, allora scegli una porta a caso (ma maggiore di 1023) e fai un tentativo.

Se la porta è già in uso, ottieni l'errore "Address already in use" (N.d.T: Indirizzo già in uso) quando provi a fare **`bind()`**. Scegli un'altra porta. (E' una buona idea consentire ai tuoi utenti di specificare una porta alternativa con un file di configurazione o uno switch a linea di comando).

C'è un elenco di numeri di porta ufficiali tenuto dalla Internet Assigned Numbers Authority (IANA). Solo perché qualcosa (sopra 1023) è in quella porta non significa che puoi usare quella lista. Per esempio, le Id del software DOOM usando la stessa porta di "mdqs", qualunque cosa sia. Tutto quello che importa è che nessun altro *sulla stessa macchina* usi la stessa porta quando vuoi usarla.

8. Pagina Man

Nel mondo Unix, ci sono un sacco di manuali. Hanno piccole pagine che descrivono le singole funzioni che hai a tua disposizione.

Naturalmente, **manual** sarebbe troppo da digitare. Intendo, a nessuno in Unix, compreso me stesso, piace scrivere troppo. Indubbiamente potrei continuare a lungo su quanto preferisco essere conciso ma invece sarò breve e non voglio annoiarti con verbose diatribe di quanto sorprendentemente preferisco essere sintetico virtualmente in ogni circostanza.

[Applauso]

Grazie. Quello che cerco di far capire è che queste pagine sono chiamate "man pages" nel mondo Unix, e ho incluso la mia personale versione tronca per una vostra piacevole lettura. Il fatto è che molte di queste funzioni sono di uso più generale di quello che gli permetto, ma presenterò solo le parti che sono rilevanti per la Programmazione dei Socket Internet.

Ma aspetta! Questo non è tutto quello che c'è di sbagliato nelle pagine man:

- Sono incomplete e mostrano solo le basi di questa guida.
- Ci sono più pagine man di questa nel mondo reale.
- Sono diverse da quelle del tuo sistema.
- I file header potrebbero essere diversi per certe funzioni nel tuo sistema.
- I parametri potrebbero differire per certe funzioni nel tuo sistema.

Se vuoi informazioni originali, controlla le tue locali man page digitando **man *qualsiasi cosa***, dove "qualsiasi cosa" è un qualcosa in cui sei incredibilmente interessato, come "`accept`". (Sono sicuro che Microsoft Visual Studio ha qualcosa di simile nella guida in linea. Ma "man" è migliore perché è un byte più conciso di "help". Unix vince ancora!).

Quindi, se sono così piene di difetti, perché includerle persino tutte nella Guida? Beh, ci sono poche ragioni, ma le migliore sono che (a) queste versioni sono adattate specificatamente per la programmazione di rete e sono più facili da assimilare di quelle reali, e (b) queste versioni contengono degli esempi!

Oh! E parlando degli esempi, Non tendo a mettere tutto il controllo degli errori perché incrementerebbe veramente la lunghezza del codice, ma devi assolutamente fare il controllo degli errori ogni volta che fai una system call a meno che non sei totalmente sicuro che non fallirà, e dovresti probabilmente farlo anche in quel caso!

8.1. `accept()`

Accetta una connessione in entrata su un socket in ascolto

Prototipo

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Descrizione

Una volta che hai superato la difficoltà di ottenere un socket `SOCK_STREAM` e impostarlo per le connessioni in ingresso con **`listen()`**, allora chiami **`accept()`** per avere un nuovo descrittore di socket da usare per successive comunicazioni con ogni nuovo client connesso.

Il vecchio socket che usi per ascoltare è ancora lì, e sarà usato per ulteriori call ad **`accept()`**.

s	Il descrittore socket di **`listen()`**.
addr	Questo viene riempito con l'indirizzo di chi si connette.
addrlen	Questo viene riempito con le dimensioni (tramite sizeof()) della struttura restituita dal parametro *addr*. Puoi ignorarlo liberamente se supponi di avere indietro una `struct sockaddr_in`, come sai che è, perché è il tipo che gli ha passato in *addr*.

`accept()` normalmente blocca, e puoi usare **`select()`**per sapere in anticipo se il descrittore di socket in ascolto è "pronto per essere letto"; se è così, allora c'è una nuova connessione in attesa di essere accettata! Sii! In alternativa puoi settare il flag `O_NONBLOCK` sul socket in ascolto usando **`fcntl()`**, e allora non si bloccherà mai, altrimenti restituirà `-1` con *errno* settata a `EWOULDBLOCK`.

Il descrittore socket restituito da **`accept()`** è autentico, aperto e connesso all'host remoto. Lo devi chiudere (con **`close())`** quando hai finito con esso.

Valore di Ritorno

`accept()` restituisce un nuovo descrittore di socket connesso, o `-1` in caso di errore, con *errno* impostata di conseguenza.

Esempio

```
int s, s2;
struct sockaddr_in myaddr, remoteaddr;
socklen_t remoteaddr_len;

myaddr.sin_family = AF_INET;
```

```
myaddr.sin_port = htons(3490); // i client si connettono a questa
porta
myaddr.sin_addr.s_addr = INADDR_ANY; /* seleziona automaticamente
un indirizzo IP*/

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)myaddr, sizeof myaddr);

listen(s, 10); // imposta s per essere il socket del server (in
ascolto)
for(;;) {
    s2 = accept(s, &remoteaddr, &remoteaddr_len);

    // ora puoi usare send() e  recv() con il
    // client connesso tramite il socket s2
}
```

Vedi anche

socket(), **listen()**, struct sockaddr_in

8.2. `bind()`

Associa un socket ad un indirizzo IP e a un numero di porta

Prototipo

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Descrizione

Quando una macchina remota vuole connettersi al tuo server, ha bisogno di due informazioni: l'indirizzo IP e il numero di porta. La chiamata **`bind()`** ti consente appunto di fare questo.

Primo, chiami **`socket()`** per ottenere un descrittore di socket, e poi carichi una `struct sockaddr_in` con l'indirizzo IP e il numero di porta, e poi li passi entrambi a **`bind()`**, e l'indirizzo IP e la porta sono magicamente (veramente per magia) legati al socket!

Se non conosci il tuo indirizzo IP, o sai che hai un solo indirizzo IP sulla macchina, o non ti curi di quali numero IP viene usato, puoi semplicemente impostare il campo `s_addr` nella tua `struct sockaddr_in` a `INADDR_ANY` e inserirà l'indirizzo IP al tuo posto.

Per ultimo, il parametro *`addrlen`* deve essere impostato a `sizeof my_addr`.

Valore di Ritorno

Ritorna zero in caso di successo, o `-1` su errore (ed **`errno`** è settata di conseguenza).

Esempio

```
struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);

// puoi specifare un indirizzo IP:
inet_aton("63.161.169.137", &myaddr.sin_addr.s_addr);

// o puoi lasciargli sceglierne uno in automatico:
myaddr.sin_addr.s_addr = INADDR_ANY;

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)myaddr, sizeof myaddr);
```

Vedi Anche

`socket()`, `struct sockaddr_in`, `struct in_addr`

8.3. `connect()`

Connette un socket ad un server

Prototipo

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);
```

Descrizione

Una volta che hai un descrittore di socket a seguito di una call a **`socket()`**, puoi connettere quel socket ad un server remoto usando la chiamata di sistema **`connect()`**. Tutto ciò di cui hai bisogno è passargli il descrittore di socket e l'indirizzo del server a cui sei interessato. (Oh, e la lunghezza dell'indirizzo, la quale viene comunemente passata a funzioni come questa).

Se non hai ancora chiamato **`bind()`**, il descrittore di socket viene automaticamente collegato al tuo indirizzo IP e ad una porta locale casuale. Di solito questo ti va bene, dato che non t'importa qual'è la tua porta locale; t'interessa solo sapere qual è la porta remota cosicché la puoi mettere nel parametro *`serv_addr`*. Puoi chiamare **`bind()`** se veramente vuoi che il tuo client sia su uno specifico indirizzo IP e porta, ma questo è piuttosto raro.

Una volta che il socket è connesso, sei libero d'inviare (**`send()`**) e ricevere (**`recv()`**) tramite esso.

Nota speciale: se connetti un socket UDP `SOCK_DGRAM` ad un host remoto, puoi usare **`send()`** e **`recv()`** così come **`sendto()`** e **`recvfrom()`**. Se vuoi.

Valore di Ritorno

Ritorna zero in caso di successo, o `-1` in caso di errore (ed **`errno`** verrà impostata di conseguenza).

Esempio

```
int s;
struct sockaddr_in serv_addr;

/* facciamo finta che il server è in ascolto all'indirizzo
63.161.169.137 sulla porta 80:*/

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(80);
inet_aton("63.161.169.137", &myaddr.sin_addr.s_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
connect(s, (struct sockaddr*)myaddr, sizeof myaddr);

// ora siamo pronti a inviare e a ricevere
```

Vedi Anche

`socket()`, **`bind()`**

8.4. `close()`

Chiude un descrittore di socket

Prototipo

```
#include <unistd.h>

int close(int s);
```

Descrizione

Dopo che hai finito di usare il socket per qualsiasi folle progetto che hai escogitato e non vuoi inviare (**`send()`** o ricevere (**`recv()`** oppure, di sicuro, non fare nient'altro con il socket, lo puoi chiudere (**`close()`**, e verrà rilasciato (dalla memoria), per non essere più usato.

La parte remota può dire se questo si è verificato, in due modi. Uno: se dal lato remoto una **`recv()`** ritorna `0`. Due: se dalla parte remota una call a **`send()`** riceve un segnale di `SIGPIPE`, con la funzione che ritorna `-1` con *`errno`* impostato a `EPIPE`.

Utenti **Windows**: la funzione che dovete usare si chiama **`closesocket()`**, non **`close()`**. Se tenti di usare c**`lose()`** su un descrittore di socket, è possibile che Windows si arrabbi... E questo non ti piacerà.

Valore di Ritorno

Ritorna zero in caso di successo, o `-1` su errore (ed **`errno`** verrà impostata conseguentemente).

Esempio

```
s = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// tutto quanto...*BRRRONNNN!*
.
.
.
close(s);  // non c'è davvero molto.
```

Vedi Anche

socket(), **shutdown()**

8.5. `gethostname()`

Ritorna il nome del sistema

Prototipo

```
#include <sys/unistd.h>

int gethostname(char *name, size_t len);
```

Descrizione

Il tuo sistema ha un nome. Tutti ce l'hanno. Questo è una cosa leggermente più Unix rispetto a quello di cui abbiamo parlato, ma ha ancora i suoi usi.

Per esempio, puoi avere il tuo host name, e allora chiamare **`gethostbyname()`** per trovare il tuo indirizzo IP.

Il parametro *`name`* deve puntare ad un buffer che contiene il nome host, e *`len`* è la lunghezza del buffer in byte. **`gethostname()`** non sovrascrive la fine del buffer (potrebbe restituire un errore, o potrebbe solo smettere di scrivere), e inserisce un carattere terminatore (NULL) nella stringa se c'è spazio nel buffer.

Valore di Ritorno

Restituisce zero in caso di successo, o `-1` in caso di errore (ed **`errno`** impostata di conseguenza.)

Esempio

```
char hostname[128];

gethostname(hostname, sizeof hostname);
printf("Il mio hostname è: %s\n", hostname);
```

Vedi Anche

`gethostbyname()`

8.6. `gethostbyname()`, `gethostbyaddr()`

Restituisce un indirizzo IP dato un hostname, o viceversa

Prototipi

```
#include <sys/socket.h>
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

```
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Descrizione

Queste funzioni convertono da e verso host name e indirizzi IP. Dopo tutto, tu vuoi un indirizzo IP da passare a connect(), giusto? Ma nessuno vuole ricordarsi un indirizzo IP, quindi lascia che gli utenti digitino cose come "www.yahoo.com" invece di "66.94.230.35".

`gethostbyname()` prende una stringa come "www.yahoo.com", e restituisce una struct hostent che contiene una gran quantità d'informazioni, incluso l'indirizzo IP. (Altre informazioni sono il nome host canonico, un elenco di alias, il tipo d'indirizzo, la lunghezza degli indirizzi, e l'elenco degli indirizzi—è una struttura ad uso generale che è abbastanza facile da usare per scopi specifici una volta che sai come).

`gethostbyaddr()` prende una `struct in_addr` e ti dà il nome host corrispondente (se c'è), per cui è una specie di **`gethostbyname()`** inversa. Per quanto riguarda i parametri, sebbene *`addr`* è un `char*`, in effetti tu vuoi passare un puntatore a `struct in_addr`. *`len`* deve essere `sizeof(struct in_addr)`, e *`type`* `AF_INET`.

Quindi cos'è questa `struct hostent` che viene restituita? Ha un numero di campi che contengono informazioni sull'host in questione.

`char *h_name`	Il nome host canonico.
`char **h_aliases`	Un elenco di alias a cui si può accedere con array—l'ultimo elemento è `NULL`
`int h_addrtype`	Il tipo d'indirizzo, che per i nostri scopi può essere `AF_INET`.
`int length`	La lunghezza degli indirizzi in byte, che è 4 per gli indirizzi IPv4.
`char **h_addr_list`	Un elenco d'indirizzi IP per questo host. Anche se questo è un `char**`, è in effetti un array di `struct in_addr*`s camuffato. L'ultimo elemento dell'array è `NULL`.
`h_addr`	Un alias comunemente definito per *`h_addr_list[0]`*. Se vuoi proprio un vecchio indirizzo IP per questo host (Si, possono averne più di uno) usa proprio questo campo.

Valore di Ritorno

Restituisce un puntatore alla risultante `struct hostent` in caso di successo, o `NULL` su errore.

Invece della normale **`perror()`** e tutto quello che normalmente uso per il report degli errori, queste funzioni hanno dei risultati paralleli nella variabile *`h_errno`*, che possono essere stampati usando la funzione **`herror()`** o **`hstrerror()`**. Questi funzionano proprio come i classici *`errno`*, **`perror()`**, e **`strerror()`** a cui sei abituato.

Esempio

```
int i;
struct hostent *he;
struct in_addr **addr_list;
struct in_addr addr;

// ottiene gli indirizzi di www.yahoo.com:

he = gethostbyname("www.yahoo.com");
if (he == NULL) { // fa un controllo degli errori
    herror("gethostbyname"); // herror(), NON perror()
    exit(1);
}

// stampa le informazioni di questo host:
printf("Nome canonico is: %s\n", he->h_name);
printf("Indirizzo IP: %s\n", inet_ntoa(*(struct in_addr*)he-
>h_addr));
printf("Tutti gli indirizzi: ");
addr_list = (struct in_addr **)he->h_addr_list;
for(i = 0; addr_list[i] != NULL; i++) {
    printf("%s ", inet_ntoa(*addr_list[i]));
}
printf("\n");

// ottiene l'host name di 66.94.230.32:

inet_aton("66.94.230.32", &addr);
he = gethostbyaddr(&addr, sizeof addr, AF_INET);

printf("Nome host: %s\n", he->h_name);
```

Vedi Anche

gethostname(), *errno*, **perror()**, **strerror()**, struct in_addr

8.7. `getpeername()`

Ritorna informazioni sulla parte remota della connessione

Prototipo

```
#include <sys/socket.h>

int getpeername(int s, struct sockaddr *addr, socklen_t *len);
```

Descrizione

Una volta che hai sia accettato (**`accept()`**) una connessione remota, o ti sei connesso (**`connect()`**) ad un server, hai quello che si chiama un *peer* (un nodo). Il tuo peer è semplicemente il computer a cui sei connesso, identificato da un indirizzo IP e da una porta. Quindi...

`getpeername()` restituisce semplicemente una `struct sockaddr_in` riempita con le informazioni sulla macchina su cui sei connesso.

Perché si chiama "name"? Bene, ci sono diversi tipi di socket, non solo socket Internet come quelli che usiamo in questa guida, e così "name" era un bel termine d'uso generale che comprendeva tutti i casi. In questo caso, sebbene, il "name" del peer e il suo indirizzo IP e numero di porta.

Anche se la funzione restituisce le dimensioni dell'indirizzo risultante in *`len`*, devi precaricare *`len`* con le dimensioni di *`addr`*.

Valore di Ritorno

Restituisce zero in caso di successo, o `-1` su errore (ed **`errno`** sarà impostato di conseguenza).

Esempio

```
int s;
struct sockaddr_in server, addr;
socklen_t len;
```

```
// crea un socket
s = socket(PF_INET, SOCK_STREAM, 0);

// si connette ad un server
server.sin_family = AF_INET;
inet_aton("63.161.169.137", &server.sin_addr);
server.sin_port = htons(80);

connect(s, (struct sockaddr*)&server, sizeof server);

// ottiene il nome del peer
/* sappiamo che ci siamo appena connessi a 63.161.169.137:80, così
questo dovrebbe stampare:*/
//    Indirizzo IP del peer: 63.161.169.137
//    Porta del peer      : 80

len = sizeof addr;
getpeername(s, (struct sockaddr*)&addr, &len);
printf("Indirizzo IP del peer: %s\n", inet_ntoa(addr.sin_addr));
printf("Porta del peer : %d\n", ntohs(addr.sin_port));
```

See Also

gethostname(), **gethostbyname()**, **gethostbyaddr()**

8.8. *errno*

Contiene il codice di errore dell'ultima chiamata di sistema

Prototipo

```
#include <errno.h>

int errno;
```

Descrizione

Questa è la variabile che contiene le informazioni sugli errori di molte system call. Se ti ricordi, funzioni come **`socket()`** e **`listen()`** ritornano `-1` in caso di errore, e impostano *`errno`* al valore esatto per farti conoscere lo specifico errore verificatosi.

Il file header *`errno.h`* elenca un insieme di nomi simbolici per gli errori, come `EADDRINUSE`, `EPIPE`, `ECONNREFUSED`, etc. Le tue pagine man locali quali codici di errore possono essere restituiti, e puoi vedere questo a run time, per gestire differenti errori in diversi modi.

O, più comunemente, puoi chiamare **`perror()`** o **`strerror()`** per ottenere una versione di errore leggibile dagli umani.

Valore di Ritorno

Il valore della variabile è l'ultimo errore che è occorso, che potrebbe anche essere il codice di "successo" se l'ultima azione è stata completata con successo.

Esempio

```
s = socket(PF_INET, SOCK_STREAM, 0);
if (s == -1) {
    perror("socket"); // or use strerror()
}

tryagain:
if (select(n, &readfds, NULL, NULL) == -1) {
    // si è verificato un errore!!

    // se siamo stati solo interrotti, riavviamo select():
    if (errno == EINTR) goto tryagain;  // AAAA!  goto!!!

    // altrimenti è un errore più grave:
    perror("select");
    exit(1);
}
```

Vedi Anche

`perror()`, **`strerror()`**

8.9. `fcntl()`

Controlla i descrittori di socket

Prototipo

```
#include <sys/unistd.h>
#include <sys/fcntl.h>

int fcntl(int s, int cmd, long arg);
```

Descrizione

Questa funzione viene comunemente usata per il bloccaggio e altre cose sui file, ma ha anche un paio di funzioni correlate ai socket che potresti vedere o usare di tanto in tanto.

Il parametro *s* è il descrittore di socket su cui desideri operare, *cmd* deve essere impostato a `F_SETFL`, e *arg* può essere uno dei seguenti comandi. (Come ho detto, c'è di più di **`fcntl()`** rispetto a quello che divulgo qui, ma sto cercando di restare orientato ai socket.)

`O_NONBLOCK` Imposta il socket come non bloccante. Guarda la sezione sul bloccaggio per maggiori dettagli.

`O_ASYNC` Setta il socket per eseguire dell'I/O. Quando i dati sono pronti per essere ricevuti, sarà notificato il segnale `SIGIO`. Questo è raro ed oltre gli scopi di questa guida. E penso che sia disponibile solo in certi sistemi.

Valore di Ritorno

Restituisce zero in caso di successo, o `-1` su errore (ed **`errno`** sarà settata di conseguenza).

Per la verità differenti usi di **`fcntl()`** hanno differenti valori di ritorno, ma non gli ho trattati perché non riguardano i socket. Guarda la tua pagina man locale su **`fcntl()`** per maggiori informazioni.

Esempio

```
int s = socket(PF_INET, SOCK_STREAM, 0);

fcntl(s, F_SETFL, O_NONBLOCK);  // imposta come non bloccante
fcntl(s, F_SETFL, O_ASYNC);     // imposta come I/O asincrono
```

Vedi Anche

Bloccaggio, **`send()`**

8.10. `htons()`, `htonl()`, `ntohs()`, `ntohl()`

Converte tipi interi multi-byte da host byte order a network byte order

Prototipi

```
#include <netinet/in.h>

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

Descrizione

Giusto per renderti veramente infelice, i computer usano internamente differenti ordinamenti di byte per i loro interi multibyte (cioè ogni interi più grande di un `char`). Il risultato di questo è che se invii due `short int` a 2 byte da un box Intel a uno Mac (prima che gli ultimi sono diventati anche loro box Intel, Intendo), quello che un computer “pensa” sia il numero `1`, l’altro penserà essere `256`, e viceversa.

Il modo di eludere il problema per chiunque è accantonare le differenze e concordare che Motorola e IBM avevano ragione, e che Intel scelse il modo strano, e quindi convertiamo tutti i nostri ordinamenti di byte in "big-endian" prima d’inviarli. Dato che Intel è una macchina "little-endian", è molto più politcamente corretto chiamare il nostro ordinamento di byte preferito "Network Byte Order". Quindi queste funzioni convertono da un byte order nativo a network byte order e viceversa.

(Questo significa che in Intel queste funzioni scambiano tutti i byte e su PowerPC non fanno niente perché i byte sono già in Network Byte Order, ma dovresti sempre usarle nel tuo codice, poiché qualcuno potrebbe voler sviluppare su macchine Intel e avere ancora le cose funzionanti).

Nota che i tipi coinvolti sono numeri a 32-bit (4 byte, forse `int`) e 16-bit (2 byte, molto probabilmente `short`). Macchine a 64-bit potrebbero avere una **`htonll()`** per `int` a 64-bit, ma non l’ho vista. Dovresti scrivertela.

Ad ogni modo, il modo in cui sono queste funzioni è che devi prima decidere se devi convertire *da* host (la tua macchina) byte order o da network byte order, se è "host", la prima lettera della funzione che chiamerai è "h”, altrimenti è "n" per "network". Nel mezzo delle funzioni c’è sempre "to" perché stai convertendo da uno (“from”) all’altro ("to"), e la penultima lettera mostra a cosa stai convertendo. L’ultima lettera sono le dimensioni del dato, "s" per short, o "l" per long. perciò:

`htons()` Da **h**ost **a** short **n**etwork

`htonl()` Da **h**ost a **`long`** **n**etwork

`ntohs()` Da **n**etwork **`a short`** **h**ost

`ntohl()` Da **n**etwork **`a long`** **h**ost

Valore di Ritorno

Ciascuna funzione restituisce il valore convertito.

Esempio

```
uint32_t some_long = 10;
uint16_t some_short = 20;

uint32_t network_byte_order;

// converte e invia
network_byte_order = htonl(some_long);
send(s, &network_byte_order, sizeof(uint32_t), 0);

some_short == ntohs(htons(some_short)); // questa espressione è
vera
```

8.11. `inet_ntoa()`, `inet_aton()`

Converte indirizzi IP da una stringa punti-e-numeri a una `struct in_addr`, e viceversa

Prototipi

```
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>

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

Descrizione

Tutte queste funzioni convertono da una `struct in_addr` (parte della tua `struct sockaddr_in`, molto probabilmente) a una stringa nel formato punti e numeri (es. "192.168.5.10") , e viceversa. Se hai un indirizzo IP passato da linea di comando o da qualche parte, questo è il modo più semplice di ottenere una `struct in_addr` a cui connettersi. Se vuoi più potere,prova qualche funzione DNS come **`gethostbyname()`** o tenta un *coup d'État* (*colpo di stato*) nel tuo paese.

La funzione **`inet_ntoa()`** converte un indirizzo di rete in una `struct in_addr` ad una stringa nel formato punti-e-numeri. La "n" in "ntoa" sta per network, e la "a" sta per ASCII ragioni storiche (quindi è "Network To ASCII"—il prefisso "toa" ha un amico simile nella libreria C di nome **`atoi()`**, che converte una stringa ASCII in un intero).

La funzione **`inet_aton()`** è l'opposto, converte da una stringa "punti-e-numeri" ad una `in_addr_t` (che è il tipo di campo `s_addr` nella tua `struct in_addr`).

Infine, la funzione **`inet_addr()`** è una vecchia funzione che fondamentalmente fa la stessa cosa di **`inet_aton()`**. Teoricamente è deprecata, ma la vedrai ancora per molto e non verrà la polizia se la usi.

Valore di Ritorno

`inet_aton()` restituisce un valore non nullo se l'indirizzo è valido, e zero se non è appunto valido.

`inet_ntoa()` restituisce la stringa "punti e numeri" in un buffer statico che viene sovrascritto ad ogni chiamata della funzione.

`inet_addr()` restituisce l'indirizzo sotto forma di `in_addr_t`, o `-1` se c'è un errore. (Che è lo stesso risultato che hai se cerchi di convertire la stringa "255.255.255.255", il quale è un indirizzo IP valido. Ecco perché **`inet_aton()`** è meglio).

Esempio

```
struct sockaddr_in antelope;
char *some_addr;

inet_aton("10.0.0.1", &antelope.sin_addr); // memorizza IP in
antelope

some_addr = inet_ntoa(antelope.sin_addr); // ritorna l'IP
printf("%s\n", some_addr); // stampa "10.0.0.1"
```

```
// e questa chiamata è la stessa di inet_aton():
antelope.sin_addr.s_addr = inet_addr("10.0.0.1");
```

Vedi Anche

gethostbyname(), **gethostbyaddr()**

8.12. `listen()`

Dice al socket di restare in ascolto per connessioni in ingresso.

Prototipo

```
#include <sys/socket.h>

int listen(int s, int backlog);
```

Descrizione

Puoi prendere il tuo descrittore di socket (creato con la system call **`socket()`**) e dirgli di rimanere in ascolto di connessioni entranti. Questo è quello che differenzia i server dai client, ragazzi.

Il parametro *backlog* può significare un paio di cose diverse secondo il sistema in cui sei, ma approssimativamente è quante connessioni in sospeso puoi avere prima che il kernel inizi a rigettarne nuove. Quindi come arrivano nuove connessioni, devi essere veloce ad accettarle (**accept()**) affinché non si riempia il backlog (Lett.: lavoro arretrato). Cerca d'impostarlo a 10 o circa, e se i tuoi client iniziano ad avere "Connection refused" ("Connessione rifiutata") sotto carico pesante, impostalo più alto.

Prima di chiamare **listen()**, il tuo server deve invocare **bind()** per collegarsi ad uno specifico numero di porta. Quel numero di porta (all'indirizzo IP del server) sarà quello a cui si connetteranno i client.

Valore di Ritorno

Restituisce zero in caso di successo, o `-1` su errore (ed **errno** sarà impostata in conformità).

Esempio

```
int s;
struct sockaddr_in myaddr;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490); // i client si connettono a questa
porta
myaddr.sin_addr.s_addr = INADDR_ANY; // autoseleziona indirizzo IP

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)myaddr, sizeof myaddr);

listen(s, 10); // imposta s per essere un socket di un server (in
ascolto)

// poi abbiamo un accept() che segue giù da qualche parte
```

Vedi Anche

accept(), **bind()**, **socket()**

8.13. `perror()`, `strerror()`

Stampa una stringa d'errore leggibile dagli umani

Prototipi

```
#include <stdio.h>
#include <string.h>   // for strerror()

void perror(const char *s);
char *strerror(int errnum);
```

Descrizione

Poiché così tante funzioni restituiscono `-1` su errore e impostano il valore della variabile *`errno`* a un qualche valore, sarebbe sicuramente bello se potessi facilmente visualizzarlo in una forma che per te abbia un senso.

Compassionevolmente, **`perror()`** fa quello. Se vuoi che sia stampata una più ampia descrizione prima dell'errore, ci puoi far puntare il parametro *`s`* (o lasciarlo a `NULL` e non sarà visualizzato niente di più).

In poche parole, questa funzione prende valori *`errno`*, come `ECONNRESET`, e li stampa in modo gradevole, come "Connection reset by peer." (Lett.: connessione resettata dal nodo [client]).

La funzione **`strerror()`** è molto simile a **`perror()`**, eccetto che restituisce un puntatore alla stringa del messaggio di errore, per un dato valore (che di solito passi nella variabile *`errno`*).

Valore di Ritorno

`strerror()` ritorna un puntatore alla stringa del messaggio di errore.

Esempio

```
int s;
```

```
s = socket(PF_INET, SOCK_STREAM, 0);

if (s == -1) { // si è verificato un errore
    // prints "errore socket: " + il messaggio di errore:
    perror("errore socket ");
}

// similarly:
if (listen(s, 10) == -1) {
    // questa stampa "Errore: " + il messaggio di errore di errno:
    printf("Errore: %s\n", strerror(errno));
}
```

Vedi Anche

errno

8.14. `poll()`

Testa degli eventi su socket multipli simultaneamente

Prototipo

```
#include <sys/poll.h>
```

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

Descrizione

Questa funzione è molto simile a **`select()`** perché entrambe controllano insiemi di descrittori per il verificarsi di certi eventi, come arrivo di dati in ingresso, socket pronti a inviare dati, dati out-of-band pronti a ricevere, errori, etc.

L'idea di base è passare array *`nfds`* `struct pollfd` in *`ufds`*, assieme ad un timeout in millisecondi (1000 millisecondi sono un secondo). Il *`timeout`* può essere negativo se vuoi aspettare per sempre, se non si verifica nessun evento sui descrittori di socket entro il timeout, **`poll()`** ritornerà.

Ciascun elemento dell'array `struct pollfd` rappresenta un descrittore di socket, e contiene i seguenti campi:

```
struct pollfd {
    int fd;         // il descrittore di socket
    short events;   // insieme di eventi a cui siamo interessati
    short revents;  /*quando poll() ritorna, restituisce gli eventi
che si sono verificati*/
};
```

Prima di chiamare **`poll()`**, carica *`fd`* con il descrittore di socket (se imposti *`fd`* ad un numero negativo, questa `struct pollfd` viene ignorata e il suo campo *`revents`* è impostato a zero) e costruisce il campo di eventi *`events`* tramite OR bit a bit con le seguenti macro:

`POLLIN`	Avvisami quando il dato è pronto per essere ricevuto **`recv()`** su questo socket.
`POLLOUT`	Avvisami quando posso inviare dati **`send()`** a questo socket senza bloccare.
`POLLPRI`	Avvisami quando dati out-of-band sono pronti ad essere ricevuti su questo socket.

Una volta che **`poll()`**ritorna, il campo *`revents`* sarà ricostruito tramite OR bit a bit con i campi di cui sopra, che ti dicono quale evento si è verificato. In aggiunta, potrebbero essere presenti questi campi:

`POLLERR`	Si è verificato un errore nel socket.
`POLLHUP`	Si è interrotta la connessione remota.
`POLLNVAL`	C'era qualcosa di sbagliato nel descrittore di socket *`fd`*—forse non è stato inizializzato?

Valore di Ritorno

Restituisce il numero d'elementi nell'array *`ufds`* in cui si sono verificati degli eventi; che può essere zero se si è andati in timeout. Restituisce al solito `-1` su errore (con **`errno`** impostata di conseguenza).

Esempio

```
int s1, s2;
int rv;
char buf1[256], buf2[256];
struct pollfd ufds[2];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// qui s'ipotizza che siano entrambi connessi ad un server
//connect(s1, ...)...
//connect(s2, ...)...

// settiamo l'array dei descrittori di file.
                                                                        //
/* in questo esempio, vogliamo sapre quando si stanno arrivando
dati normali o out-of-band  */

ufds[0].fd = s1;
ufds[0].events = POLLIN | POLLPRI; // controlla per dati normali o
out-of-band

ufds[1] = s2;
ufds[1].events = POLLIN; // controlla solo dati normali

// attende questi eventi per 3,5 secondi
rv = poll(ufds, 2, 3500);

if (rv == -1) {
    perror("poll"); // si è verificato un errore in poll()
} else if (rv == 0) {
    printf("Timeout!  Nessun dato dopo 3,5 secondi.\n");
} else {
    // controlla eventi su s1:
    if (ufds[0].revents & POLLIN) {
        recv(s1, buf1, sizeof buf1, 0); // riceve normali dati
    }
    if (ufds[0].revents & POLLPRI) {
        recv(s1, buf1, sizeof buf1, MSG_OOB); // dati out-of-band
    }

    // controlla eventi su s2:
    if (ufds[1].revents & POLLIN) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

Vedi Anche

`select()`

8.15. `recv()`, `recvfrom()`

Riceve dati su un socket

Prototipi

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

Descrizione

Appena hai un socket pronto e connesso, puoi leggere i dati che arrivano dalla parte remota usando la funzione **`recv()`** (per socket TCP `SOCK_STREAM`) e **`recvfrom()`** (per socket UDP `SOCK_DGRAM`).

Entrambe prendono il descrittore di socket *`s`*, un puntatore al buffer *`buf`*, la lunghezza (in byte) del buffer *`len`*, e una serie di *`flag`* che controllano come lavora la funzione.

In aggiunta **recvfrom()** prende una `struct sockaddr*`, *`from`* da cui ti sarà detto da dove provengono i dati, e sarà riempito *`fromlen`* con le dimensioni di `struct sockaddr`. (Devi anche inizializzare *`fromlen`* con le dimensioni di *`from`* o `struct sockaddr`).

Allora quali meravigliosi flag puoi passare a questa funzione? Ecco alcuni di loro, ma dovresti controllare le tue pagine man locali per maggiori informazioni su cosa è effettivamente supportato nel tuo sistema. Puoi fare un OR bit a bit tra loro, o solo impostare *`flags`* a `0` se vuoi che sia una regolare **recv()**.

`MSG_OOB`	Ricevi dati Out of Band. Questo è come ricevere dati che ti sono stati spediti con il flag `MSG_OOB` in **send()**. Come parte ricevente, avresti dovuto avere il segnale `SIGURG` notificato per dirti che ci sono dati urgenti. Per gestire questo segnale, puoi chiamare **recv()** con questo flag `MSG_OOB`.
`MSG_PEEK`	Se vuoi invocare **recv()** "solo per finta", puoi chiamarlo con questo flag. Ti dirà cosa c'è in attesa nel buffer quando chiami "davvero" **recv()**(i.e. *senza* il flag `MSG_PEEK`. E' come uno sguardo furtivo alla prossima call a **recv()**.
`MSG_WAITALL`	Dice a **recv()** di non ritornare finché tutti i dati che hai specificato nel parametro *`len`* siano spediti. Ignorerà i tuo desideri in circostanze estreme, ad ogni modo, come quando un segnale interrompe la chiamata o si verifica qualche errore oppure la parte remota chiude la connessione, etc. Non andarne pazzo.

Quando chiami **recv()**, si bloccherà fino a che c'è qualche dato da leggere. Se non vuoi che si blocchi, imposta il socket come non bloccante o controlla con **select()** o **poll()** per vedere se ci sono dati in ingresso prima di chiamare **recv()** o **recvfrom()**.

Valore di Ritorno

Restituisce il numero di byte effettivamente ricevuti (che potrebbero essere meno di quelli richiesti nel parametro *`len`*), o `-1` su errore (ed **errno** verrà impostato conformemente).

Se dall'altra parte è stata chiusa la connessione, **recv()** ritornerà `0`. Questo è il normale metodo per determinare se la parte remota ha chiuso la connessione. Normalmente è buono, ribelle!

Esempio

```
int s1, s2;
int byte_count, fromlen;
struct sockaddr_in addr;
char buf[512];

// mostriamo prima un esempio con un socket stream TCP
```

```
s1 = socket(PF_INET, SOCK_STREAM, 0);

// informazioni sul server
addr.sin_family = AF_INET;
addr.sin_port = htons(3490);
inet_aton("10.9.8.7", &addr.sin_addr);

connect(s1, &addr, sizeof addr); // si connette al server

// Benissimo!  Ora che siamo connessi, possiamo ricevere dei dati!
byte_count = recv(s1, buf, sizeof buf, 0);
printf("ricevuti %d byte di dati in buf\n", byte_count);

// ora demo per datagram socket UDP:
s2 = socket(PF_INET, SOCK_DGRAM, 0);

fromlen = sizeof addr;
byte_count = recvfrom(s2, buf, sizeof buf, 0, &addr, &fromlen);
printf("Ricevuti %d byte di dati in buf\n", byte_count);
printf("Dall'indirizzo IP %s\n", inet_ntoa(addr.sin_addr));
```

Vedi Anche

`send()`, **`sendto()`**, **`select()`**, **`poll()`**, Blocking

8.16. `select()`

Controlla se un descrittore di socket è pronto in lettura/scrittura

Prototipo

```
#include <sys/select.h>

int select(int n, fd_set *readfds, fd_set *writefds, fd_set
*exceptfds,
           struct timeval *timeout);

FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Descrizione

La funzione **`select()`** ti dà il modo di controllare socket multipli per vedere se hanno dati pronti da essere ricevuti (con **`recv()`**, o se puoi spedire a loro dei dati (con **`send()`** senza bloccaggio, o se si è verificata qualche eccezione.

Popoli il tuo set di descrittori socket usande le macro, come **`FD_SET()`**. Una volta che hai il set, lo passi alla funzione come uno dei seguenti parametri: *`readfds`* se vuoi sapere se qualsiasi socket di quell'insieme è pronto a ricevere (con **`recv()`** dati, *`writefds`* se qualsiasi socket è pronto ad inviare (con **`send()`** dati, e/o *`exceptfds`* se hai bisogno di sapere quando si è verificata un eccezione (errore) su uno dei socket. Uno di questi o tutti i parametri possono essere `NULL` se non sei interessato a quei tipi d'eventi. Dopo che **`select()`** ritorna, i valori nei set saranno modificati per mostrare quali sono pronti in lettura o scrittura e quali hanno eccezioni.

Il primo parametro, *`n,`* è il descrittore socket di più alto valore (sono solo `int`, ricordi?) più uno.

Per ultimo, `struct timeval`, *`timeout`*, ti permette di scegliere per quanto tempo controllare questi set. La **`select()`** ritornerà dopo il timeout, o quando si verifica un evento, qualunque sia il primo. La `struct timeval` ha due campi: *`tv_sec`* è il numero dei secondi, a cui viene aggiunto *`tv_usec`*, il numero di microsecondi (1.000.000 di microsecondi in un secondo).

Le macro di aiuto fanno quanto segue:

`FD_SET(int fd, fd_set *set);`	Inserisce *`fd`* al *`set`*.
`FD_CLR(int fd, fd_set *set);`	Rimuove *`fd`* dal *`set`*.
`FD_ISSET(int fd, fd_set *set);`	Restituisce true (vero) se *`fd`* è nel set.
`FD_ZERO(fd_set *set);`	Elimina tutti gli elementi del *`set`*.

Valore di Ritorno

Restituisce il numero di descrittori nel set in caso di successo, `0` in caso di timeout, o `-1` su errore (ed **`errno`** sarà impostato di conseguenza). Poi, i set vengono modificati per mostrare quali socket sono pronti.

Esempio

```
int s1, s2, n;
fd_set readfds;
struct timeval tv;
char buf1[256], buf2[256];

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_STREAM, 0);

// ipotizziamo che siano entrambi connessi al server a questo punto
//connect(s1, ...)...
//connect(s2, ...)...

// libera il set in anticipo
FD_ZERO(&readfds);

// aggiunge i nostri descrittore al set
FD_SET(s1, &readfds);
FD_SET(s2, &readfds);

// poiché abbiamo s2 secondi, questo è il valore "più grande", così usiamo quello per
// il nostro parametro n in select()
n = s2 + 1;

// aspettiamo finché o i socket sono pronti a ricevere i dati o si verifica il timeout 10,5 secondi)
tv.tv_sec = 10;
tv.tv_usec = 500000;
rv = select(n, &readfds, NULL, NULL, &tv);

if (rv == -1) {
    perror("select"); // si è verificato un errore in select()
} else if (rv == 0) {
    printf("Timeout!  Nessun dato dopo 10,5 secondi.\n");
} else {
    // uno o entrambi i descrittori hanno dei dati
    if (FD_ISSET(s1, &readfds)) {
        recv(s1, buf1, sizeof buf1, 0);
    }
    if (FD_ISSET(s2, &readfds)) {
        recv(s1, buf2, sizeof buf2, 0);
    }
}
```

Vedi Anche

poll()

8.17. `setsockopt()`, `getsockopt()`

Imposta varie opzioni di un socket

Prototipi

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);
```

Descrizione

I socket sono delle bestie abbastanza configurabili. Di fatto, sono così configurabili, che qui non tratterò tutto. Probabilmente dipende comunque dal sistema, ma ti parlerò delle basi.

Ovviamente, queste funzioni ottengono e impostano certe opzioni su un socket. Su una Linux box, tutte le informazioni sui socket si trovano nell'omonima man page alla sezione 7. (Digita: "**man 7 socket**" per avere tutte queste "delizie").

Per quanto riguarda i parametri, *s* è il socket a cui stai parlando, *level* può essere impostato a SOL_SOCKET. Poi puoi settare *optname* al name a cui sei interessato. Di nuovo, guarda le tue pagine man per tutte le opzioni, ma ecco qua alcune delle più interessanti:

SO_BINDTODEVICE	Collega il socket ad un nome di dispositivo simbolico come eth0 invece di usare **bind()** per collegarlo ad un indirizzo IP. Digita il comando **ifconfig** in Unix per vedere tutti nomi di dispositivi.
SO_REUSEADDR	Consente agli altri socket di eseguire il **bind()** a questa porta, a meno che non c'è già un socket attivo in ascolto. Questo ti consente di evitare il messaggio di errore "address already in use" (N.d.T.: indirizzo già in uso) quando cerchi di riavviare il server dopo un crash.
SO_BROADCAST	Consente a socket datagram (SOCK_DGRAM) UDP d'inviare e ricevere pacchetti da e verso indirizzi di broadcast. Non fa niente—*NIENTE!!*—su stream socket TCP! Hahaha!

Per quanto concerne il parametro *optval*, è di solito un puntatore ad un int che ne indica il valore in questione. Per i tipi booleani zero è falso, e non zero è vero. E' un fatto assoluto nel tuo sistema. Se non c'è nessun parametro da passare, *optval* può essere NULL.

Il parametro finale, *optlen*, viene riempito al tuo posto da **getsockopt()**con le dimensioni (devi usare sizeof(int)) di *optname.*
Avvertimento: su alcuni sistemi (in particolare Sun e Windows), l'opzione può essere un char invece di un int, per esempio, il carattere '1' invece dell'intero 1. Di nuovo, controlla le tue pagine man per maggiori info con "**man setsockopt**" e "**man 7 socket**"!

Valore di Ritorno

Ritorna zero in caso di successo, o -1 su errore (ed **errno** è impostata di conseguenza).

Esempio

```
int optval;
int optlen;
char *optval2;

// imposta SO_REUSEADDR a true su un socket (1):
optval = 1;
setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval);

// collega un socket ad un device name (potrebbe non funzionare su
tutti i sistemi):
optval2 = "eth1"; // lungo 4 byte:
setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4);
```

```
// Vedi se il flag SO_BROADCAST è impostato:
getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen);
if (optval != 0) {
    print("SO_BROADCAST abilitato su s3!\n");
}
```

Vedi Anche

fcntl()

8.18. `send()`, `sendto()`

Invia dati su un socket

Prototipi

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, const void *buf, size_t len, int flags);
ssize_t sendto(int s, const void *buf, size_t len,
               int flags, const struct sockaddr *to,
               socklen_t tolen);
```

Descrizioni

Queste funzioni inviano dati ad un socket. Parlando in generale, **`send()`** viene usato per socket `SOCK_STREAM` TCP connessi, e **`sendto()`** per socket `SOCK_DGRAM` UDP non connessi. Con i socket non connessi, devi specificare la destinazione del pacchetto ogni volta che lo invii, e questo perché l'ultimo parametro di **`sendto()`** indica dove va il pacchetto.

Sia con **`send()`** che con **`sendto()`**, il parametro *`s`* è il socket, *`buf`* è un puntatore ai dati che vuoi inviare, *`len`* è il numero di byte che vuoi spedire, e *`flags`* serve per specificare ulteriori informazioni su come vanno spediti i dati. Setta *`flags`* a zero se vuoi che siano dati "normali". Ecco qui alcuni dei più comuni flag utilizzati, ma controlla le tue pagine man locali **`send()`** per maggiori dettagli:

`MSG_OOB`	Invia dati "out of band". TCP supporta questo, ed è un modo di dire al ricevitore che questi dati hanno una priorità più alta del normale. Il destinatario riceverà il segnale `SIGURG` e potrebbe ricevere questi dati prima di quelli normali in coda.
`MSG_DONTROUTE`	Non inviare questi dati ad un router, mantienili locali.
`MSG_DONTWAIT`	Se **`send()`** blocca a causa di traffico in uscita congestionato, fagli ritornare `EAGAIN`. Questo è come "abilità il non bloccaggio solo per questo invio". Vedi la sezione sul bloccaggio per ulteriori dettagli.
`MSG_NOSIGNAL`	Se invii ad un host remoto che non sta più ricevendo, tipicamente otterrai il segnale `SIGPIPE`. Aggiungere questo flag impedisce che sia sollevata quest'eccezione.

Valore di Ritorno

Ritorna il numero di byte effettivamente inviati, o `-1` su errore (ed **`errno`** sarà impostata di conseguenza). Nota che il numero di byte inviati davvero potrebbe essere inferiore al numero che gli hai chiesto di spedire! Vedi la sezione maneggiare send() parziali per una funzione che aiuta a risolvere questo problema.

Poi, se il socket è stato chiuso da qualunque parte, il processo che chiama **`send()`** riceverà il segnale `SIGPIPE`. (A meno che **`send()`** non sia stato chiamato con il flag `MSG_NOSIGNAL`.)

Esempio

```
int spatula_count = 3490;
char *secret_message = "Il formaggio è nel tostapane";

int stream_socket, dgram_socket;
struct sockaddr_in dest;
int temp;

// first with TCP stream sockets:
```

```
stream_socket = socket(PF_INET, SOCK_STREAM, 0);
.
.
.
// converte a network byte order
temp = htonl(spatula_count);
// invia i dati normalmente:
send(stream_socket, &temp, sizeof temp, 0);

// invia un messaggio segreto out of band:
send(stream_socket, secret_message, strlen(secret_message)+1,
MSG_OOB);

// ora con socket datagram UDP:
dgram_socket = socket(PF_INET, SOCK_DGRAM, 0);
.
.
.
// calcola la destinazione
dest.sin_family = AF_INET;
inet_aton("10.0.0.1", &dest.sin_addr);
dest.sin_port = htons(2223);

// invia il messaggio segreto normalmente:
sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0,
       (struct sockaddr*)&dest, sizeof dest);
```

Vedi Anche

recv(), **recvfrom()**

8.19. `shutdown()`

Ferma ulteriori invii e ricezioni su un socket

Prototipo

```
#include <sys/socket.h>

int shutdown(int s, int how);
```

Descrizione

E' tutto! Basta! Non sono consentiti altri **`send()`** su questo socket, ma voglio ancora ricevere dati! O viceversa! Come posso fare questo?

Quando chiudi (con **`close()`**o **`closesocket()`**) un descrittore di socket, si chiudono entrambi i lati: in scrittura e lettura, ed è liberato il descrittore di socket; se vuoi solo chiudere un lato, puoi usare la call **`shutdown()`**.

Per quanto concerne i parametri, *s* è ovviamente il descrittore di socket su cui vuoi eseguire quest'azione, e il tipo d'operazione con il parametro *how*. Questo può essere `SHUT_RD` per impedire ulteriori **`recv()`**, `SHUT_WR` per proibire nuove **`send()`**s, o `SHUT_RDWR` per negare entrambe.

Nota che **`shutdown()`** non libera (la memoria occupata da) il descrittore di socket, quindi devi usare **`close()`** anche se il socket è stato "chiuso".

Questa system call è raramente usata.

Valore di Ritorno

Restituisce zero in caso di successo, o `-1` su errore (ed **`errno`** sarà impostata di conseguenza).

Esempio

```
int s = socket(PF_INET, SOCK_STREAM, 0);

// ...eseguo alcune send() e altre cose...

// e ora che abbiamo finito, non sono consentite altre send():
shutdown(s, SHUT_RD);
```

Vedi Anche

close()

8.20. `socket()`

Alloca un descrittore di socket

Prototipo

```
#include <sys/types.h>
#include <sys/socket.h>

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

Descrizione

Restituisce un nuovo descrittore di socket che può essere usato per fare varie cose di rete. Questa è in genere la prima chiamata nell'enorme procedura di scrivere un programma socket, e puoi usare il risultato per ulteriori chiamate a **`listen()`**, **`bind()`**, **`accept()`**, o per altre funzioni.

`domain`

`domain` descrive il tipo di socket a cui sei interessato. Questo può essere, credimi, un'ampia varietà di cose, ma dato che questa è una guida sui socket Internet, per te sarà `PF_INET`. E, di conseguenza, quando carichi la tua `struct`

`sockaddr_in`, imposterai il campo `sin_family` a `AF_INET`.

(D'interesse è anche `PF_INET6` per cose IPv6. Se non sai cos'è, non preoccupartene...ancora).

`type`

Inoltre, il parametro *`type`* può essere diverse cose, probabilmente lo imposterai a `SOCK_STREAM` per socket TCP affidabili (**`send()`**, **`recv()`**) o a `SOCK_DGRAM` per socket UDP non affidabili (**`sendto()`**, **`recvfrom()`**).

(Un altro tipo di socket interessante è `SOCK_RAW` che può essere usato per costruire manualmente dei pacchetti. E' piuttosto figo).

`protocol`

Infine, *`protocol`* dice quale protocollo usare per un certo tipo di socket. Come ho già detto, ad esempio, `SOCK_STREAM` usa TCP. Fortunatamente per te, quando usi `SOCK_STREAM` o `SOCK_DGRAM`, puoi impostare il protocollo a 0, e userà quello appropriato automaticamente. Altrimenti, puoi utilizzare **`getprotobyname()`** per cercare il numero di protocollo adatto.

Valore di Ritorno

Un nuovo descrittore di socket che può essere usato in successive chiamate, o `-1` su errore (ed **`errno`** sarà impostato di conseguenza).

Esempio

```
int s1, s2;

s1 = socket(PF_INET, SOCK_STREAM, 0);
s2 = socket(PF_INET, SOCK_DGRAM, 0);

if (s1 == -1) {
    perror("socket");
}
```

Vedi Anche

`accept()`, **`bind()`**, **`listen()`**

8.21. `struct sockaddr_in`, `struct in_addr`

Strutture per maneggiare indirizzi internet

```
#include <netinet/in.h>

struct sockaddr_in {
    short            sin_family;   // es. AF_INET
    unsigned short   sin_port;     // es. htons(3490)
    struct in_addr   sin_addr;     // vedi struct in_addr, below
    char             sin_zero[8];  // imposta a zero se vuoi
};

struct in_addr {
    unsigned long s_addr;  // carica con inet_aton()
};
```

Descrizione

Queste sono le strutture basilari di tutte le syscall e funzioni che trattano indirizzi internet. Ricordo che `struct sockaddr_in` ha la stessa dimensione di `struct sockaddr`, e puoi liberamente fare cast di un puntatore di un tipo ad un altro senza pericolo, eccetto la possibile fine dell'universo.

Sto solo scherzando sulla fine dell'universo...se l'universo finisce se fai un cast da `struct sockaddr_in*` a `struct sockaddr*`, ti giuro che è pura coincidenza e non dovresti esserne preoccupato.

Quindi, con questo in mente, ricorda che ogni volta che una funzione dice di prendere una `struct sockaddr*` puoi eseguire un cast della tua `struct sockaddr_in*` a quel tipo facilmente e in sicurezza.

C'è anche questo campo `sin_zero` che alcuni sostengono deve essere impostato a zero. Altri non affermano niente a riguardo (la documentazione di Linux non lo menziona neanche), settarlo a zero non sembra essere veramente necessario. Per cui, se ne hai voglia, impostalo a zero con **`memset()`**.

Adesso, questa `struct in_addr` è una strana bestia in diversi sistemi. Talvolta è una pazza `union` con tutti i tipi di `#define` e altre sciocchezze, ma quello che dovresti fare è usare solo il campo *`s_addr`* di questa struttura, poiché molti sistemi implementano solo quella.

Con IPv4 (quello che fondamentalmente chiunque nel 2005 usa ancora), `struct s_addr` è un numero a quattro byte che rappresenta una cifra in un indirizzo IP in un byte. (Non vedrai mai un indirizzo IP con un numero maggiore di 255).

Esempio

```
struct sockaddr_in myaddr;
int s;

myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(3490);
inet_aton("10.0.0.1", &myaddr.sin_addr);

s = socket(PF_INET, SOCK_STREAM, 0);
bind(s, (struct sockaddr*)&myaddr, sizeof myaddr);
```

Vedi Anche

`accept()`, **`bind()`**, **`connect()`**, **`inet_aton()`**, **`inet_ntoa()`**

9. Riferimenti

Sei arrivato così lontano, e ora strilli per averne di più! Dove altro puoi andare per imparare di più di tutta questa roba?

9.1. Libri

Per dei tascabili economici di vecchia scuola, prova uno di questi eccellenti libri. Ero solito essere affiliato ad un popolare venditore di libri online, ma il loro nuovo sistema di ricerca di nuovi clienti è incompatibile con un documento stampato. In quanto tale, non ho nessuna percentuale. Se hai compassione per la mia sventura, fai una donazione con paypal a `beej@beej.us`. `:-)`

Unix Network Programming, volumes 1-2 di W. Richard Stevens. Pubblicato da Prentice Hall. ISBN dei volumi 1-2: 0131411551, 0130810819.

Internetworking with TCP/IP, volumes I-III di Douglas E. Comer e David L. Stevens. Pubblicato da Prentice Hall. ISBNs for volumes I, II, and III: 0131876716, 0130319961, 0130320714.

TCP/IP Illustrated, volumes 1-3 di W. Richard Stevens e Gary R. Wright. Pubblicato da Addison Wesley. ISBNs dei volumi 1, 2, e 3 (e il set 3 volumi): 0201633469, 020163354X, 0201634953, (0201776316).

TCP/IP Network Administration di Craig Hunt. Published di O'Reilly & Associates, Inc. ISBN 0596002971.

Advanced Programming in the UNIX Environment di W. Richard Stevens. Pubblicato da Addison Wesley. ISBN 0201433079.

9.2. Riferimenti Web

Sul web:

BSD Sockets: A Quick And Dirty Primer (Info anche sulla programmazione di sistema!)

The Unix Socket FAQ

Intro to TCP/IP

TCP/IP FAQ

The Winsock FAQ

Ed ecco alcune rilevanti pagine Wikipedia:

Berkeley Sockets

Internet Protocol (IP)

Transmission Control Protocol (TCP)

User Datagram Protocol (UDP)

Client-Server

Serialization (impacchettare e spacchettare i dati)

9.3. RFC

RFC—le reali basi:

RFC 768—The User Datagram Protocol (UDP)

RFC 791—The Internet Protocol (IP)

RFC 793—The Transmission Control Protocol (TCP)

RFC 854—The Telnet Protocol

RFC 951—The Bootstrap Protocol (BOOTP)

RFC 1350—The Trivial File Transfer Protocol (TFTP)

RFC 4506—External Data Representation Standard (XDR)

IETF ha un buon tool online per cercare e sfogliare RFC.

Indice