Socket Programming HOWTO

Release 0.00

Gordon McMillan

12 dicembre 2003

gmcm@hypernet.com

Sommario

I socket hanno vasta diffusione, ma restano una delle tecnologie meno comprese. Questo documento è solo una panoramica sui socket. Non è un vero e proprio tutorial - dovrete ancora lavorare parecchio per far funzionare le cose. Non si occupa delle sottigliezze (e ve ne sono parecchie), ma spero che vi dia conoscenze sufficienti per iniziare a usarli decentemente. Traduzione italiana a cura di Giorgio Zoppi (deneb at penguin.it) e Riccardo Fabris (python.it at tiscalinet.it).

Questo documento è disponibile, in lingua originale, presso la pagina dei Python HOWTO http://www.python.org/doc/howto, la traduzione presso la pagina http://www.zonapython.it/doc/howto/.

Indice

1 I socket

I socket hanno vasta diffusione, ma restano una delle tecnologie meno comprese. Questo documento è solo una panoramica sui socket. Non è un vero e proprio tutorial - dovrete ancora lavorare parecchio per far funzionare le cose. Non si occupa delle sottigliezze (e ve ne sono parecchie), ma spero che vi dia conoscenze sufficienti per iniziare a usarli decentemente.

Ho intenzione di trattare solo i socket INET, che comunque sono stimati essere almeno il 99% dei socket in uso. E parlerò dei socket STREAM: a meno che voi non sappiate veramente quello che fate (in tal caso questo HOWTO non è per voi!), otterrete comportamento e prestazioni migliori da un socket STREAM che da qualsiasi altro. Proverò sia a risolvere il mistero di cosa sia un socket che a dare alcuni suggerimenti su come lavorare con i socket bloccanti e non bloccanti. Ma inizierò a parlare dei socket bloccanti. È necessario che voi sappiate come funzionano prima di trattare quelli non bloccanti.

Parte del problema nel comprendere la questione è che il termine “socket” può significare una quantità di cose sottilmente differenti, a seconda del contesto. Quindi innanzitutto facciamo una distinzione tra un socket “client” - l’estremo di una conversazione, e un socket “server”, che è più simile ad un operatore di centralino. L’applicazione client (il vostro browser, p.e.) usa esclusivamente socket “client”; il server web con il quale sta conversando usa sia socket “server” sia “client”.

1.1 Storia

Tra le varie forme di IPC (*Inter Process Communication* - comunicazione tra processi), i socket sono di gran lunga la più popolare. Data una qualsiasi piattaforma, è probabile che ci siano altre forme di IPC più veloci, ma per la comunicazione tra piattaforme diverse i socket sono quasi una scelta obbligata.

Furono inventati a Berkeley come parte dello UNIX BSD. Si diffusero assai rapidamente con Internet. Per buone ragioni la combinazione dei socket con INET rende la comunicazione con macchine di qualunque tipo sparse qua e là per il mondo incredibilmente facile (almeno se comparata con gli altri sistemi).

2 Creare un socket

Parlando per sommi capi, quando voi avete cliccato sul link che vi ha portato a questa pagina, il vostro browser ha fatto qualcosa del tipo:

```
# crea un socket INET di tipo STREAM
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
# ora si connette al server web sulla porta 80
#  - la normale porta http
s.connect(("www.mcmillan-inc.com", 80))
```

Quando la connessione (`connect`) è stabilita, il socket `s` può essere usato per inoltrare una richiesta del testo di questa pagina. Lo stesso socket leggerà la risposta, e sarà poi distrutto. È giusto: distrutto. I socket client sono normalmente usati per un solo scambio (o un piccolo insieme di scambi sequenziali).

Quello che succede nel server web è un po’ più complesso. Prima, il server web crea un “socket server”.

```
# crea un socket INET di tipo STREAM
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
# associa il socket a un host pubblico
# e a una delle porte ben-note
serversocket.bind((socket.gethostname(), 80))
# diventa un socket server
serversocket.listen(5)
```

Un paio di cose da notare: abbiamo usato `socket.gethostname()` cosicché il socket sia visibile al mondo esterno. Se avessimo usato `serversocket.bind((”, 80))` o `serversocket.bind((’localhost’, 80))` o `serversocket.bind((’127.0.0.1’, 80))` avremmo avuto ancora un socket “server”, ma visibile solo all’interno della stessa macchina.

Seconda cosa da notare: le porte con un numero basso sono di solito riservate per servizi “ben noti” (HTTP, SNMP, ecc). Se state facendo esperimenti, usate un numero piuttosto alto (almeno 4 cifre).

Infine, il parametro passato a `listen` dice alla libreria socket che noi vogliamo che si mettano in coda 5 richieste di connessione (il massimo, normalmente) prima di rifiutare connessioni esterne. Se il resto del codice è scritto in maniera adeguata, dovrebbero essere sufficienti.

Bene, ora abbiamo un socket “server”, in ascolto sulla porta 80. Ora introduciamo il ciclo principale del server web:

```
while 1:
    # accetta le connessioni dall'esterno
    (SocketClient, address) = serversocket.accept()
    # ora fa qualcosa con il socket client
    # in questo caso, fingiamo che sia un server che usa i thread
    ct = client_thread(SocketClient)
    ct.run()
```

Ci sono in realtà 3 modi comuni per far funzionare questo ciclo: smistare il `SocketClient` a un thread che lo gestisca, creare un nuovo processo per gestire il `SocketClient`, o ristrutturare questa applicazione per usare i socket non bloccanti, e lavorare in "multiplexing" tra il nostro socket "server" e un qualunque `SocketClient` attivo usando `select`. Ne parleremo più avanti. La cosa importante da capire ora è che questo è *tutto* quello che un socket "server" fa. Non invia nessun dato. Non riceve nessun dato. Produce solo socket "client". Ciascun `SocketClient` è creato in risposta a qualche *altro* socket "client" che fa un `connect()` a host e porta ai quali siamo associati. Non appena abbiamo creato quel `SocketClient`, torniamo a restare in attesa di ulteriori connessioni. I due "client" sono liberi di continuare a conversare; stanno usando delle porte allocate dinamicamente, che saranno riciclate quando la conversazione sarà finita.

2.1 IPC

Se avete bisogno di IPC veloce tra due processi su un'unica macchina, dovreste esaminare a fondo qualsiasi forma di memoria condivisa che la piattaforma offre. Un semplice protocollo basato su memoria condivisa e lock o semafori, è di gran lunga la tecnica più veloce.

Se decidete di usare i socket, associate il socket "server" a `'localhost'`. Sulla maggior parte delle piattaforme, questa scorciatoia permetterà di eludere un paio di strati del codice di rete e si acquisterà in velocità.

3 Usare un socket

La prima cosa da notare è che il socket "client" del browser e il socket "client" del server web sono la stessa bestia. Cioè questa è una conversazione da pari a pari ("peer to peer"). O, per metterla in altro modo, *come progettisti, dovete decidere quali sono le regole di etichetta per una conversazione*. Normalmente, il socket che si `connette` inizia la conversazione, inviando una richiesta o forse un segnale di connessione. Ma questa è una decisione a livello progettuale, non è una regola dei socket.

Ora ci sono due insiemi distinti di verbi da usare per la comunicazione. Potete usare `send` e `recv` [sta per 'receive' NdT], o potete trasformare il vostro socket client in una cosa simile a un file e usare `read` e `write`. L'ultimo è il modo in cui Java presenta i propri socket. Non ho intenzione di parlarne qui, eccetto che per avvisarvi che avete bisogno di usare `flush` sui socket. Sono "file" bufferizzati, e un errore comune è `scrivere` qualcosa, poi `leggere` per avere una risposta. Senza un `flush` potreste aspettare una risposta all'infinito, perché la richiesta potrebbe essere ancora nel vostro buffer di uscita.

Ora veniamo allo scoglio maggiore che si deve affrontare coi socket: `send` e `recv` operano sui buffer di rete. Non necessariamente gestiscono tutti i byte che passate loro (o che aspettate da loro), in quanto il loro scopo principale è gestire i buffer di rete. In generale ritornano quando i buffer di rete ad essi associati sono stati riempiti (`send`) o svuotati (`recv`). Poi vi dicono quanti byte hanno gestito. È *vostra* responsabilità chiamarli di nuovo finché il vostro messaggio non sia stato completamente trattato.

Quando `recv` restituisce 0 byte, significa che l'altro lato ha chiuso la connessione (o ne sta effettuando la chiusura). Non riceverete più dati su questa connessione. Mai. Potreste comunque essere in grado di inviare dati con successo; parlerò di questo più avanti.

Un protocollo come HTTP usa un socket per un unico trasferimento. Il client manda una richiesta e poi legge una risposta. È tutto. Il socket viene abbandonato. Ciò significa che un client può accorgersi della fine della risposta ricevendo 0 byte.

Ma se pianificate di riusare il vostro socket per ulteriori trasferimenti, dovete rendervi conto che *non esiste una cosa come un "EOT" (End of Transfer - Fine del Trasferimento) su un socket.* Ripeto: se `send` o `recv` di un

socket ritorna dopo aver gestito 0 byte, la connessione è stata interrotta. Se la connessione *non* è stata interrotta, aspetterete un `recv` all'infinito, in quanto il socket *non* vi dirà che non c'è più niente da leggere (per ora). Ora, se ci pensate su un po', arriverete a comprendere una fondamentale verità sui socket: *i messaggi devono essere di una "determinata" lunghezza* (sigh!), *o essere delimitati* (tzè...!?), *o indicare quanto sono lunghi* (molto meglio!), *o finire facendo cadere la connessione*. La scelta è interamente vostra, (ma alcune strade sono più giuste di altre).

Assumendo che voi non vogliate terminare la connessione, la soluzione più semplice è un messaggio di lunghezza fissa:

```
class mysocket:
    '''classe solamente dimostrativa
      - codificata per chiarezza, non per efficenza'''
    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock
    def connect(host, port):
        self.sock.connect((host, port))
    def mysend(msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError, \\
                    "connessione socket interrotta"
            totalsent = totalsent + sent
    def myreceive():
        msg = ''
        while len(msg) < MSGLEN:
            chunk = self.sock.recv(MSGLEN-len(msg))
            if chunk == '':
                raise RuntimeError, \\
                    "connessione socket interrotta"
            msg = msg + chunk
        return msg
```

Il codice di invio di questo esempio è utilizzabile per quasi ogni schema di scambio di messaggi - in Python si inviano stringhe, e si può usare `len()` per determinare la loro lunghezza (anche se contengono caratteri `\0` interni). Di solito è il codice per la ricezione a essere più complesso (e in C non è molto peggio, eccetto che non si può usare `strlen` se il messaggio contiene al suo interno degli `\0`).

Il miglioramento più semplice da apportare è rendere il primo carattere del messaggio un indicatore del tipo di messaggio, il tipo ne determina la lunghezza. Ora avete due `recv`, il primo per ottenere (almeno) quel primo carattere, così da poter sapere rapidamente la lunghezza, e il secondo in un ciclo per ottenere il resto. Se scegliete la strada dei messaggi delimitati, vi troverete a ricevere spezzoni di lunghezza arbitraria (4096 o 8192 trovano di frequente buona corrispondenza nelle dimensioni dei buffer di rete), e analizzerete ciò che avete ricevuto alla ricerca di un delimitatore.

Una complicazione cui fare attenzione: se il vostro protocollo di conversazione permette che messaggi multipli vengano mandati uno di seguito all'altro (senza un qualche tipo di risposta nel mezzo), e `ricevete` spezzoni di lunghezza arbitraria, potreste finire col leggere l'inizio di un messaggio successivo. Dovete metterlo da parte e tenerlo in sospeso fino a che non sia necessario.

Preporre al messaggio la sua lunghezza (per dire, 5 caratteri numerici) diventa piú complesso, perché (credeteci o no), potreste non ottenere tutti i 5 caratteri con un solo `recv`. Nei vostri esperimenti potete fare a meno di pensarci, ma in caso di elevati carichi di rete il vostro codice finirebbe ben presto per collassare, a meno che non usiate due cicli `recv` - il primo per determinare la lunghezza, il secondo per ottenere la sezione dati del messaggio. Disgustoso. Questo vale anche per quando scoprirete che `send` non sempre riesce a liberarsi di tutto in un solo passaggio. E malgrado lo abbiate letto, alla fine non vi servirà a molto!

Per risparmiare spazio e per rendervi forti nelle avversità (e mantenere la mia posizione privilegiata), tali miglioramenti sono lasciati come esercizi per il lettore. Diamoci una mossa per finire.

3.1 Dati binari

È perfettamente possibile inviare dati binari su un socket. Il problema maggiore è che non tutte le macchine usano gli stessi formati per i dati binari. Per esempio, un chip Motorola rappresenta un intero a 16 bit di valore pari a 1 con due byte in esadecimale 00 01 [il cosiddetto 'big-endian' NdT]. Intel e DEC, invece, usano invertire l'ordine dei byte [il cosiddetto 'little-endian' NdT] - cio è lo stesso 1 di prima è 01 00. Le librerie socket posseggono chiamate per convertire gli interi a 16 e 32 bit: `ntohl, htonl, ntohs, htons` dove "n" significa *network* e "h" significa *host*, "s" significa *short* e "l" significa *long*. Dove l'ordine di rete è l'ordine di host queste funzioni non fanno niente, ma dove la macchina usa invertire l'ordine dei byte, queste funzioni scambiano tra di loro i byte in maniera appropriata.

In questi tempi di macchine a 32 bit, la rappresentazione ascii dei dati binari occupa di frequente meno spazio di quella binaria. Questo perché un numero sorprendente di volte tanti long hanno un valore 0, oppure 1. "0" come stringa occupa due byte, mentre come dato binario ne occupa quattro. Certamente è una cosa che non va molto d'accordo coi messaggi di lunghezza fissata. Decisioni, decisioni.

4 Sconnettersi

A rigor di termini, si suppone usiate `shutdown` su un socket prima di chiuderlo con `close`. Lo `shutdown` è un avvertimento al socket all'altro capo. A seconda dall'argomento che gli passate, può significare "non intendo più inviare, ma rimango ancora in ascolto", o "non sto ascoltando, che sollievo!". La maggior parte delle librerie socket, tuttavia, si sono talmente adattate all'abitudine dei programmatori di trascurare questa fase del cerimoniale che di norma un `close` è la stessa cosa di uno `shutdown(); close()`. Quindi nella maggior parte delle situazioni, uno `shutdown` esplicito non è necessario.

Un modo per usare efficacemente `shutdown` è in uno scambio stile HTTP. Il client manda una richiesta e poi fa uno `shutdown(1)`. Questo dice al server "Questo client ha finito l'invio, ma può ancora ricevere". Il server può rilevare "EOF" da un "receive" di 0 byte. Può assumere di aver ricevuto la richiesta per intero. Il server invia una risposta. Se il `send` è completato con successo allora di certo il client stava ancora ricevendo.

Python porta lo shutdown automatico un passo più in là: quando un socket finisce in garbage collection, esso farà automaticamente un `close` se necessario. Ma farci affidamento è una pessima abitudine. Se il vostro socket semplicemente sparisce senza fare un `close`, il socket all'altro capo potrebbe rimanere in sospeso a tempo indefinito, ritenendo che voi siate semplicemente lenti. Quindi *per favore* fate un bel `close` sui vostri socket quando avete finito.

4.1 Quando i socket muoiono

Probabilmente la cosa peggiore coi socket bloccanti è quando l'altro capo va giù di brutto (senza un `close`). Dopo di ciò è probabile che il vostro socket rimanga bloccato in sospeso. SOCKSTREAM è un protocollo affidabile, e aspetterà molto, molto tempo prima di mollare una connessione. Se state usando i thread, l'intero thread è di fatto morto. Non c'è molto che possiate fare. Fino a quando non farete qualcosa di stupido, come mantenere un lock mentre state facendo una lettura bloccante, il thread non consumerà molte risorse in verità. *Non* provate a uccidere il thread - parte della ragione per la quale i thread sono più efficenti rispetto ai processi è che evitano l'overhead associato con il riciclo automatico delle risorse. In altre parole, se provate a uccidere il thread è probabile che il vostro intero processo venga fregato.

5 Socket non bloccanti

Se avete capito tutto fino a questo punto, ormai saprete già la maggior parte di quello che vi serve sapere sui meccanismi di utilizzo dei socket. Userete ancora le stesse chiamate, perlopiù negli stessi modi. È solo che, se lo fate bene, la vostra applicazione sarà pressoché rivoltata.

In Python usate `socket.setblocking(0)` per rendere il socket non bloccante. In C è più complesso (per una cosa, avrete bisogno di scegliere tra lo stile `O_NONBLOCK` e il quasi indistinguibile stile `O_NDELAY`, che è completamente differente da `TCP_NODELAY`), ma l'idea è esattamente la stessa. Fatelo dopo aver creato il socket, ma prima di usarlo (in realtà se siete pazzi potete scattare avanti e indietro).

La principale differenza a livello di codice è che `send`, `recv`, `connect` e `accept` possono ritornare senza aver fatto nulla. Avete (certamente) un buon numero di scelte possibili. Potete verificare il codice di ritorno e i codici di errore e in genere questo vi farà ammattire. Se non mi credete, provateci qualche volta. La vostra applicazione diventerà enorme, piena di bachi ed esosa in termini di risorse. Quindi tralasciamo le soluzioni idiote e facciamo le cose per bene.

Usiamo `select`.

In C, scrivere codice per `select` è abbastanza complesso. In Python è liscio come il burro, ma è abbastanza simile a quanto si fa in C, capendo l'uso di `select` in Python avrete pochi problemi in C.

```
pronti_da_leggere, pronti_da_scrivere, in_errore = \\
                select.select(
                   letture_potenziali,
                   scritture_potenziali,
                   errori_potenziali,
                   timeout)
```

Potete passare a `select` tre liste: la prima contiene tutti i socket che vorreste provare a leggere, la seconda tutti i socket che vorreste provare a scrivere e l'ultima (normalmente lasciata vuota) quelli che vorreste controllare per eventuali errori. Dovreste notare che un socket può essere presente in più di una lista. La chiamata `select` è bloccante, ma potete darle un timeout. Questo è generalmente una cosa sensata da fare - datele un bel timeout lungo (diciamo un minuto) a meno che non abbiate una buona ragione per fare altrimenti.

La funzione restituirà tre liste, che saranno composte dai socket effettivamente leggibili, scrivibili e in errore. Ciascuna di queste liste sarà un sottoinsieme (possibilmente vuoto) della corrispondente lista che avete passato. E se mettete un socket in più di una lista in ingresso, esso sarà presente al più solo in una delle liste in uscita.

Se un socket è nella lista in uscita dei socket leggibili, potete essere sicuri-quanto-più-non-si-potrebbe-in-tale-ambito che un `recv` restituirà *qualcosa*. Lo stesso per la lista dei socket scrivibili: sarete in grado di inviare *qualcosa*. Forse non tutto quello che volete, ma qualcosa è meglio di niente. In realtà qualsiasi socket ragionevolmente robusto verrà restituito nella lista dei socket scrivibili, esserci significa solo che c'è spazio nel buffer di rete in uscita.

Se avete un socket "server" mettetelo nella lista letture_potenziali. Se compare nella corrispondente lista in uscita, il vostro `accept` (quasi certamente) funzionerà. Se avete creato un nuovo socket per `connettervi` a qualcun altro, mettetelo nella lista scritture_potenziali. Se comparirà nella lista in uscita, avrete una garanzia decente dell'avvenuta connessione.

Un problema antipatico con `select`: se nelle liste in ingresso c'è un socket che è morto di una brutta morte, `select` fallirà. Avrete quindi bisogno di verificare in un ciclo ogni singolo dannato socket presente nelle liste con un `select([sock],[],[],0)` fino a trovare il responsabile. Il timeout a 0 significa che non ci metterà molto, ma resta un orrore.

In realtà `select` può essere utile anche con i socket bloccanti. È un modo per determinare se li bloccherete - il socket verrà restituito come leggibile se c'è qualcosa nei buffer. Tuttavia non è comunque d'aiuto col problema di determinare se l'altro capo ha finito o è solo occupato altrove.

Avviso di portabilità: su UNIX `select` funziona sia coi socket che coi file. Non provateci su Windows. Su Windows `select` funziona solo coi socket. Notate anche che in C molte opzioni avanzate dei socket sono gestite in modo diverso sotto Windows. Infatti su Windows io di solito uso i thread (che funzionano molto, molto bene) per i miei socket. Se desiderate prestazioni decenti dovete affrontare il problema: il vostro codice per Windows sarà molto diverso da quello per UNIX. (Non ho la minima idea di come affrontare la questione su Mac).

5.1 Prestazioni

Non c'è dubbio che il codice socket più prestante utilizza socket non bloccanti e select per gestirli in multiplexing. Potete mettere insieme qualcosa in grado di saturare una connessione LAN senza troppo sforzo per la CPU. Il guaio è che una applicazione scritta in questo modo non potrà fare molto altro - ha bisogno di essere pronta in ogni momento a smistare in giro byte.

Posto che la vostra applicazione debba in realtà servire a qualcosa di meglio, usare i thread è la soluzione ottimale, (e usando i socket non bloccanti sarete più veloci che usando i socket bloccanti). Sfortunatamente il supporto ai thread nei vari UNIX varia sia nell'interfaccia che per qualità. Quindi la soluzione normale in UNIX è fare il fork di un sottoprocesso per occuparsi di ogni singola connessione. Il carico accessorio di elaborazione ("overhead") è però significativo (e non fatelo sotto Windows, dove sarebbe enorme). Significa anche che nel caso ogni sottoprocesso non sia completamente indipendente avrete bisogno di usare un'altra forma di comunicazione tra processi, per dire una pipe, o memoria condivisa e semafori, per la comunicazione tra processi genitore e figli.

Infine ricordate che, anche se i socket bloccanti sono alquanto più lenti in confronto ai non bloccanti, in molti casi sono la soluzione "corretta". Dopo tutto, se la vostra applicazione è pilotata dai dati che riceve su un socket, non è molto sensato complicarne la logica di programmazione per farla rimanere in attesa su una `select` piuttosto che su una `recv`.