Avevamo accennato i SOCKET come comitiva di comunicazione utilizzabile da un’applicazione.

L’idea è di avere dei processi che mandiamo in esecuzione, tali processi possono aprire dei “buchi” dai quali poter far entrare e uscire sequenze di byte. Il funzionamento è analogo a quello dei file descriptor per accedere al file system: la differenza è che la system call per creare il descriptor in questo caso è “socket()”(fd = socket()).

Il file descriptor è necessario perché il processo non ha i mezzi per uscire dalla propria scatola virtuale, ma il sistema operativo, tramite i suoi privilegi, sì (per questo si usano le system call).

Nei sistemi POSIX sono realizzati due tipi principali di socket: quelli di tipo “DATAGRAM” e quelli di tipo “STREAM”.  
  
I Socket di tipo Datagram permettono di ricevere e inviare messaggi, che vengono chiamati datagrammi. Un datagramma è una certa quantità di byte che contiene delle informazioni codificate in qualche modo (da parte dell’applicazione, che seguirà una certa convenzione in modo che si possano poi decifrare). Questi datagrammi, quindi, oltre al payload avranno delle intestazioni (gli headers) che dipendono dal protocollo usato (cioè dal suo livello). Si ricorda che le intestazioni contengono gli indirizzi, mentre il payload è il messaggio.

Con una struttura di tipo Datagram il dispositivo A può indicare un’applicazione di destinazione B e noi non ci preoccupiamo di come sia fatto il meccanismo di trasmissione.  
Le system call utilizzabili sul file descriptor di tipo socket sono la “send(fd, …)” che serve per inviare le informazioni, e contiene un file descriptor, (e, in genere) un puntatore all’area di memoria contenente i dati che voglio inviare e la quantità di byte che vogliamo inviare, e la “recv(fd, …)” che serve per ricevere i dati.

Il protocollo di tipo DATAGRAM, nella sua implementazione classica, ha le stesse caratteristiche dei protocolli che si usano al livello 3 (quindi il protocollo IP), dunque nell’header ci sarà la rappresentazione di un numero su 16 bit che rappresenterà la lunghezza effettiva del payload. Chi vuole inviare deve inoltre specificare un buffer (tramite un puntatore a un’area di memoria).

Esempio: char buf[1024]; send(fd, buf, 1024);

Dalla parte del ricevente, bisogna fare una cosa analoga: serve aver dichiarato un buffer di lunghezza adeguata per ricevere il messaggio ed effettuare la ricezione passando come parametro un puntatore all’area di memoria e la lunghezza del messaggio. Ma come fa il ricevente a sapere di quanti byte è il messaggio? Di base non può saperlo, anche in caso di comunicazione a priori ci potrebbero essere dei fraintendimenti. Se arrivassero meno byte del previsto, la recv restituirebbe come valore il numero di byte effettivamente scritti nel buffer (lo fa comunque, ma è significativo solo se è un numero diverso da quello che ci aspettavamo). Se arrivassero più byte del previsto, verrebbero letti e scritti solo i primi x byte (dove x è la dimensione del buffer di ricezione) e tutti i restanti verrebbero buttati via.

La send e la recv restituiscono sempre un valore, che equivale al numero di byte effettivamente inviati/letti. Possono restituire anche il valore -1, che indicherebbe operazione fallita; a quel punto per sapere per quale motivo l’operazione è fallita sarebbe necessario accedere alla variabile errno.

La modalità DATAGRAM NON GARANTISCE IL RECAPITO DI TUTTO IL MESSAGGIO, è una trasmissione NON affidabile: ci sono situazione in cui sarebbe complicato far sapere al destinatario quante informazioni gli arriveranno, dunque il protocollo datagram è come un mettere le mani avanti e ammettere già la possibilità che non arrivino tutte.

Se si volesse mandare un datagramma a più utenti, risulterebbe comodo avere una variante che permetta di specificare più indirizzi diversi. In effetti è così, oltre alla send, esiste la system call “send\_to()” che oltre alle informazioni della send permette di inserire un indirizzo da usare solo per quel messaggio (poiché altrimenti viene utilizzato l’indirizzo inserito all’apertura della socket, e quindi il buco servirebbe per comunicare con un solo utente). Se si volesse associare il socket a un solo destinatario si può usare la system call “connect()”, a quel punto si potrebbe usare la send (altrimenti bisogna usare per forza la send\_to).   
Esiste anche la system call “recv\_from()”, che serve per avere come parametro in output l’indirizzo del mittente, oltre la lunghezza del messaggio: infatti l’indirizzo è ottenuto passando un parametro per riferimento.

Riassumendo, la modalità Datagram si serve dei protocolli UDP / IP, le caratteristiche sono analoghe a quelle del protocollo IP (l’indirizzo di mittente e destinatario sono quelle specificate nell’indirizzo IP, mentre le porte sono quelle specificate nel protocollo UDP -> La porta però sta a livello di trasporto e l’indirizzo a livello di rete).

Passiamo adesso alla versione STREAM. La versione Stream dei socket è molto più complicata da implementare, ma è più semplice da usare. Esso si basa sul protocollo di trasporto TCP / IP. La complicatezza nella realizzazione del protocollo ha infatti il vantaggio di assicurare l’arrivo di tutte le informazioni al destinatario.

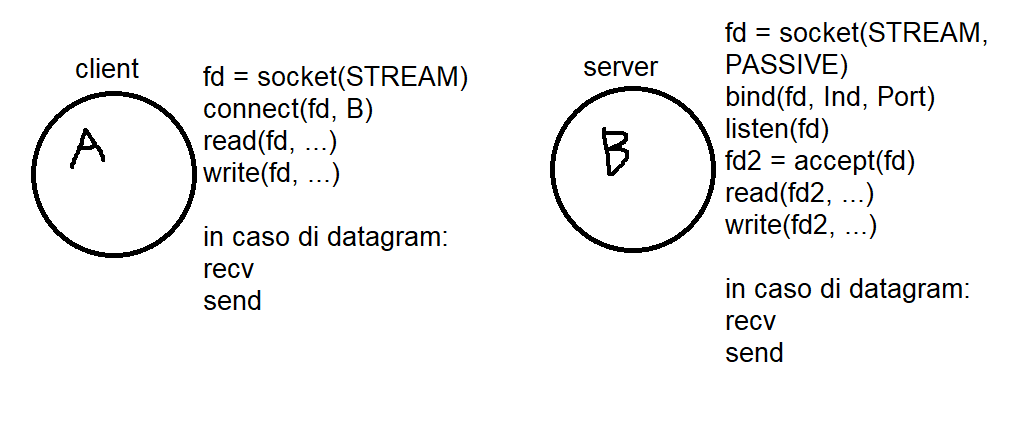
Inviare un messaggio in modalità stream è come scrivere una sequenza di byte all’interno di un file, mentre leggerlo è come leggere il contenuto di un file. Non si sa quindi quanti byte sono stati inviati in totale.

Esempio: Alice potrebbe aver fatto 3 send, due da mille byte e uno da cinquecento. Bob però ne vuole leggere 1500, Bob legge quindi i 1500 byte che voleva, ma i 1000 che non ha ancora letto non vanno scartati, rimangono leggibili per Bob in un momento successivo. Analogamente se Bob ne volesse leggere di più, leggerebbe solo quelli che gli sono arrivati, ma se Alice facesse un’altra send gli arriverebbero anche i nuovi byte. Lo Stream è quindi come un tubo, in cui passano le informazioni.

Siccome la modalità di funzionamento è simile a quella di lettura e scrittura di un file, per i socket di tipo STREAM funzionano le system call di tipo read e write (il programma vede il socket di tipo stream come un file, anzi due file, a tutti gli effetti -> un file è letto da uno e scritto dall’altro e per l’altro file viceversa).  
Però, per i socket di tipo stream, a differenza di quelli di tipo datagram, è necessario sempre l’utilizzo della system call “connect()”(poiché è l’unico modo per specificare l’indirizzo del destinatario): dunque i socket di tipo STREAM permettono la comunicazione con un solo dispositivo per socket (NON è neanche permesso di chiamare più volte la connect, mentre per i protocolli di tipo DATAGRAM è permesso).

Quando si ha finito di usare un socket lo si può distruggere (/chiudere) per recuperare risorse nel sistema.

Poiché chi manda e riceve usano istruzioni/codice diverse (così vuole il protocollo), si definisce come CLIENT chi prende l’iniziativa, mentre come SERVER chi subisce. Il SERVER è dunque un’unità passiva, mentre il CLIENT è un’unità attiva.  
Poiché il server si deve connettere a un dispositivo su una particolare porta, per poter specificare su quale porta l’applicazione sta in ascolto bisogna usare la system call “bind(fd, indirizzo, porta)” (l’indirizzo si riferisce alla macchina stessa che “hosta” il server). Dopodiché, il server chiama la system call “listen(fd)”, con la quale esso dichiara al sistema operativo di essere pronto a ricevere la richiesta di connessione. Fatto ciò, e quindi ricevuto l’OK dal sistema operativo, il processo del server chiama la system call “accept(fd)”, con la quale il server resta in attesa finché non arriva una richiesta di connessione da un client. Quando tale richiesta avviene, la connessione viene stabilita e si può procedere con lo scambio di informazioni. Una cosa a cui bisogna prestare attenzione è che l’accept ritorna un file descriptor, che è il descriptor che permette di comunicare col client: dunque per poter scrivere e ricevere messaggi dal client, il file a cui il server dovrà accedere sarà quello il cui descriptor è ritornato dall’accept (fd2).



Tutto ciò permette al Server di connettersi a qualsiasi client, basta che questi abbia l’indirizzo ip e la porta tcp giusta. Il Server dal canto suo, con ogni accept (le altre 3 operazioni per creare il socket possono/devono essere fatte una volta sola) con un client diverso otterrà un nuovo file descriptor che gli permetterà di comunicare anche col nuovo utente. La parte del server è più complicata perché la sua non è una semplice comunicazione tra 2, con questo set up più generale può infatti connettersi anche a 50 dispositivi diversi contemporaneamente.

Il server in questo caso non può rifiutare la condizione, al massimo si può chiudere il file descriptor (la connessione) appena la si è aperto.

**Veniamo al livello di trasporto**. Partiamo col protocollo TCP perché è più complicato.

Come viene implementato nel protocollo TCP il meccanismo della connessione?

Quando il Client manda la richiesta di connessione, viene inviato un datagramma (che porta la richiesta di connessione). Il server invia la risposta in cui accetta la connessione ( a questo punto il client sa che la connessione c’è) e quando il client lo riceve viene inviato un terzo messaggio per comunicare al server che la connessione è andata a buon fine. Poiché servono 3 messaggi per confermare la connessione, il protocollo viene definito 3-way handshake.

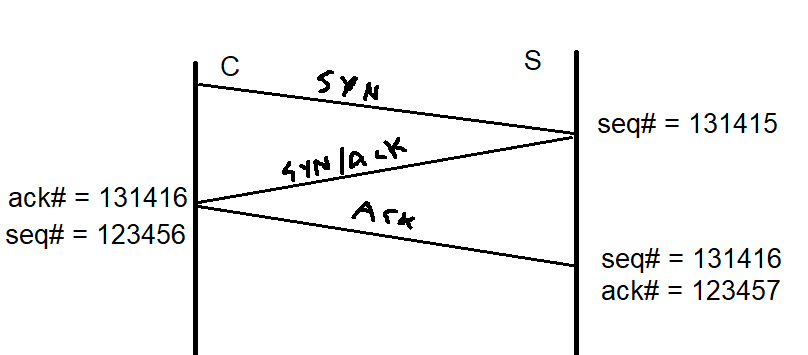
Lo scopo del TCP è quello di aggiungere affidabilità a un protocollo di comunicazione (il datagramma) che di per sé non è affidabile.

Vediamo l’header del protocollo TCP.

Le informazioni contenute sono costituite da due numeri interi e da un certo numero di flag, costituiti da singoli bit. A noi ne interessano 2, il flag syn e il flag ack. I due numeri interi sono espressi su 32 bit e sono chiamati “numero di sequenza” (seq#) e “numero di acknowledgement” (ack#).

Il primo messaggio viene chiamato messaggio syn, perché in questa fase syn assume il valore 1 e ack il valore 0. Il secondo messaggio viene chiamato syn/ack, perché sia syn che ack assumono il valore 1. Dal terzo messaggio in poi solo il bit ack assume il valore 1, mentre syn ha il valore 0 (dunque syn = stiamo stabilendo la connessione, ack = connessione stabilita, quando il primo è 0 e il secondo è 1 entrambi i dispositivi sanno di essere connessi).

I due numeri servono per implementare l’affidabilità del protocollo, come? Nel primo messaggio viene inserito nel seq# un valore casuale e, se il server accetta la connessione legge il numero del seq# e lo riscrive incrementato di 1 nell’ack#, dopodiché il server genera a sua volta un numero casuale indipendente e lo inserisce nello seq#, poi invia la risposta (syn/ack).  
Quando il client riceve la risposta prende nota del ack# del server e lo inserisce nel seq#, mentre mette in ack# il valore di seq# del server +1. È importante che tali numeri siano generati in modo casuale, per ragione di sicurezza (il fatto che client e server condividano seq# e ack# dà conferma della connessione tra i due dispositivi).



I seq# e ack# vengono usati per determinare la posizione dei byte trasmessi nello stream.

Stabilita la connessione il client potrebbe mandare una qualsiasi sequenza di byte. Esso può iniziare a mandare il suo payload nel momento in cui sa che la connessione è andata a buon fine (quindi già dal terzo messaggio). Il seq# rivelerà la posizione all’interno dello stream dei byte inviati. Se come client si volessero mandare ulteriori byte, basterà inviare un secondo messaggio incrementando di x il sequence number (seq#) (dove x è il numero di byte inviati la prima volta). Se per esempio si avessero i seq# e ack# dell’immagine e si inviassero 1000 byte di payload al terzo messaggio, se il client ne mandasse un quarto con altri 1000 byte il sequence number da inviare a quel punto dovrebbe essere 132416.

L’ack# serve invece per mandare degli acknowledgement, è un segnale che viene mandato per indicare “ho ricevuto quello che hai inviato” (essendoci la possibilità di perdere un messaggio l’acknowledgement ci conferma che sia andato tutto OK). Assieme all’acknowledgement c’è un timer che scatta se l’ack# non viene ricevuto per un certo periodo di tempo e se scatta il messaggio viene rinviato. Porco dio.