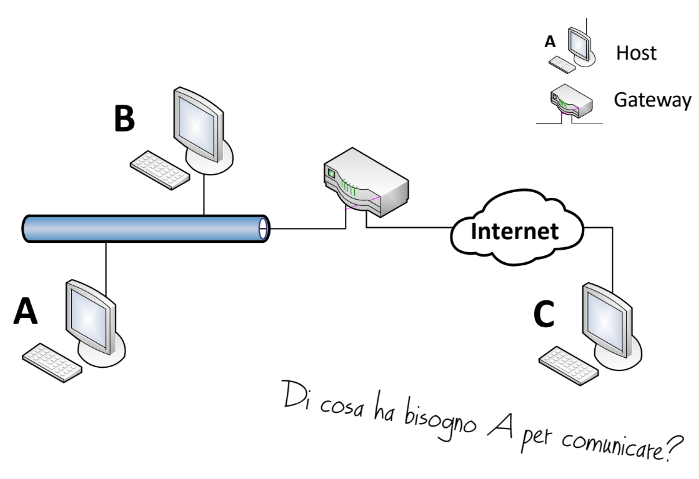
**CONFIGURAZIONE INTERFACCE DI RETE, GATEWAY E DNS** *03-10-22*

Supponiamo di avere il seguente scenario:



Abbiamo tre host, connessi ad Internet. A e B fanno parte della stessa sottorete (sono connessi allo stesso router, detto anche **gateway**), mentre C è connesso ad un’altra sottorete.

Considerando ad esempio l’host A, per comunicare su Internet deve avere 4 cose:

1. Un indirizzo IP;
2. Una maschera di rete;
3. L’indirizzo del gateway;
4. L’indirizzo del server DNS.

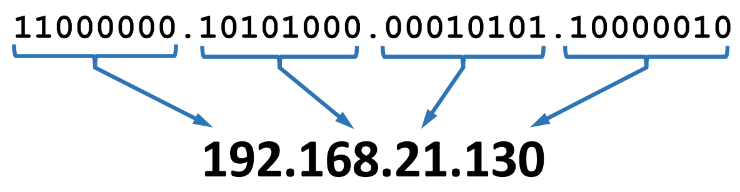
Analizziamo per ora le prime due.

**INDIRIZZO IP**

L’indirizzo IP è un numero a 32 bit che identifica univocamente un’interfaccia di rete (una scheda di rete) in una rete. Un host dunque ha un indirizzo IP solo se è connesso ad una rete, ne ha tanti quanti sono le sue schede di rete connesse ad una qualche rete e, se una scheda si sconnette da una rete e si collega ad un’altra, tale scheda otterrà un indirizzo IP diverso dal precedente (quel *‘in una rete’* indica infatti che l’indirizzo IP dà informazioni anche sulla rete cui il nodo è connesso, quindi cambiando rete, cambierà anche l’indirizzo IP).

Per comodità, gli indirizzi IP si rappresentano in notazione decimale puntata: si dividono i 32 bit in 4 ottetti separati da un punto, e ogni ottetto viene scritto come un numero naturale in base 10.

*Es.*



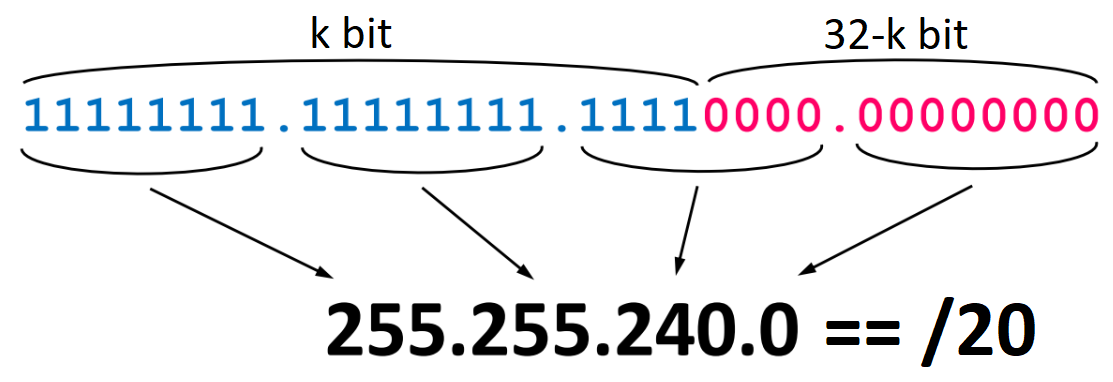
L’indirizzo IP è diviso in due parti:

* I primi *k* bit identificano la rete;
* I restanti 32 – *k* bit identificano l’host all’interno della rete.

Il valore di *k* può essere imposto in maniera arbitraria, a seconda se si vuole una rete più popolosa in termini di host o meno (aumentando o meno il numero di bit usati per identificare un host all’interno della rete).

**MASCHERA DI RETE (NETMASK)**

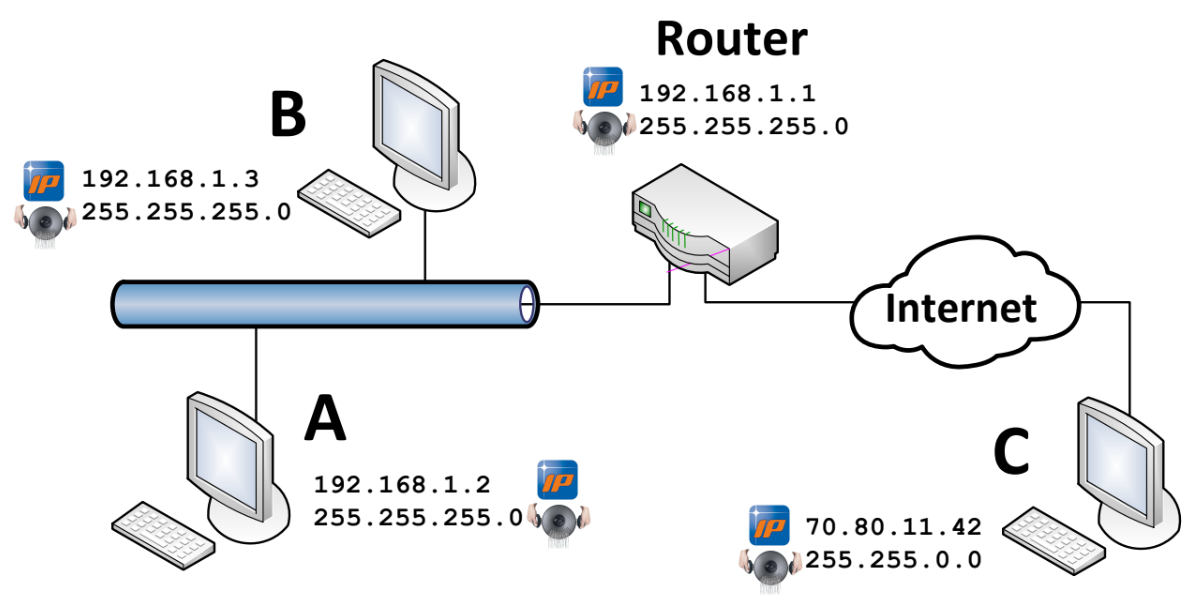
La maschera di rete è un numero a 32 bit fatto dai primi *k* bit più significativi a 1, dove *k* è il numero di bit che identificano la rete considerata. Si può indicare in maniera compatta con **/*k*.**



La maschera di rete è dunque un numero che caratterizza un’interfaccia di rete, ma più in particolare la rete a cui tale interfaccia è connessa, ed è utile in quanto, dato ad esempio un certo host A:

* Se vuole ricavare l’indirizzo di rete relativo ad una certa rete, e cioè un numero a 32 bit che presenta come *k* bit più significativi i bit che identificano la rete, gli basta fare un AND bit a bit tra l’indirizzo IP e la maschera di rete, entrambi dell’interfaccia usata per connettersi a tale rete;
* Se vuole ricavare l’indirizzo di broadcast relativo ad una certa rete, e cioè l’indirizzo IP che consente di inviare messaggi a tutti gli host della rete, gli basta fare un OR bit a bit tra l’indirizzo IP e la maschera di rete negata, entrambi dell’interfaccia usata per connettersi a tale rete.

Lo scenario visto prima diventa dunque il seguente:



A, B e il router (o meglio, le loro interfacce di rete) hanno un certo indirizzo IP e una certa maschera di rete.

C (o meglio, la sua interfaccia di rete) ha un certo indirizzo IP e una certa maschera di rete.

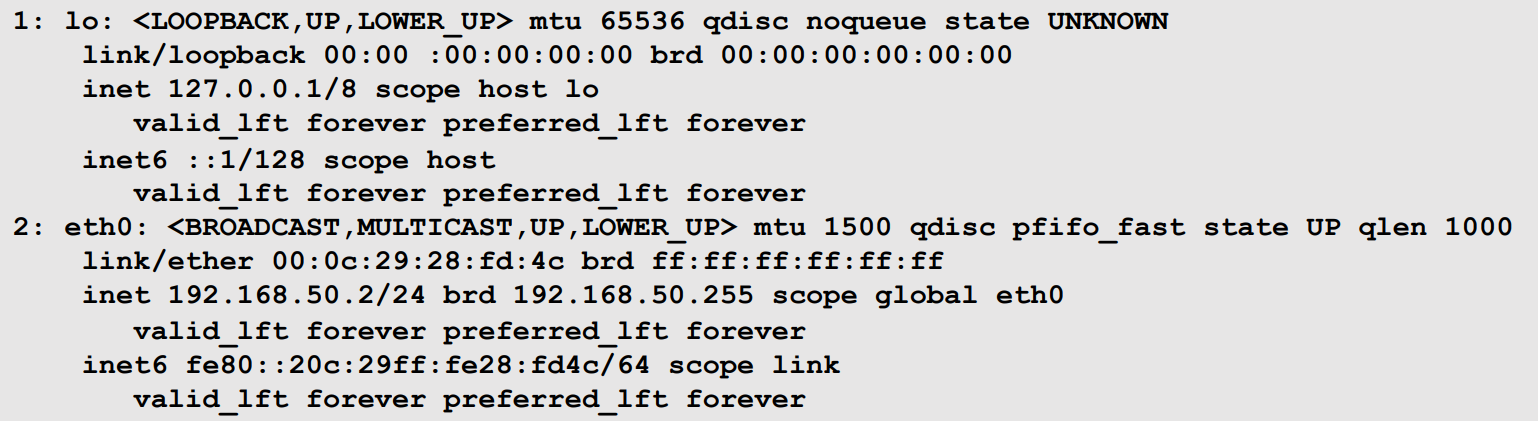
**COMANDO IP**

Il comando **ip** permette di visualizzare e manipolare le impostazioni di rete. Questo comando presenta a sua volta vari sottocomandi (detti anche oggetti). Vediamone alcuni.

**OTTENERE INFORMAZIONI SULLE INTERFACCE**

* **ip addr show:** mostra tutte le interfacce di rete che si trovano sul proprio host;
* **ip addr show up:** mostra tutte le interfacce di rete abilitate che si trovano sul proprio host;

*Esempio di output di ip addr show:*

**

Analizziamo alcune delle informazioni scritte:

* *1, 2*: è il numero dell’interfaccia di rete nell’elenco;
* *lo, eth0*: è il nome dell’interfaccia di rete;
* *LOOPBACK:* indica che quell’interfaccia di rete permette di mandare messaggi a se stessi;
* *BROADCAST:* indica che attraverso quell’interfaccia è possibile mandare un messaggio a tutti gli host della rete cui tale interfaccia è connessa;
* *MULTICAST:* indica che attraverso quell’interfaccia è possibile mandare un messaggio a più host della rete cui tale interfaccia è connessa (quindi a tutti, o solo ad un certo numero)
* *UP*: indica che l’interfaccia di rete è abilitata;
* *LOWER\_UP*: indica che l’interfaccia di rete è connessa;
* *inet IP*: è l’indirizzo IP dell’interfaccia di rete;
* */k*: è la maschera di rete dell’interfaccia di rete;
* *brd:* è l’indirizzo di broadcast dell’interfaccia di rete.

**ABILITARE E DISABILITARE LE INTERFACCE (E’ NECESSARIO ESSERE L’UTENTE ROOT)**

* **ip link set *interface* up**: abilita l’interfaccia di rete che ha il nome *interface*;
* **ip link set *interface* down**: disabilita l’interfaccia di rete che ha il nome *interface*.

**CONFIGURARE UN’INTERFACCIA (E’ NECESSARIO ESSERE L’UTENTE ROOT)**

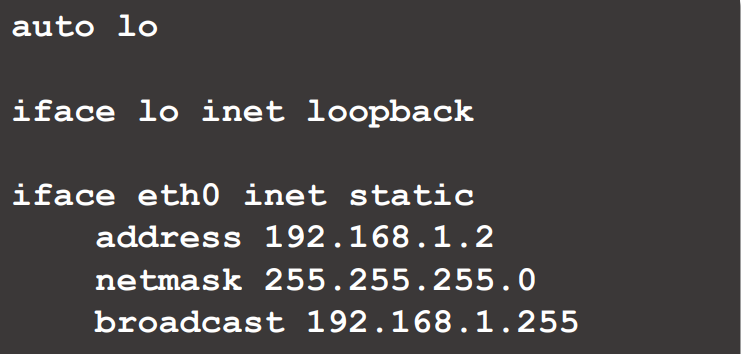
* **ip addr add *IP/k* dev *interface***: aggiunge l’indirizzo IP indicato (e va indicata anche la maschera di rete in forma compatta) all’interfaccia di rete di nome *interface*;
  + *es.* ip addr add 192.168.1.42/24 dev eth0
* **ip addr del *IP/k* dev *interface:*** rimuove l’indirizzo IP indicato (e va indicata anche la maschera di rete in forma compatta)dall’interfaccia di rete di nome *interface;*
  + *es.* ip addr del 192.168.1.42/24 dev eth0
* **ip addr flush dev *interface:*** rimuove tutti gli indirizzi IP dall’interfaccia di rete di nome *interface*

Tutte le configurazioni fatte con *ip* vanno perse al riavvio del calcolatore. Per avere una configurazione manuale permanente, bisogna modificare il file di configurazione */etc/network/interfaces*. Questo file è fatto per esempio così:

**Auto lo:** questa riga fa sì che, non appena si accende il calcolatore, venga abilitata l’interfaccia *lo*;

**iface lo inet loopback**: questa riga fa sì che l’interfaccia *lo* sia configurata come interfaccia di loopback.

**iface eth0 inet static**: questa riga fa sì che l’interfaccia eth0 abbia una configurazione statica, e cioè che, quando tale interfaccia viene abilitata, venga configurata con i parametri indicati nel file di configurazione (e quindi che abbia come indirizzo IP, maschera di rete e indirizzo di broadcast quelli indicati).



Vediamo ora altri comandi (è necessario essere l’utente root per usarli):

* **ifup *interface:*** abilita l’interfaccia di nome *interface*con la configurazione indicata in */etc/network/interfaces*;
* **ifdown *interface:*** disabilita l’interfaccia di nome *interface;*
* **ifup -a:** abilita tutte le interfacce che si trovano nella sezione *auto* di */etc/network/interfaces*, nello stesso ordine in cui sono scritte, con la configurazione indicata nel file. Viene eseguito all’avvio del calcolatore.

**DEFAULT GATEWAY**

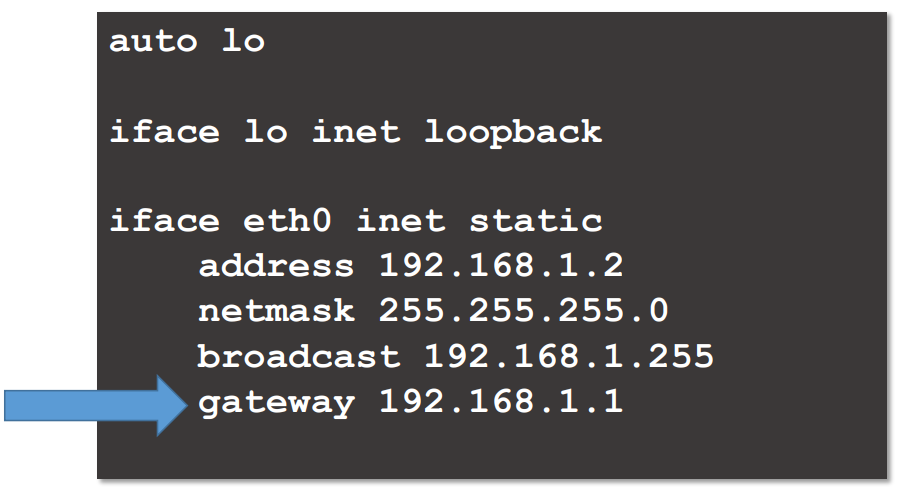
Quando un host sorgente vuole inviare un pacchetto ad un altro host destinatario, fa questo ragionamento:

* Se fa parte della stessa rete dell’host destinatario, allora manda il pacchetto direttamente a lui;
* Altrimenti, manda il pacchetto al default gateway, e cioè al router di frontiera della propria sottorete, e il router poi gestirà come mandare il pacchetto all’host destinatario.

Per vedere se l’host destinatario fa parte della sua stessa rete, l’host sorgente fa l’AND bit a bit tra l’indirizzo IP di destinazione e la maschera di rete di ciascuna delle proprie interfacce di rete:

* Se l’AND bit a bit dà l’indirizzo di rete della rete cui è connessa una delle proprie interfacce di rete, allora significa che l’host destinatario fa parte della sua stessa rete;
* Altrimenti, non fa parte.

L’host deve quindi sapere l’indirizzo del default gateway. Per far sì che lo sappia, bisogna aggiungere il campo “gateway *IP\_gateway”* nel file di configurazione */etc/network/interfaces*, all’interfaccia che si vuole venga usata per mandare i pacchetti fuori dalla propria sottorete (nel nostro caso, solo eth0):



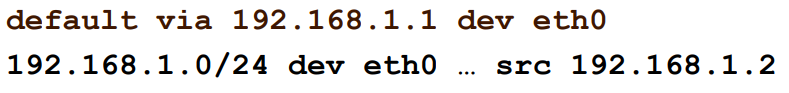
Vediamo ora altri comandi:

* **ip route show:** visualizza la tabella di routing, e cioè quali sono le rotte che l’host segue per inviare pacchetti in base all’indirizzo IP di destinazione.

La prima riga dice che, in generale, i pacchetti vengono mandati all’indirizzo IP 192.168.1.1 (che è l’IP del default gateway) tramite l’interfaccia eth0;

La seconda riga dice che, se un pacchetto deve essere mandato ad un host che ha indirizzo IP 192.168.1.0 e netmask /24, viene mandato direttamente a tale host tramite l’interfaccia eth0.

*Esempio di output:*

**

* **ip route add *IP/k* dev *interface***: permette di aggiungere una rotta per l’invio diretto nella sottorete locale, dicendo che se l’host deve mandare un pacchetto ad un host che ha l’indirizzo IP indicato e la netmask indicata, questo va mandato direttamente a tale host tramite l’interfaccia di nome *interface.* Questa aggiunta viene persa al riavvio del calcolatore (essendo una configurazione fatta con IP).   
  E’ necessario essere l’utente root per usare questo comando.

*es.* ip route add 192.168.1.0/24 dev eth0

* **ip route add default via *IP\_default\_gateway*:** permette di aggiungere l’indirizzo del default gateway. Questa aggiunta viene persa al riavvio del calcolatore (essendo una configurazione fatta con *ip*).   
  E’ necessario essere l’utente root per usare questo comando.

*es.* ip route add default via 192.168.1.1

* **ip route get *IP:*** permette di vedere qual è la rotta usata quando si deve mandare un pacchetto all’indirizzo IP indicato (quale tra le rotte che vengono mostrate facendo *ip route show*)

**DNS**

Un host, se ha un indirizzo IP, una maschera di rete e sa qual è l’indirizzo del router, è in grado di comunicare con altri host su Internet. Tuttavia, per un utente al momento è scomodo usare tale host: dovrebbe indicare ogni volta l’indirizzo IP dell’host con cui vuole comunicare, mentre lui era abituato per esempio a scrivere sul web browser l’hostname di tale host (es. [unipi.it](http://www.xnxx.com)) e accedere subito al suo contenuto. L’host dunque deve essere in grado di tradurre gli hostname in indirizzo IP, e questa traduzione può farla in due modi:

* **In modo statico**: l’host mantiene all’interno del file */etc/hosts* una serie di associazioni   
  *indirizzo IP – hostname,* equando deve tradurre un certo hostname in indirizzo IP, vede prima se la traduzione è contenuta all’interno di quel file;

*esempio di file /etc/hosts:*

127.0.0.1 localhost

127.0.1.1 studenti

131.114.73.85 unipi.it

* **In modo dinamico**: se la traduzione non è contenuta all’interno di quel file, l’host contatta un DNS server locale chiedendogli di tradurre quell’hostname in indirizzo IP; il DNS server locale, dopo un po’, gli dirà qual è l’indirizzo IP corrispondente. Gli indirizzi IP del DNS server locale primario e secondario che l’host può contattare sono contenuti nel file */etc/resolv.conf*.

*esempio di file /etc/resolv.conf:*

nameserver 8.8.8.4

nameserver 8.8.4.4

Dunque di default un host quando deve risolvere un hostname prima cerca nel file */etc/hosts;*  poi se non trova la traduzione in tale file chiede a un local DNS server. Questo comportamento può essere cambiato modificando il file */etc/nsswitch.conf*:

*esempio di file /etc/nsswitch.conf:*

*…*

Questo significa che, quando bisogna risolvere un hostname, prima l’host cerca nel file */etc/hosts,* poi se non trova la traduzione chiede a un local DNS server*.* Scambiando di ordine “files” e “dns” il comportamento sarà l’opposto.

hosts: files dns

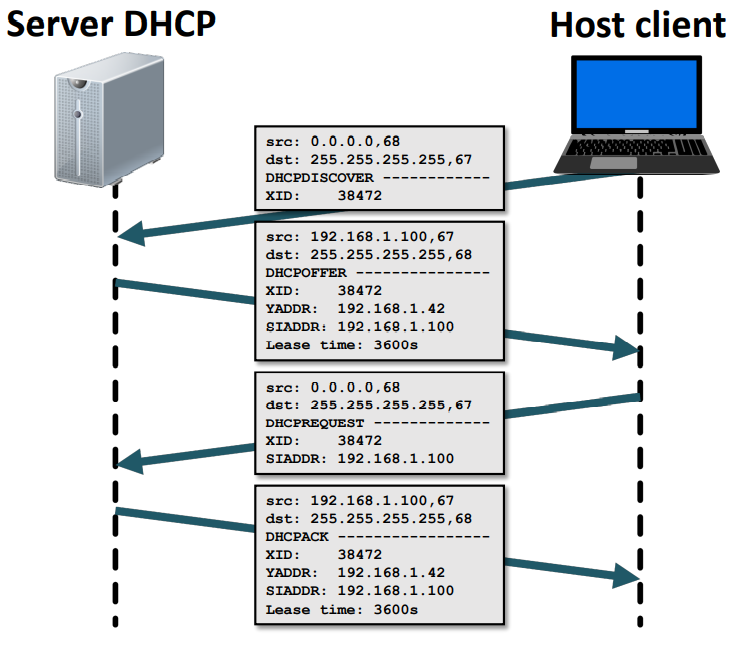
…

Se tutto è stato configurato correttamente, attraverso il comando **nslookup *hostname*** verrà mostrato l’indirizzo IP corrispondente all’hostname indicato (purché tale hostname esista).

**DHCP**  *14-10-22*

Configurare manualmente l’indirizzo IP, la netmask, l’indirizzo del gateway e l’indirizzo del server DNS per ogni interfaccia di rete è palloso. E’ possibile fare questa configurazione in maniera automatica, attraverso i **server DHCP**. Questi sono degli host che vengono inseriti all’interno della propria sottorete (in particolare, ogni router contiene al suo interno un server DHCP), e quando un nuovo host vuole connettersi a tale rete, il server DHCP gli assegna i quattro parametri necessari per navigare su Internet.

Vediamo cosa succede in dettaglio:



Supponiamo che un certo host client voglia connettersi ad una sottorete (attraverso il cavo Ethernet o attraverso il WiFi, quindi attraverso una certa interfaccia di rete). Succede questo:

1. Innanzitutto, l’host client vede prima se c’è un server DHCP nella rete cui si vuole connettere. A tale scopo, un processo che gira sull’host client, detto client DHCP, manda un messaggio DHCPDISCOVER.

L’indirizzo IP destinatario è 255.255.255.255. Questo è un altro indirizzo di broadcast, dunque questo messaggio viene mandato a tutti (visto che non sa se ci sono server DHCP, e quindi neanche il loro eventuale indirizzo IP). Quel “67” è la porta del processo che gira sul server DHCP, detto a sua volta server DHCP. L’indirizzo sorgente utilizzato è 0.0.0.0 (che chiaramente è un indirizzo IP fittizio, visto che l’host client al momento non ha ancora alcun indirizzo IP). Quel “68” è la porta del client DHCP.

In questo messaggio è presente anche il campo XID, che è un ID scelto dal client DHCP per identificare la comunicazione (poiché, come si vedrà, sia il client DHCP che il server DHCP manderanno i successivi messaggi a tutti, e poiché potrebbero esserci più host che contemporaneamente vogliono connettersi alla stessa rete e che hanno tutti lo stesso IP fittizio 0.0.0.0, il client DHCP e il server DHCP inseriscono all’interno dei loro messaggi questo XID, così che il server DHCP sappia esattamente da quale client DHCP gli è arrivato il messaggio, e il client DHCP sappia se il server DHCP ha risposto proprio a lui o ad un altro client).

1. Il server DHCP risponde con un messaggio DHCPOFFER in cui offre all’host client un indirizzo IP che ha un certo tempo di scadenza (lease time).

L’indirizzo sorgente utilizzato è l’indirizzo IP vero del server DHCP, e il messaggio viene mandato a tutti, sia perché l’host client ancora non ha un indirizzo IP, ma anche perché nella rete ci potrebbero essere più server DHCP, e quindi in questo modo gli altri server DHCP sono al corrente di ciò che sta succedendo (altrimenti avrei che ogni server DHCP procederebbe ad assegnare un indirizzo IP all’host, mentre basta che se ne occupi un solo server DHCP). Anche qui è presente il campo XID per identificare la comunicazione.

1. Il client DHCP risponde con un messaggio DHCPREQUEST, accettando l’offerta (è una “richiesta di accettazione”).

L’indirizzo sorgente è ancora 0.0.0.0, visto che l’host client ancora non ha un indirizzo IP (il server DHCP deve ancora rispondere alla richiesta di accettazione), e il messaggio viene mandato a tutti, così che anche gli altri server DHCP siano al corrente di ciò che sta succedendo. Anche qui è presente il campo XID per identificare la comunicazione.

1. Il server DHCP risponde con un messaggio DHCPACK, confermando la ricezione dell’accettazione.

Il messaggio viene mandato a tutti, così che anche gli altri server DHCP siano al corrente di ciò che sta succedendo. Anche qui è presente il campo XID per identificare la comunicazione.

A questo punto, il client DHCP manderà un altro messaggio al server DHCP per ottenere gli altri parametri di rete (netmask, indirizzo del gateway ed indirizzo del DNS server), e una volta ottenuti, l’host client sarà in grado di connettersi ad Internet.

Client DHCP e server DHCP costituiscono dunque un’applicazione di rete basata sul modello client-server, e DHCP è il protocollo di livello application utilizzato. DHCP usa UDP come protocollo di trasporto.

**NB:** se in una sottorete è presente un server DHCP, non significa che non è più possibile assegnare staticamente un indirizzo IP ad un host connesso a tale rete! Infatti, un server DHCP assegna ad ogni host un indirizzo IP, e questo indirizzo IP viene preso da un certo insieme di indirizzi IP; tutti gli IP che non fanno parte di questo insieme possono essere assegnati staticamente agli host.

**TEST DI CONNETTIVITA’**

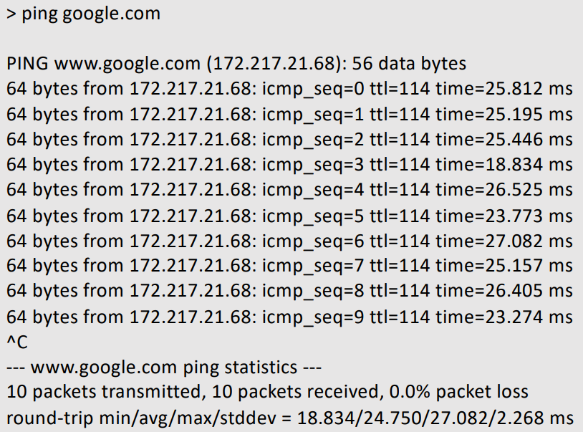
Esistono due test per vedere se il proprio host è connesso ad Internet.

**PRIMO TEST: COMANDO PING**

Il comando ***“ping hostname”*** o ***“ping IP”*** funziona in questo modo:

* Non appena digitato, inizia a mandare una serie di pacchetti ICMP ECHO\_REQUEST all’host che ha l’hostname/IP indicato;
* Per ogni pacchetto ICMP\_ECHO\_REQUEST, l’host destinatario risponde mandando un pacchetto ICMP ECHO\_RESPONSE;
* Il comando *ping* a questo punto calcola varie statistiche, come il RTT (quanto tempo passa da quando manda un pacchetto a quando riceve il corrispondente pacchetto di risposta) e se ci sono stati pacchetti persi (cioè se non ha ricevuto alcuna risposta entro un certo timeout a qualche richiesta che ha fatto);
* Premendo CTRL + C, viene stoppato il comando e vengono mostrate le statistiche calcolate.

*Esempio di output di “ping google.com”:*

**

Se il comando *ping* viene eseguito mettendo come *hostname/IP* quello di un host che si trova al di fuori della propria sottorete e non si ha alcun errore, allora significa che il proprio host è correttamente connesso ad Internet. Altrimenti, c’è qualche problema, e bisogna vedere che tipo di messaggio di errore compare:

* ***Network Unreachable:*** il proprio host non ha route valide per raggiungere l’host destinatario (e cioè, vedendo l’indirizzo IP dell’host destinatario, questo pacchetto non può essere mandato né direttamente ad un host della propria sottorete, né al gateway). Questo potrebbe essere dovuto ad una cattiva configurazione dei parametri di rete (*es*. l’host non sa l’indirizzo del gateway), sia essa statica (attraverso il file */etc/network/interfaces*) o automatica (attraverso il server DHCP).
* ***100% packet loss*:** il proprio host non ha ricevuto alcuna risposta alle richieste che ha fatto. Questo potrebbe essere dovuto:
  + A un problema a livello dell’ISP, che non fornisce connettività ad Internet;
  + A qualche firewall (dell’host sorgente/destinatario o di qualche router nel mezzo), che ha bloccato le richieste in uscita o le risposte in entrata;
  + Se viene usato *ping hostname*, potrebbe essere dovuto a una sbagliata associazione *hostname – IP* nel file */etc/hosts*, oppure a un problema legato al server DNS, che ha tradotto male l’hostname indicato. In ogni caso, si cercherebbe di “pingare” un indirizzo IP a cui non corrisponde alcun host, e quindi nessun host risponderà.
* ***Unknown host*:** se viene usato *ping hostname*, indica che non è stato possibile risolvere l’hostname specificato. Potrebbe essere dovuto ad un malfunzionamento dei server DNS, oppure a una cattiva configurazione del parametro di rete relativo al server DNS, sia essa statica (*es.* non si è indicato alcun indirizzo IP di un DNS server locale in */etc/resolv.conf*) o dinamica (il server DHCP non ha assegnato un indirizzo di un DNS server locale all’host).

**SECONDO TEST: COMANDO TRACEROUTE**

Il comando ***“*traceroute *hostname”*** o ***“*traceroute *IP”*** mostra “il percorso di router” che un pacchetto effettua partendo dal proprio host per raggiungere l’host destinatario che ha l’*hostname/IP* indicato. Funziona in questo modo:

* Innanzitutto, i datagram IP hanno un campo **TTL** (Time To Live), che è il numero di router che tale datagram può attraversare prima di essere buttato. Ogni volta che un datagram arriva ad un nuovo router, questo decrementa il campo TTL e, se arriva a 0, butta il datagram, mandando all’host sorgente di tale pacchetto un messaggio di errore ***ICMP Time Exceeded*,** contenente il proprio indirizzo IP (e quindi contenente l’indirizzo IP del router che ha portato TTL a 0);
* Il comando *traceroute* sfrutta questo campo, iniziando a mandare verso l’host destinatario una serie di terne di pacchetti UDP con TTL sempre crescente, partendo da TTL = 1 (quindi manda la prima terna con TTL = 1, la seconda terna con TTL = 2 …).

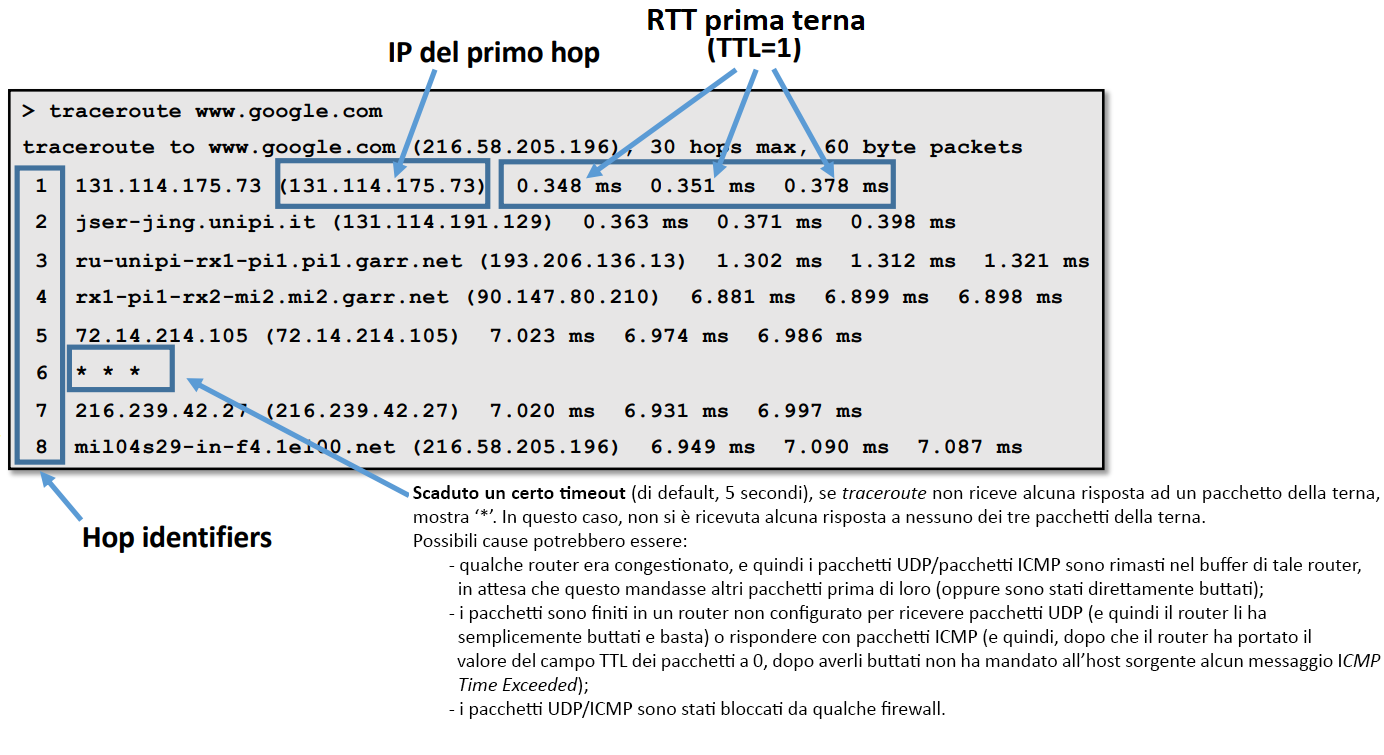
Man mano che arrivano i messaggi *ICMP Time Exceeded* di risposta, *traceroute* si salva gli indirizzi IP contenuti al loro interno e calcola il RTT (e cioè quanto tempo è passato da quando ha inviato una terna a quando ha ricevuto i messaggi di risposta per ogni pacchetto della terna).

Quando una terna arriverà all’host destinatario, questo non risponderà con *ICMP Time Exceeded*, ma con un messaggio di errore ***ICMP Destination Unreachable***. Infatti, questi pacchetti verranno mandati “ad una certa porta”. *traceroute* sceglie come numero di porta un numero molto grande, così da rendere improbabile che sull’host destinatario giri un processo che abbia tale numero di porta, e così l’host destinatario risponderà con il messaggio di errore scritto sopra;

* A questo punto, *traceroute* ricostruisce il “percorso di router” effettuato dalle terne, ordinando gli IP salvati in base al TTL crescente, e mostra questo percorso.

Prima di vedere l’output, analizziamo alcuni problemi di questo comando:

* I pacchetti di una stessa terna possono seguire percorsi diversi, arrivando persino ad un router finale diverso e quindi portare a risposte *ICMP Time Exceeded* da router diversi. Questo può succedere perché ogni pacchetto viene inoltrato dai router lungo il percorso che attualmente è il migliore da seguire (si vedrà meglio in futuro). Per evitare ciò, *traceroute* invia i 3 pacchetti quasi simultaneamente, così che il percorso migliore da seguire sia lo stesso per tutti e 3 i pacchetti;
* Terne diverse possono seguire percorsi diversi (*es.* una terna può seguire un percorso, la terna successiva uno totalmente diverso), quindi il percorso mostrato in output potrebbe essere in realtà l’insieme di più percorsi diversi;
* I messaggi *ICMP Time Exceeded* possono seguire un percorso totalmente diverso da quello effettuato dai pacchetti delle terne di cui tali messaggi sono la risposta, per cui il calcolo del RTT è inaffidabile (visto che all’andata e al ritorno potrebbe esser fatto un percorso diverso).

E*sempio di output di traceroute google.com:*

**PROGRAMMARE APPLICAZIONI DI RETE** *28-10-22*

**INTRODUZIONE AL LINGUAGGIO C**

Iniziamo ora a vedere come programmare un’applicazione di rete, e cioè un’applicazione costituita da due o più processi che girano contemporaneamente su dispositivi diversi e che possono interagire tra loro.

Nel farlo, useremo il linguaggio C, che è quello più utilizzato per questo tipo di applicazioni. E’ un po’ diverso rispetto al C++, dunque prima vediamo quali sono le principali differenze tra i due linguaggi.

**DICHIARAZIONE/DEFINIZIONE DI VARIABILI**

Le variabili possono essere dichiarate/definite solo all’inizio di un blocco.

*es.*

int main(){

int a = 5, i;

int b = f(a);

a = func(1000);

for (i = 0; a < 100; i++){

int c = 0;

b = f(a);

}

}

**STRUTTURE**

Si deve inserire sempre “*struct*” all’inizio della dichiarazione/definizione di una variabile di tipo struttura.

*es.*

struct Complex {

double Re;

double Im;

};

int main() {

int a = 4;

struct Complex c;

}

**MEMORIA DINAMICA (HEAP)**

Lo heap viene gestito attraverso due funzioni della libreria *stdlib*:

* **malloc(*n*):** alloca un’area di memoria di *n* byte sullo heap. Restituisce un *void\** contenente l’indirizzo del primo byte dell’area allocata;
* **free(*punt*):** rilascia l’area di memoria sullo heap puntata da *punt*.

*es1:*

#include <stdlib.h>

int main() {

int n = 5;

void\* punt;

punt = malloc(n);

…

free(punt);

}

*es2:*

#include <stdlib.h>

int main() {

Se si vuole assegnare l’indirizzo restituito da malloc ad una variabile di tipo diverso da *void\** (*es.* a un puntatore a double) è necessario fare un cast.

int n = 10;

double\* punt;

punt = (double\*)malloc(sizeof(double)\*dimensione)

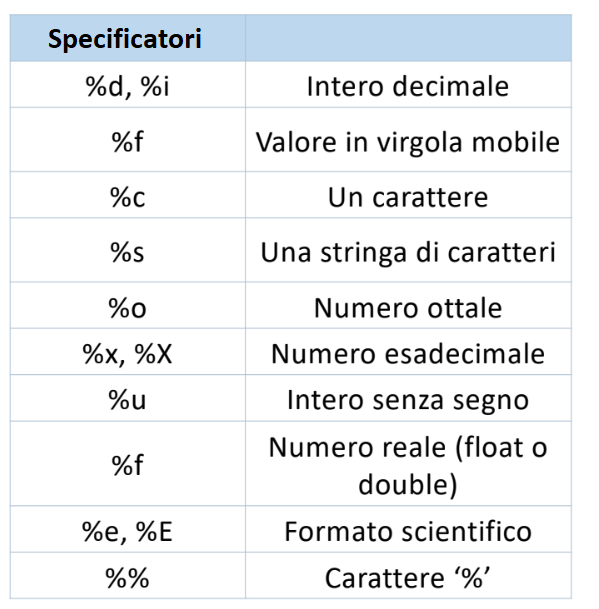
…

free(punt);

}

**INGRESSO/USCITA**

L’ingresso/uscita è gestito attraverso due funzioni della libreria *stdio:*

* **printf(const char\* *formato*, *lista\_variabili*)**: stampa sullo *stdout* (a video) la stringa *formato.* Per stampare delle variabili in punti specifici della stringa, in ognuno di questi punti va messo uno **specificatore di formato**, che serve ad indicare, per ogni variabile, in che modo deve essere stampata (*es*. come un intero, un numero in virgola mobile, un carattere…). Le variabili da stampare vanno poi passate in *lista\_variabili*,e verranno stampate nel punto e nel modo specificato dal corrispondente specificatore.

La funzione restituisce il numero di caratteri stampati.

* **scanf(const char\* *formato*, *lista\_indirizzi):***legge dallo *stdin* (dalla tastiera) sequenze di caratteri (cifre o lettere). Ogni sequenza viene memorizzata al corrispondente indirizzo presente nella *lista\_indirizzi*, nel modo indicato dal corrispondente specificatore presente nella stringa *formato (es.* verrà memorizzata come un intero, un numero in virgola mobile, un carattere… e quindi occuperà un certo numero di byte partendo dall’indirizzo specificato).

La funzione restituisce il numero di sequenze di caratteri correttamente memorizzate.

*es:*

#include <stdio.h>

int main() {

inti, j;

char\* str = "Com’è bello il C!\n";

Se non si vuole stampare alcuna variabile all’interno della   
stringa *formato*, non si passa alcuna *lista\_variabili*.

printf(str);

printf("Da oggi pane e C!");

printf("Messaggio:\n %s", str);

Altrimenti, si mette l’apposito specificatore di formato nel punto della stringa *formato* in cui si vuole venga stampata una variabile e si passa poi tale variabile in *lista\_variabili.*

i = 5;

printf("i = %d\n", i);

La scanf leggerà da tastiera una sequenza di caratteri e la memorizzerà come un intero nella variabile *j* (si è passato infatti l’indirizzo della variabile *j*).

scanf("%d", &j);

}

**GESTIONE STRINGHE**

Le stringhe si possono gestire attraverso la libreria *string*:

* **strlen(*str*):** restituisce la lunghezza della stringa *str*, e cioè il numero di caratteri di cui è composta, partendo dall’inizio fino alla prima marca di fine stringa esclusa;
* **strcmp(*str1, str2*):** confronta le due stringhe *str1* e *str2* e restituisce un numero:
  + = 0 se le due stringhe sono identiche;
  + < 0 se il primo carattere di *str1* diverso dal corrispondente di *str2* ha codifica ASCII minore;
  + > 0 se il primo carattere di *str2* diverso dal corrispondente di *str2* ha codifica ASCII maggiore;
* **strncpy(dest, src, n):** copia nell’array di caratteri puntato da *dest*, partendo dall’inizio,i primi *n* caratteri della stringa *src.* Non mette in automatico la marca di fine stringa, che dunque verrà messa nell’array di caratteri di destinazione solo se rientra tra gli *n* caratteri copiati. Se *n* è maggiore della lunghezza di *src*, dopo aver copiato la marca di fine stringa, l’array di caratteri di destinazione viene riempito di altre marche di fine stringa fino a quando non si saranno scritti in totale *n* caratteri.
* **strncat(dest, src, n):** concatena alla stringa *dest*, partendo dalla prima marca di fine stringa, i primi *n* caratteri della stringa *src*. Mette in automatico la marca di fine stringa, se non dovesse rientrare tra gli *n* caratteri concatenati. Se *n* è maggiore della lunghezza di *src*, dopo aver concatenato la marca di fine stringa, non viene concatenato altro.

Tenendo conto del comportamento delle ultime due funzioni, bisogna quindi stare attenti che in *dest* ci sia sufficiente spazio, altrimenti si avrebbe segmentation fault.

*es1:*

#include <string.h>

int main(){

char\* str1 = "Gatto \n";

char\* str2 = "Gattone \n";

int len;

int ret;

strlen(*str1*) restituisce 7, visto che i caratteri di cui è composta *str1* fino alla prima marca di fine stringa esclusa sono ‘g’, ‘a’, ‘t’, ‘t’, ‘o’, ‘ ‘, ‘\n’ (ricorda che le sequenze di escape rappresentano un unico carattere, e cioè un unico byte codificato in ASCII).

len = strlen(str1);

strcmp(*str1, str2*) restituirà un numero < 0, visto che, facendo il confronto tra ‘ ‘ e ’n’, ‘ ‘ ha una codifica ASCII minore.

ret = strcmp(str1, str2);

}

*es2:*

#include <string.h>

int main(){

char str1[20];

char\* str2 = "nero\n";

int n;

strncpy(*str1, “*Gatto \n”, n) copierà in *str1*, partendo dall’inizio, i primi *n* = 20 caratteri della stringa "Gatto \n". Visto che “Gatto \n” ha lunghezza 7, dopo aver copiato la marca di fine stringa, verranno scritti in *str1* altre marche di fine stringa fino a quando non si saranno scritti in totale *n* = 20 caratteri.

n = sizeof(str1);

strncpy(str1, "Gatto \n", n);

*strncat*(*str1, str2*, n) concatenerà a *str1*, partendo dalla prima marca di fine stringa, i primi *6* caratteri di *str2*.

strncat(str1, str2, 6);

}

**LETTURA E SCRITTURA SU FILE**

La lettura e scrittura su file è gestita attraverso la libreria *stdio.*

* **FILE\* *fd***. Innanzitutto, per leggere/scrivere su un file, bisogna prima creare una variabile di tipo FILE\*. Questa variabile è uno stream, già agganciato al nostro programma, e che agganceremo poi al file cui vogliamo accedere.
* ***fd*** = **fopen(“*path\_file*”, “*modalità\_di\_apertura*”):** in questo modo “apriamo” il file che si trova in *path\_file*, agganciandoci lo stream *fd*. La *modalità\_di\_apertura* specifica in che modo apriamo il file (e quindi che tipo di accesso ci consentirà di fare lo stream), e può essere di vari tipi:
  + **r** = apertura in sola lettura;
  + **w** = apertura in sola scrittura;
  + **r+** = apertura in lettura e scrittura;
  + **a** = apertura in solo append;
  + **a+** = apertura in lettura e append.

La lettura e la scrittura viene fatta partendo dall’inizio del file; la scrittura in append viene fatta partendo dalla file del file.

Vediamo ora dei dettagli:

* + Se la modalità di apertura è *sola scrittura* o *solo append* e il file non esiste, viene creato.
  + Se la modalità di apertura è *sola scrittura* e il file esiste, tutto il suo contenuto viene cancellato.
  + Se a seguito dell’apertura *fd == NULL*, allora si è verificato un errore:
    - in caso di *lettura*, può succedere se il file non esiste o se non si ha il permesso di accesso di accesso in lettura a tale file;
    - in caso di *scrittura*, può succedere se non si ha il permesso di accesso in scrittura a tale file.
* **fprintf(FILE\* *fd*, const char\* *formato*, *lista\_argomenti*):** come la printf, con la differenza che stampa sul file cui è agganciato lo stream *fd.*
* **fscanf(FILE\* *fd*, const char\* *formato*, *lista\_indirizzi*):** come la scanf, con la differenza che legge dal file cui è agganciato lo stream *fd.*
* **fclose(FILE\* *fd*):** in questo modo “chiudiamo” il file cui è agganciato lo stream *fd*, sganciandoci tale stream.

*es:*

#include <stdio.h>

int ret, n;

char\* str = “Gatto!\n”;

FILE\* fd;

Apriamo il file in sola lettura, agganciandoci lo stream *fd*.

Dopo aver controllato che non ci sono stati errori, *fscanf* leggerà dal file, partendo dall’inizio, una sequenza di caratteri e la memorizzerà come intero nella variabile n. In *ret* finirà il numero di sequenze di caratteri correttamente memorizzate (se tutto è andato bene, avrà come valore 1).

Dopodiché, chiudiamo il file, sganciandoci lo stream fd.

fd = fopen("/tmp/gatto.txt", "r");

if (fd == NULL) {

// Gestione errore

}

ret = fscanf(fd, "%d", &n);

fclose(fd);

Riapriamo il file, questa volta in sola scrittura, agganciandoci lo stream *fd*.

*fprintf* stamperà sul file, partendo dall’inizio, il contenuto della variabile *str*. In *ret* finirà il numero di caratteri stampati (se tutto è andato bene, avrà come valore 1).

Dopodiché, chiudiamo il file, sganciandoci lo stream fd.

fd = fopen("/tmp/gatto.txt", “w”);

ret = fprintf(fd2, "%s", str);

fclose(fd);

**ELEMENTI UTILI PER LE APPLICAZIONI DI RETE**

**TIPI TERMINANTI IN “\_t”**

Il C presenta alcuni tipi speciali terminanti con **‘\_t’** la cui dimensione è fissa, indipendentemente dall’architettura usata. Questi tipi sono fondamentali per le applicazioni di rete: infatti, dato un qualsiasi tipo, la sua dimensione dipende dall’architettura usata. Questo significa che, se ad esempio un processo A che gira su un host volesse mandare un *int* ad un processo B che gira su un altro host:

* A potrebbe mandare 8 byte, visto che quella è la dimensione di un *int* nell’architettura su cui gira;
* B invece potrebbe aspettarsi 4 byte, visto che quella è la dimensione di un *int* nell’architettura su cui gira.

B allora potrebbe memorizzare male il dato ricevuto. Se invece A mandasse a B un *int32\_t*, questo occupa 32 bit su tutte le architetture, e quindi non si avrebbero problemi di questo genere.

**ENDIANNESS**

Un altro elemento che dipende dall’architettura usata è il modo in cui un dato viene salvato. In particolare, se si ha un dato che occupa 2 o più byte, questo si può memorizzare in due modi:

* **Formato *little*-endian:** viene memorizzato prima (== ad indirizzo minore) il byte più significativo;
* **Formato *big-endian*:** viene memorizzato prima (== ad indirizzo minore) il byte meno significativo.

*es.* dato il numero 422990, si può memorizzare su 4 byte in due modi:

* Immagine che contiene testo

  Descrizione generata automaticamenteFormato *little-endian:*
* Formato *big-endian:*

Immagine che contiene testo

Descrizione generata automaticamente

In rete viene usato il formato big-endian, che per questo motivo è anche detto **network order**.

Questo significa che:

* Se un processo A che gira su un host che memorizza i dati in formato little-endian vuole mandare attraverso la rete un dato da 2 o più byte ad un altro processo B che gira su un altro host, dovrà prima convertirlo in big-endian;
* Se B gira su un host che memorizza i dati in formato little-endian, prima di memorizzare tale dato ricevuto dalla rete dovrà convertirlo in little-endian.

Per fare questo, il C fornisce delle apposite **funzioni di conversione:**

#include <arpa/inet.h>

**La prima è “Host To Network Long”:** si passa come parametro un unsigned su 32 bit memorizzato in host order (e cioè nel formato usato dall’host) e lo restituisce convertito network order. Se l’host memorizza già i dati in network order (e quindi host order == network order), la funzione restituisce semplicemente l’unsigned passato come argomento.

**La seconda è “Host to Network Short”:** fa la stessa cosa ma per gli unsigned su 16 bit.

**La terza è “Network to Host Long”:** si passa come parametro un unsigned su 32 bit memorizzato in network order e lo restituisce convertito in host order. Se l’host memorizza già i dati in network order, la funzione restituisce semplicemente l’unsigned passato come argomento.

**La terza è “Network to Host Short”:** e fa la stessa cosa ma per gli unsigned su 16 bit.

uint32\_t htonl(uint32\_t hostlong);

uint16\_t htons(uint16\_t hostshort);

uint32\_t ntohl(uint32\_t netlong);

uint16\_t ntohs(uint16\_t netshort);

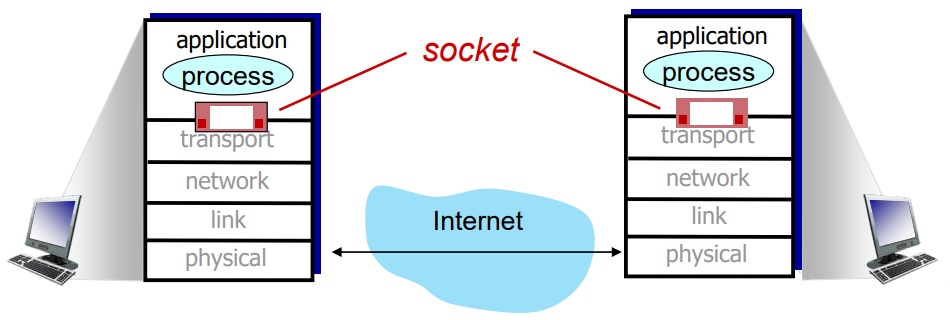
**NB:** l’endianness riguarda il modo in cui un dato viene memorizzato, non l’ordine dei dati! Per esempio, dato un array di interi, se questo viene memorizzato in formato big-endian, si avrà sempre che prima viene memorizzato il primo elemento, poi il secondo, poi il terzo… Non viene invertito l’ordine dell’array! Similmente, se un processo manda tale array ad un altro processo attraverso la rete, questo lo riceverà con gli elementi che sono ancora in ordine; semplicemente ogni intero sarà in formato big-endian.

**COMUNICAZIONE TRA PROCESSI: SOCKET**

Nelle applicazioni di rete, processi diversi che girano su host diversi possono comunicare tra loro scambiandosi messaggi. Per rendere possibile tale comunicazione, i sistemi operativi moderni mettono a disposizione i cosiddetti **socket**, e cioè delle interfacce, simili a delle cassette della posta, ognuna associata ad un processo.

In particolare, dato un processo A che gira su un certo host, se vuole inviare ad un messaggio ad un altro processo B che gira su un altro host, gli basta inviarlo al proprio socket. Da qui, il messaggio raggiungerà il socket del processo B, che per leggere il messaggio ricevuto gli basta leggere dal proprio socket.

Chiaramente ci sono poi tutti i livelli inferiori (transport, network, link e physical) che fanno sì che i messaggi inviati da un processo al proprio socket finiscano effettivamente al socket del processo destinatario, ma di questo i processi non se ne devono preoccupare.



Consideriamo un’applicazione client-server costituita da un processo client e un processo server, in esecuzione su host diversi. Il sistema operativo offre ad ognuno di questi processi delle **primitive** per:

* *Creare* un socket;
* *Assegnargli* un indirizzo;
* *Connetterlo* ad un altro socket;
* *Inviare* e *ricevere* dati attraverso il socket;
* …

A questo punto, vediamo quali primitive deve chiamare il server e quali deve chiamare il client, così che i due processi possano comunicare tra loro.

**SERVER**

**PRIMITIVA SOCKET()**

Innanzitutto, il server deve creare un socket. Questo lo fa attraverso la primitiva socket():

#include <sys/types.h>

#include <sys/socket.h>

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

* ***domain*:** specifica qual è la famiglia di protocolli che verrà poi usata nella comunicazione. Noi metteremo sempre la costante AF\_INET, in modo che vengano usati i protocolli IPv4 e TCP/UDP.
* ***type*:** specifica la tipologia di socket. Noi vedremo due tipi:
  + SOCK\_STREAM: socket che garantisce una comunicazione in cui verrà usato TCP come protocollo di trasporto;
  + SOCK\_DGRAM: socket che garantisce una comunicazione in cui verrà usato UDP come protocollo di trasporto;
* ***protocol*:** per noi sarà sempre 0 (non vediamo perché).

La primitiva restituisce un **descrittore di file**, e cioè un intero non negativo che identifica univocamente un file, una pipe o un socket (come in questo caso) aperto da un processo. In caso di errore, restituisce -1.

**STRUTTURE SOCKET**

Creato un socket, bisogna creare la struttura per contenere il suo indirizzo. Questo è costituito da 3 elementi:

* La famiglia di protocolli che verrà poi usata nella comunicazione;
* Numero di porta.
* Indirizzo IP dell’host su cui gira il server;

La struttura usata per memorizzare l’indirizzo di un socket è dunque la seguente:

#include <sys/socket.h>

#include <netinet/in.h>

struct sockaddr\_in {

sa\_family\_t sin\_family; // Conterrà la famiglia di protocolli che verrà poi usata nella comunicazione (per noi sempre AF\_INET).

in\_port\_t sin\_port; // Conterrà il numero di porta espresso in network order

struct in\_addr sin\_addr; // Istanza di struttura *in\_addr,* conterrà l’indirizzo IP espresso in network order

};

Dove *in\_addr* è una struttura fatta in questo modo:

struct in\_addr {

uint32\_t s\_addr; // Conterrà l’indirizzo IP espresso in network order.

};

L’indirizzo IP, oltre ad essere espresso in network order, è espresso in **formato numerico**. Esistono infatti principalmente due modi per rappresentare un indirizzo IP:

* Formato **presentation,** e cioè come stringa che contiene l’indirizzo IP in formato decimale puntato;
* Formato **numerico,** e cioè come naturale su 32 bit.

Per convertire un indirizzo IP da un formato ad un altro si usano delle apposite **funzioni di conversione**:

#include <arpa/inet.h>

**La prima è “Presentation To Numeric”** e converte in formato numerico l’indirizzo IP rappresentato in formato presentation contenuto nella stringa *src*.

Il risultato della conversione viene memorizzato in network order nella variabile di tipo *in\_addr* puntata da *dst*.

**La seconda è “Numeric to Presentation”** e converte in formato presentation l’indirizzo IP rappresentato in formato numerico (in host order) contenuto nella variabile di tipo *in\_addr* puntata da *src.*

Il risultato della conversione viene memorizzato nell’array di caratteri puntato da *dst*, di dimensione *size,* indicata in byte.

*socklen\_t* è un tipo riservato, ma si può passare tranquillamente un letterale/variabile *int*.

Come parametro *af*, per entrambe passeremo sempre la costante AF\_INET.

int inet\_pton(int af, const char\* src, void\* dst);

const char\* inet\_ntop(int af, const void\* src,

char\* dst, socklen\_t size);

**PRIMITIVA BIND()**

Una volta che il server ha creato un socket e una variabile di tipo *sockaddr\_in* in cui ha messo quello che vuole sia l’indirizzo del socket, deve effettivamente assegnare tale indirizzo a quel socket. Questo lo fa attraverso la primitiva bind():

int bind(int *sockfd*, const struct sockaddr\* *addr*, socklen\_t *addrlen*);

* ***sockfd*:** si passa il descrittore del socket cui si vuole assegnare un indirizzo;
* ***addr*:** si passa un puntatore alla struttura di tipo *sockaddr\_in* contenente l’indirizzo che si vuole assegnare al socket. Visto però che la funzione accetta solo *sockaddr\**, va fatto un cast.
* ***addrlen*:** si passa la dimensione in byte della struttura di tipo *sockaddr\_in* del punto precedente.

La primitiva restituisce 0 se tutto è andato bene, -1 altrimenti.

**PRIMITIVA LISTEN()**

Il server ora ha creato un socket e gli ha assegnato un indirizzo. Supponiamo sia un socket di tipo SOCK\_STREAM: questo garantisce una comunicazione basata su TCP, protocollo che prima richiede sia stabilita una connessione tra i processi che vogliono comunicare. Il server allora deve indicare che tale socket è un **socket di ascolto**, e cioè un socket usato per ricevere le richieste di connessione mandate dai client. Questo lo fa attraverso la primitiva listen():

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

* ***sockfd*:** si passa il descrittore del socket che si vuole rendere un socket di ascolto;
* ***backlog*:** una volta chiamata la primitiva, viene creata una coda in cui salvare le richieste di connessione che arrivano al socket. Come parametro *backlog* si passa proprio la dimensione di questa coda, e cioè il numero massimo di richieste di connessione che possono rimanere in attesa di essere accettate (oltre il quale, nuove richieste verrebbero scartate).

La primitiva restituisce 0 se tutto è andato bene, -1 altrimenti.

**PRIMITIVA ACCEPT()**

Per accettare una richiesta di connessione arrivata sul socket di ascolto, il server chiama la primitiva accept():

int accept(int *sockfd*, struct sockaddr\* *addr,* socklen\_t\* *addrlen*);

* ***sockfd*:** si passa il descrittore del socket di ascolto interessato;
* ***addr*:** si passa un puntatore a una struttura di tipo *sockaddr\_in* in cui verrà salvato l’indirizzo del socket del client la cui richiesta è stata appena accettata. Visto però che la funzione accetta solo *sockaddr\**, va fatto un cast.
* ***addrlen:*** si passa un puntatore a una variabile contenente la dimensione in byte della struttura di tipo *sockaddr\_in* del punto precedente.

E’ una primitiva bloccante: una volta invocata, il server si blocca fino a quando non avrà accettato una richiesta di connessione (dunque se nella coda in cui sono salvate le richieste di connessione arrivate al socket di ascolto interessato non ce n’è nessuna, il server si blocca in attesa che ne arrivi una, così da poterla accettare).

Una volta accettata una richiesta di connessione, questa viene rimossa dalla coda di attesa e la primitiva restituisce il descrittore di un nuovo socket, detto **socket di comunicazione**. In caso di errore, restituisce -1.

Vediamo adesso un esempio di codice del server:

Dichiaro:

* *ret* che userò per mettere valori di ritorno;
* *len* che userò per metterci dimensioni;
* *sd* e *new\_sd* che conterranno rispettivamente il descrittore del socket di ascolto e il descrittore del socket di comunicazione;
* *my\_addr* e *client\_addr* che conterranno rispettivamente l’indirizzo del socket di ascolto e l’indirizzo del socket del client di cui si accetterà la richiesta di connessione.

A questo punto, creo il socket di ascolto, e in *sd* finirà il descrittore di tale socket. Metto poi in *my\_addr* quello che io programmatore voglio sarà l’indirizzo del socket:

* Come famiglia di protocolli che verrà poi usata nella comunicazione, AF\_INET come sempre;
* Come numero di porta, un numero arbitrario (basta che non sia tra 0 e 1023, che sono numeri di porta riservati ad applicazioni di rete “well-known”). Questo va espresso in network order (e a tale scopo uso la funzione *htons,* così da convertire in network order “4242”, un letterale memorizzato in formato host order; *short* perché il numero di porta è un numero a 16 bit).
* Come IP, quello dell’host su cui gira il server. Questo va espresso in network order e in formato numerico (e a tale scopo uso la funzione *inet\_pton*, così da convertire in formato numerico l’indirizzo IP indicato in formato presentation. Il risultato verrà poi correttamente memorizzato in *my\_addr.sin\_add*r in network order)

Notare però che prima ho azzerato tutta la variabile *my\_addr*.   
In generale, quando si vuole scrivere in una certa area di memoria, è buona norma prima metterla tutta a 0 con memset. In questo modo, nel caso in cui non si scrivesse tutta, nella parte non scritta si ha 0, e non valori a caso che potrebbero portare a comportamenti indefiniti, senza capire così dove stanno gli errori.

Fatto ciò, assegno questo indirizzo al socket creato (notare il cast, dovuto a quanto si è detto prima parlando della *bind*).

Successivamente, indico che tale socket è di ascolto, e faccio sì che si crei una coda di attesa di dimensione 10.

Infine, accetto una richiesta di connessione arrivata sul socket di ascolto. In *client\_addr* finirà l’indirizzo del socket del client la cui richiesta è stata appena accettata, mentre in *new\_sd* finirà il descrittore del socket di comunicazione che la *accept()* avrà creato per permettere lo scambio di dati con tale client (notare il cast e il fatto che si è dovuto prima assegnare ad una variabile la dimensione in byte di *client\_addr* e poi passare l’indirizzo di tale variabile, entrambe le cose dovute a quanto si è detto prima parlando della *accept*).

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

int main(){

int ret, sd, new\_sd, len;

struct sockaddr\_in my\_addr, client\_addr;

sd = socket(AF\_INET, SOCK\_STREAM, 0);

memset(&my\_addr, 0, sizeof(my\_addr));

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(4242);

inet\_pton(AF\_INET, "192.168.4.5", &my\_addr.sin\_addr);

ret = bind(sd, (struct sockaddr\*)&my\_addr, sizeof(my\_addr));

ret = listen(sd, 10);

len = sizeof(client\_addr);

new\_sd = accept(sd, (struct sockaddr\*)&client\_addr, &len);

…

}

**CLIENT**

**PRIMITIVA SOCKET()**

Come prima cosa, il client crea un socket usando la primitiva socket(). A questo punto, non è necessario che chiami la primitiva bind(), visto che è il sistema operativo ad occuparsi di assegnare un indirizzo al socket creato, scegliendo in particolare un numero di porta libero casuale compreso tra 1024 e 65535.

Anche nel caso del server, una volta chiamata la primitiva socket(), il sistema operativo assegna un indirizzo al socket di ascolto creato. Tuttavia, il server ha bisogno di chiamare la primitiva bind(), assegnando al socket di ascolto un indirizzo scelto da lui, perché questo poi sarà l’indirizzo che pubblicizzerà ai client e al quale tali client manderanno una richiesta di connessione (altrimenti se ogni volta il socket di ascolto ottenesse un indirizzo diverso, i client non saprebbero dove mandare la propria richiesta di connessione).

**PRIMITIVA CONNECT()**

A questo punto, il client usa il socket appena creato per mandare una richiesta di connessione al socket di ascolto del server. Questo lo fa attraverso la primitiva connect():

int connect(int sockfd, const struct sockaddr\* addr, socklen\_t addrlen);

* ***sockfd*:** si passa il descrittore del socket del client;
* ***addr*:** si passa un puntatore a una struttura di tipo *sockaddr\_in* contenente l’indirizzo del socket di ascolto del server cui si vuole mandare una richiesta di connessione. Visto però che la funzione accetta solo *sockaddr\**, va fatto un cast.
* ***addrlen:*** si passa la dimensione in byte della struttura di tipo *sockaddr\_in* del punto precedente.

E’ una primitiva bloccante: una volta invocata, il client si blocca fino a quando il server non accetterà la richiesta di connessione mandata.

La primitiva restituisce 0 se tutto è andato bene; -1 altrimenti.

Vediamo adesso un esempio di codice del client:

Dichiaro:

* *ret* che userò per mettere valori di ritorno;
* *sd* che conterrà il descrittore del socket del client;
* *server\_addr* che servirà per contenere l’indirizzo del socket di ascolto del server cui si vorrà mandare una richiesta di connessione.

A questo punto, creo il socket del client, e in *sd* finirà il descrittore di tale socket.

Metto poi in *server\_addr* l’indirizzo del socket di ascolto del server cui si vorrà mandare una richiesta di connessione. Prima, però, azzero tutta la variabile *server\_addr*, per i motivi detti in precedenza parlando del codice del server.

A questo punto, mando una richiesta di connessione al socket di ascolto del server (notare il cast, dovuto a quanto si è detto prima parlando della *connect*).

#include <arpa/inet.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

int main(){

int ret, sd;

struct sockaddr\_in server\_addr;

sd = socket(AF\_INET, SOCK\_STREAM, 0);

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(4242);

inet\_pton(AF\_INET, "192.168.4.5", &server\_addr.sin\_addr);

ret = connect(sd, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));

…

}

**NB:** il “numero di porta” viene dunque assegnato ad un processo nel momento in cui viene creato il socket per tale processo. Questo numero viene assegnato automaticamente dal sistema operativo, ma usando la primitiva *bind* può essere cambiato con uno scelto arbitrariamente dal programmatore.

**SCAMBIO DI DATI ATTRAVERSO I SOCKET** *14-11-22*

Torniamo al nostro esempio di applicazione client-server, considerando il caso in cui vengano usati socket di tipo SOCK\_STREAM. Una volta instaurata una connessione tra i due processi, si dice che il socket usato dal client per mandare la richiesta di connessione e il socket di comunicazione che è stato creato a seguito della chiamata del server alla primitiva *accept()* sono ***connessi*** tra loro. Questi saranno i socket che client e server useranno per lo scambio di dati, chiamando le primitive ***send()*** e ***recv()*.**

Prima di vedere queste primitive, però, bisogna dire una cosa. In generale, per ogni socket, il sistema operativo crea:

* Un **buffer di ricezione,** da cui verranno letti i messaggi destinati a tale socket;
* Un **buffer di invio**, in cui verranno messi i messaggi che si vuole vengano inviati attraverso tale socket ad un altro socket.

La *send()* e la *recv()* citate prima (e, in generale, tutte le primitive per lo scambio di dati che vedremo) non fanno altro che leggere/scrivere su questi buffer. Saranno poi i sottostanti livelli della pila protocollare ad occuparsi che effettivamente un messaggio che si trova nel buffer di invio di un socket mittente finisca nel buffer di ricezione del socket destinatario, ma di questo il processo non se ne deve preoccupare.

**PRIMITIVA SEND()**

Permette a un processo di inviare un messaggio attraverso un socket connesso (che verrà mandato al socket cui il socket connesso è, appunto, connesso).

ssize\_t send(int sockfd, const void\* buf, size\_t len, int flags);

* ***sockfd*:** si passa il descrittore del socket connesso;
* ***buf*:** si passa un puntatore al buffer contenente il messaggio da inviare;
* ***len:*** si passa la dimensione in byte del messaggio da inviare;
* ***flags:*** serve per indicare opzioni aggiuntive. Per non indicarne nessuna, si passa *0*.

La primitiva restituisce il numero di byte inviati, e cioè quanti byte del messaggio è riuscita a copiare nel buffer di invio del socket connesso. Può infatti succedere che la primitiva copi meno di *len* byte, restituendo così un valore < *len* (per esempio, se il buffer si satura, perché magari ci sono state altre *send* prima e nel buffer sono ancora rimasti dei byte da mandare al socket destinatario). La soluzione allora è mettere la *send* in un ciclo che ha come condizione di uscita “*valore\_restituito\_da\_send* == *len*” e in cui ogni volta si incrementa *buf* e si decrementa *len* di *valore\_restituito\_da\_send*, e cioè del numero di byte che si è riusciti a inviare, così che ogni volta si cerchi di inviare i byte successivi a quelli che sono stati già inviati. In caso di errore, viene restituito -1.

La primitiva è bloccante: il processo si blocca finché non viene inviato almeno un byte del messaggio (questo significa che, se il buffer di invio fosse pieno, il processo dovrà attendere anche che si abbia almeno un byte libero, così che almeno un byte possa essere inviato).

*es:*

int ret, sd, len;

char buffer[1024];

… // Tutta la fase di inizializzazione di un socket e instaurazione di una connessione

strcpy(buffer, "Hello!");

len = strlen(buffer);

ret = send(sd, (void\*)buffer, len, 0);

if(ret < len){

… // Gestione dell’errore o dell’invio parziale

}

**PRIMITIVA RECV()**

Permette a un processo di ricevere un messaggio arrivato su un socket connesso (che l’avrà mandato il socket cui il socket connesso è, appunto, connesso).

ssize\_t recv(int sockfd, const void\* buf, size\_t len, int flags);

* ***sockfd*:** si passa il descrittore del socket connesso;
* ***buf*:** si passa un puntatore al buffer in cui salvare il messaggio da ricevere;
* ***len:*** si passa la dimensione in byte del messaggio da ricevere;
* ***flags:*** serve per indicare opzioni aggiuntive. Per non indicarne nessuna, si passa *0*.

La primitiva restituisce il numero di byte ricevuti, e cioè quanti byte del messaggio è riuscita a copiare dal buffer di ricezione del socket connesso. Può infatti succedere che la primitiva copi meno di *len* byte, restituendo così un valore < *len* (per esempio, se il messaggio non è ancora arrivato tutto). La soluzione allora è mettere la *recv* in un ciclo che ha come condizione di uscita “*valore\_restituito\_da\_recv* == *len*” e in cui ogni volta si incrementa *buf* e si decrementa *len* di *valore\_restituito\_da\_recv*, e cioè del numero di byte ricevuti, così che ogni volta si cerchi di ricevere i byte successivi a quelli che sono stati già ricevuti. In caso di errore, viene restituito -1.

La primitiva è bloccante: il processo si blocca finché non viene ricevuto almeno un byte del messaggio (questo significa che, se il buffer di ricezione è vuoto, il processo dovrà attendere anche che si riempia con almeno un byte, così che almeno un byte possa essere ricevuto)

*es:*

int ret, sd, len;

char buffer[1024];

… // Tutta la fase di inizializzazione di un socket e instaurazione di una connessione

len = 20;

ret = recv(sd, (void\*)buffer, len, 0);

if(ret < len){

… // Gestione dell’errore o della ricezione parziale

}

Passando come *flags* MSG\_WAITALL, il processo si blocca fino a quando non vengono ricevuti tutti i byte del messaggio (e dunque non c’è il problema di gestire un’eventuale ricezione parziale)

ret = recv(sd, (void\*)buffer, len, MSG\_WAITALL);

if(ret < len){

… // Gestione dell’errore

}

**PRIMITIVA CLOSE()**

Permette a un processo di chiudere un socket. Una volta chiuso, tale socket non può più essere usato per inviare/ricevere dati.

#include <unistd.h>

int close(int fd);

* ***sockfd*:** si passa il descrittore del socket da chiudere;

La primitiva restituisce 0 se tutto è andato bene; -1 in caso di errore.

La *close()* dealloca tutte le risorse che erano state allocate per il socket, dunque viene usata per chiudere una connessione tra due processi. Supponiamo infatti che il client non voglia più comunicare con il server: quello che fa, allora, è chiamare la *close()* sul proprio socket. Se tutto è andato bene, quando il server chiamerà la *recv()* sul proprio socket di comunicazione che prima era connesso al socket del client, gli verrà restituito *0.* In questo modo, il server capisce che il client ha chiuso il suo socket, e dunque chiamerà a sua volta la *close()* sul proprio socket di comunicazione, chiudendo così effettivamente la connessione tra i due processi.

**GESTIONE DEGLI ERRORI**

Tutte le primitive viste prima restituiscono -1 in caso di errore. In più, modificano il valore della variabile globale ***errno***, che dunque può essere letta per capire quale errore si è verificato (sul manuale di ogni primitiva c’è l’elenco di tutti gli errori possibili e il modo in cui *errno* viene modificato a seconda dell’errore).

*es.*

#include <errno.h>

…

ret = bind(sd, (struct sockaddr\*)&my\_addr, sizeof(my\_addr));

if (ret == -1) {

if (errno == *EADDRINUSE*) {

EADDRINUSE: indica che si sta cercando di assegnare ad un socket un indirizzo già assegnato ad un altro.

EINVALID: indica che l’indirizzo che si sta cercando di assegnare ad un socket non è valido.

… // Gestione errore

}

if (errno == *EINVAL*) {

… // Gestione errore

}

}

In molti casi, tuttavia, non ci interessa tanto gestire l’errore, quanto sapere semplicemente che errore si è verificato. A tale scopo si usa la funzione ***perror(stringa*)**, che legge il valore della variabile *errno,* capisce che errore si è verificato e stampa a video la *stringa* passata come argomento seguita da una spiegazione dell’errore.

*es.*

#include <stdio.h>

…

ret = bind(sd, (struct sockaddr\*)&my\_addr, sizeof(my\_addr));

if (ret == -1) {

Una volta stampata a video la descrizione dell’errore che si è verificato, termino il processo (e a tale scopo si usa la funzione *exit()*) .

perror(“Error: “);

exit(1);

}

**PROTOCOLLI TEXT E PROTOCOLLI BINARY**

Supponiamo che un processo mittente voglia mandare una struttura dati ad un processo destinatario. Può farlo essenzialmente in due modi:

* **Adottando un protocollo text:** il processo mittente prende i vari campi della struttura, li mette insieme convertendo tutto in una stringa che deve avere un formato concordato con il processo destinatario, e manda questa stringa.
* **Adottando un protocollo binary:** il processo mittente manda *singolarmente* i vari campi della struttura (senza prima convertirli in stringa). La dimensione dei campi e l’ordine con cui questi vengono mandati devono essere concordati tra processo mittente e processo destinatario.

In ogni caso, il processo mittente non può mandare la struttura così com’è, altrimenti si avrebbero tre problemi:

* *Dimensione dei dati*: i campi di tipo non terminante in *‘\_t’* potrebbero essere memorizzati su un numero di byte diverso tra host mittente e host destinatario.
* *Endianness*: campi che occupano più di 1 byte potrebbero essere memorizzati con un endianness diversa tra host mittente e host destinatario;
* *Padding*: passando la struttura così com’è verrebbero passati anche dei byte non significativi, sprecati per garantire l’allineamento dei vari campi, e visto che l’allineamento varia a seconda del compilatore usato (e quindi potrebbe variare tra host mittente e host destinatario), questi byte potrebbero dare fastidio.

E’ importante allora che il processo mittente si occupi della cosiddetta **serializzazione della struttura dati,** e cioè che converta questa struttura in un certo modo concordato con il processo destinatario, così che questo sappia poi ricostruire la struttura dati di partenza, effettuando la cosiddetta **deserializzazione**.

**SERIALIZZAZIONE/DESERIALIZZAZIONE IN UN PROTOCOLLO TEXT**

Innanzitutto, introduciamo due funzioni che saranno utili:

* **sprintf(char\* *str*, const char\* *formato*, *lista\_argomenti*):** come la printf, con la differenza che stampa sull’array di caratteri puntato da *str*.
* **sscanf(const char\* str, const char\* *formato,* *lista\_indirizzi*):** come la scanf, con la differenza che legge dalla stringa *str*.

Supponiamo che il processo mittente voglia inviare una struttura dati. Fa questo:

Questo è il buffer usato per l’invio.

char buffer[1024]

struct temp{

int a;

Questa è la dichiarazione della struttura dati.

char b;

};

Questa è la variabile contenente la struttura dati che il processo mittente vuole mandare.

struct temp t;

t.a = 10;

t.b = ‘c’;

Supponiamo che processo mittente e processo destinatario si siano messi d’accordo sull’adottare un protocollo text e che nella stringa va prima messo l’*int* e poi il *char*, separati da uno spazio. Per serializzare la struttura, il processo mittente chiama allora la **sprintf**, nel modo indicato a fianco, e così verrà messa in *buffer* la stringa “%d %c”, mettendo al posto degli specificatori di formato i campi *t.a* e *t.b* stampati rispettivamente come intero e char (e stampati ==   
si ha una sequenza di caratteri ASCII, dunque nel *buffer* si avrà   
{‘1’, ‘0’, ‘ ‘, ‘c’, ‘\0}).

sprintf(*buffer*, "%d %c", t.a, t.b);

… // Invio stringa

Il processo destinatario, per ricevere e memorizzare correttamente la struttura dati, fa questo:

Questo è il buffer usato per la ricezione.

char buffer[1024]

struct temp{

int a;

Questa è la dichiarazione della struttura dati.

char b;

};

Questa è la variabile in cui il processo destinatario salverà la struttura dati mandata dal processo mittente.

struct temp t

… // Ricezione stringa in *buffer*

Il processo destinatario sa il formato della stringa ricevuta. Per deserializzare la struttura, allora, chiama la ***sscanf***, nel modo indicato a fianco, e così memorizzerà la prima sequenza di caratteri in *t.a* come un intero; poi c’è lo spazio; e poi memorizzerà la seconda sequenza di caratteri in *t.b* come un carattere. In questo modo, in *t* finirà la struttura dati di partenza.

he memorizzerà nel modo indicato dalla stringa *formato* il contenuto della stringa *buffer,*  prende i vari campi della struttura e li mette insieme convertendo tutto in una stringa che ha il formato stabilito, inserendola in *buffer*   
(in particolare, per i campi numerici, si ha che ogni cifra del numero verrà convertita nel carattere ASCII corrispondente, e quindi in *buffer* si avrà   
{‘1’, ‘0’, ‘ ‘, ‘c’, ‘\0}).

sscanf(*buffer*, "%d %c", &t.a, &t.b);

**SERIALIZZAZIONE/DESERIALIZZAZIONE IN UN PROTOCOLLO BINARY**

Supponiamo che il processo mittente voglia inviare una struttura dati. Fa questo:

Questa è la dichiarazione della struttura dati. Notare i tipi utilizzati: il processo mittente manderà un campo alla volta, dunque serve che la dimensione di ogni campo sia concordata tra processo mittente e processo destinatario (altrimenti potrebbe succedere che, nel mandare per esempio un *int*, il processo mittente manda 8 byte, visto che quella è la dimensione di un *int* sulla sua architettura; il processo destinatario invece si aspetta 4 byte, visto che è quella la dimensione di un *int* sulla sua architettura, e quindi lo memorizzerebbe male). **Questo viene garantito usando i tipi terminanti in *“\_t*”**, la cui dimensione è fissa su tutte le architetture.

struct temp{

uint32\_t a;

uint8\_t b;

};

struct temp t

t.a = 10;

Questa è la variabile contenente la struttura dati che il processo mittente vuole mandare.

t.b = ‘c’;

Supponiamo che processo mittente e processo destinatario si siano messi d’accordo sull’adottare un protocollo binary e sull’inviare prima l’*uint32\_t* e poi l’*uint8\_t*. Innanzitutto, il processo destinatario deve convertire tutti i campi che occupano più di un byte da host order a network order (visto che questo è il formato usato in rete). Fatto ciò, manda al processo destinatario i campi della struttura, nell’ordine stabilito.

t.a = htonl(t.a);

ret = send(new\_sd, (void\*)&t.a, sizeof(uint32\_t), 0);

ret = send(new\_sd, (void\*)&t.b, sizeof(uint8\_t), 0);

Il processo destinatario, per ricevere e memorizzare correttamente la struttura dati, fa questo:

Questo è il buffer usato per la ricezione.

char buffer[1024]

struct temp{

int a;

Questa è la dichiarazione della struttura dati.

char b;

};

Questa è la variabile in cui il processo destinatario salverà la struttura dati mandata dal processo mittente.

struct temp t

ret = recv(new\_sd, (void\*)&t.a, sizeof(uint32\_t), 0);

if(ret < sizeof(uint32\_t)){

Il processo destinatario prima si aspetta l’*uint32\_t*, dunque attende di riceverlo (infatti attende un messaggio di dimensione “sizeof(*uint32\_t)”*) e, una volta ricevuto, lo converte da network order ad host order e se lo salva in *t.a*.

… // Gestione dell’errore o della ricezione parziale

}

t.a = ntohl(t.a);

ret = recv(new\_sd, (void\*)&t.b, sizeof(uint8\_t), 0);

Il processo destinatario poi si aspetta l’*uint8\_t*, dunque attende di riceverlo (infatti attende un messaggio di dimensione “sizeof(*uint8\_t*)”) e se lo salva in *t.b* (non bisogna convertirlo da network order ad host order, essendo un solo byte).

if(ret < sizeof(uint8\_t)){

… // Gestione dell’errore o della ricezione parziale

}

Tirando le somme:

* Il protocollo text è orientato al testo, dunque è conveniente usarlo se si deve mandare principalmente testo. Nel mandare strutture dati, invece, c’è il problema di dover prima convertire ogni campo in stringa e poi riconvertirlo nel tipo di partenza, e questo richiede del tempo sia lato processo mittente, sia lato processo destinatario.
* Il protocollo binary è orientato alle strutture dati, dunque è conveniente usarlo se si devono mandare principalmente strutture dati.

**SERVER CONCORRENTI** *21-11-22*

Finora abbiamo visto come fare un **server iterativo,** e cioè un server che serve una sola richiesta per volta.

Vediamo ora come creare un **server concorrente**, e cioè un server che serve più richieste alla volta. Questo lo fa creando altri processi, detti **processi figli,** che sono dei cloni di se stesso, detto dunque **processo padre.**   
Quello che fa il processo padre è dunque semplicemente accettare le richieste di connessione che arrivano sul socket di ascolto e creare, per ogni richiesta accettata, un processo figlio, che si occuperà effettivamente di servirla.

**PRIMITIVA FORK()**

Per clonare se stesso, il processo padre chiama la primitiva **fork():**

#include <unistd.h>

pid\_t fork();

La primitiva restituisce:

* Nel processo padre, il *PID* del processo figlio creato;
* Nel processo figlio, *0*;
* *-1* in caso di errore.

Per capire come sfruttarla, vediamo un esempio di server concorrente:

#include …

int main () {

int ret, sd, new\_sd, len;

struct sockaddr\_in my\_addr, client\_addr;

pid\_t pid;

sd = socket(AF\_INET, SOCK\_STREAM, 0);

… // Creazione indirizzo socket di ascolto

ret = bind(sd, (struct sockaddr\*)&my\_addr, sizeof(my\_addr));

ret = listen(sd, 10);

len = sizeof(client\_addr);

while(1) {

new\_sd = accept(sd, (struct sockaddr\*)&client\_addr, &len);

pid = fork();

if (pid == -1) {

… // Gestione errore

}

if (pid == 0) { // Sono nel processo figlio

close(sd);

… // Gestione richiesta (*send*, *recv*, …)

close(new\_sd);

exit(0);

}

close(new\_sd); // Sono nel processo padre

}

Una volta creato il socket di ascolto, si entra in un ciclo infinito. Come prima cosa, si accetta una richiesta di connessione arrivata sul socket di ascolto e si clona il processo padre. Verrà così creato un processo figlio che è identico al padre, e cioè esegue lo stesso codice partendo dall’istruzione successiva alla fork() e presenta una copia di tutte le strutture dati del padre, compresi i socket. Vanno allora gestiti 3 casi:

* **Se *fork* restituisce -1**, c’è stato un errore.
* **Se *fork* restituisce 0,** allora si è nel processo figlio. Come prima cosa, allora, va chiuso il suo socket di ascolto, visto che non gli servirà. Infatti, l’unica cosa di cui si deve occupare il processo figlio è servire la richiesta, usando il proprio socket di comunicazione. Una volta servita, si chiude anche questo socket e si termina il processo figlio;
* **Se *fork* restituisce qualcosa di diverso**, allora si è nel processo padre. Si chiude allora il suo socket di comunicazione, visto che di servire la richiesta se ne occuperà il figlio, e si torna all’inizio del while, con il processo padre che accetterà così una nuova richiesta di connessione arrivatagli sul suo socket di ascolto.

Un server concorrente dunque sembra più vantaggioso rispetto ad un server iterativo. Tuttavia, ci sono anche alcuni svantaggi da considerare:

* Innanzitutto, la clonazione di un processo è un’operazione abbastanza pesante da un punto di vista computazionale, quindi ci vuole del tempo per eseguirla.
* Inoltre, un server del genere potrebbe essere soggetto ad un attacco di tipo *fork bomb*: si mandano in massa tante richieste di connessione al socket di ascolto del processo padre, che crea così per ogni richiesta un processo figlio. La memoria dell’host su cui gira il server si inizia così a riempire di tutti questi processi figli fino a quando, una volta riempita tutta, il server non è più grado di gestire altre richieste (visto che il processo padre dovrebbe creare un altro processo figlio per gestirla, ma non può farlo visto che non c’è spazio in memoria in cui mettere questo processo figlio).   
  Un modo per prevenire un attacco di tipo *fork bomb* potrebbe essere porre un limite al numero di processi figli attivi (e cioè, di processi figli non ancora terminati), e dunque il processo padre crea un nuovo figlio solo se non si è raggiunto questo limite. Si potrebbe allora mantenere in memoria un contatore che tiene il conto dei processi figli attivi, e così:
  + Un processo figlio, poco prima di terminare, decrementa questo contatore;
  + Il processo padre, prima di creare un nuovo figlio, legge il valore di questo contatore e, se è sotto una certa soglia, lo incrementa e crea il figlio.

Chiaramente l’accesso a questo contatore va fatto in mutua esclusione, dunque usando i semafori, essendo un accesso ad una risorsa condivisa tra tutti i processi.

**SOCKET NON BLOCCANTI**  *28-11-22*

I socket visti finora sono **socket bloccanti,** in quanto:

* *connect()* blocca il processo finché il socket di ascolto remoto non accetta la richiesta di connessione;
* *accept()* blocca il processo finché non viene accettata una richiesta di connessione;
* *send()* blocca il processo fino a quando non viene inviato almeno un byte del messaggio;
* *recv()* blocca il processo fino a quando non viene ricevuto almeno un byte del messaggio (o tutti i byte del messaggio, nel caso in cui fosse stato usato il flag MSG\_WAITALL).

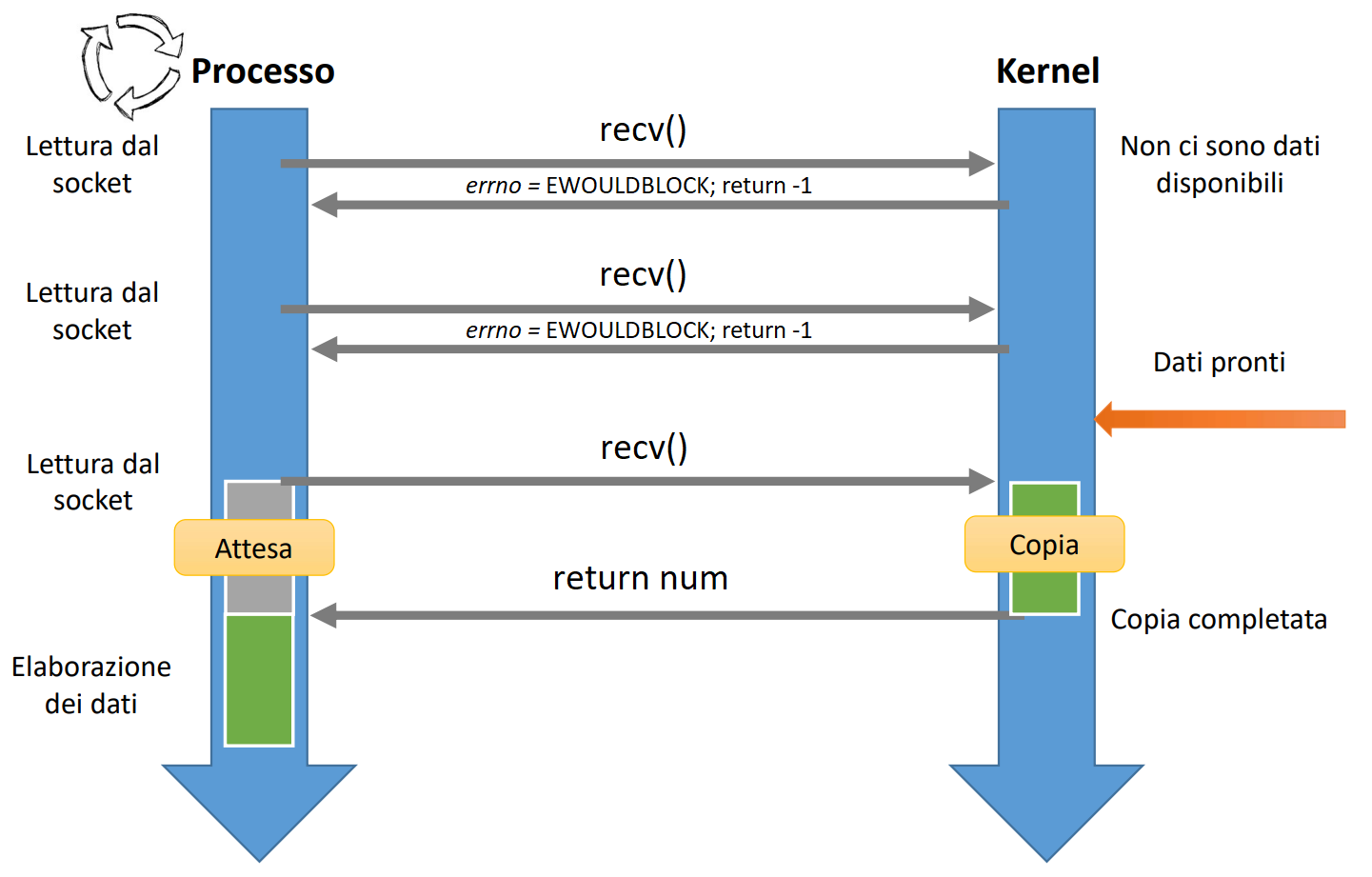
E’ possibile creare anche un **socket non bloccante,** e cioè tale che il processo non si blocca nei casi descritti sopra. Questo si fa passando *SOCK\_STREAM | SOCK\_NONBLOCK* come parametro *type* alla primitiva *socket*():

socket(AF\_INET, SOCK\_STREAM | SOCK\_NONBLOCK, 0);

Il comportamento delle primitive allora cambia:

* *connect()*: se la richiesta di connessione non viene accettata subito, la primitiva imposta *errno* a EINPROGRESS e restituisce -1, ridando il controllo al processo chiamante;
* *accept()*: se al momento non ci sono richieste di connessione da accettare sul socket, la primitiva semplicemente imposta *errno* a EWOULDBLOCK e restituisce -1, ridando il controllo al processo chiamante, che dunque dovrà poi richiamare la primitiva, nella speranza che questa volta ci sia qualche richiesta di connessione da accettare;
* *send()*: se al momento non è possibile inviare almeno un byte del messaggio, la primitiva semplicemente imposta *errno* a EWOULDBLOCK e restituisce -1, ridando il controllo al processo chiamante, che dunque dovrà poi richiamare la primitiva, nella speranza che questa volta sia possibile inviare qualcosa;
* *recv()*: se al momento non è possibile ricevere almeno un byte del messaggio, la primitiva semplicemente imposta *errno* a EWOULDBLOCK e restituisce -1, ridando il controllo al processo chiamante, che dunque dovrà poi richiamare la primitiva, nella speranza che questa volta sia possibile ricevere qualcosa;

*Esempio* *di processo che usa un socket non bloccante:*

**

Il processo chiama la *recv()* sul proprio socket, ma non c’è nessun byte nel buffer di ricezione. La primitiva allora semplicemente imposta *errno* a EWOULDBLOCK, restituisce -1 e ridà il controllo al processo.

Poco dopo, il processo richiama la *recv()* sul proprio socket, ma non c’è ancora nessun byte nel buffer di ricezione. La primitiva allora imposta *errno* a EWOULDBLOCK, restituisce -1 e ridà il controllo al processo.

Dopo un po’, nel buffer di ricezione arrivano i byte del messaggio (tutti o solo alcuni, non importa). Quando allora il processo richiama la recv() sul proprio socket, visto che ora si può fare una ricezione parziale/totale, la primitiva si occupa di farla.

Il vantaggio dunque è che, prendendo in considerazione per esempio la primitiva *recv()*, mentre usando un socket bloccante il processo si sarebbe bloccato già alla prima chiamata della primitiva in attesa che arrivasse qualcosa nel buffer di ricezione, usando un socket non bloccante invece si blocca solo quando ci sono dati da ricevere nel buffer di ricezione e solo per il tempo necessario a riceverli. Il processo dunque può sfruttare il tempo in cui non è rimasto bloccato per fare altro (chiaramente che non riguardi i dati da ricevere, visto che non li ha ancora).

**I/O MULTIPLEXING**

Torniamo a considerare i socket bloccanti. Il grosso problema di questi è che, se ci si trovasse in uno dei quattro casi visti prima, il processo si bloccherebbe, e non può fare altro. Ci vorrebbe allora un modo per sapere in anticipo se, chiamando una certa primitiva su un certo socket, il processo si bloccherebbe o meno, e questo modo è proprio l’**I/O multiplexing**. Prima, però, introduciamo alcuni concetti preliminari.

Introduciamo innanzitutto il concetto di **socket *pronto*:**

* Si dice che un **socket è *pronto in lettura*** se, chiamando una primitiva che accede a tale socket in lettura, il processo non si bloccherebbe, e quindi se (uno di questi casi):
  + Nel caso di socket connesso, c’è almeno un byte da leggere nel buffer di ricezione, c’è un errore (visto che, in questo caso, recv() subito restituisce -1) o il socket cui questo socket è connesso è stato chiuso (visto che, in questo caso, recv() subito restituisce 0);
  + Nel caso di socket di ascolto, c’è almeno una richiesta di connessione nella coda di attesa;
* Si dice che un **socket è pronto *in scrittura*** se, chiamando una primitiva che accede a tale socket in scrittura, il processo non si bloccherebbe, e quindi se (uno di questi casi):
  + Nel caso di socket connesso, c’è almeno un byte libero nel buffer di invio o c’è un errore (visto che, in questo caso, send() subito restituisce -1).

Introduciamo poi il concetto di **set (insieme) di descrittori di socket:** questo possiamo vederlo come un contenitore in cui è possibile mettere/togliere descrittori di socket. Un set si crea attraverso una variabile di tipo ***fd\_set*** e si può manipolare usando le seguenti macro:

* **FD\_SET(int fd, fd\_set\* set):** aggiunge il descrittore di socket *fd* al set puntato da *set;*
* **FD\_ISSET(int fd, fd\_set\* set):** controlla se il descrittore di socket *fd* è presente nel set puntato da *set;*
* **FD\_CLR(int fd, fd\_set\* set):** rimuove il descrittore di socket *fd* dal set puntato da *set*;
* **FD\_ZERO(fd\_set\* set):** svuota il set puntato da *set*.

L’ultimo concetto da introdurre è la **primitiva select(),** che permette di controllare i descrittori presenti in più set, vedendo quali sono quelli pronti (e cioè, descrittori di socket pronti):

#include <sys/time.h>

#include <sys/types.h>

#include <unistd.h>

int select(int nfds, fd\_set\* readfds, fd\_set\* writefds, fd\_set\* exceptfds, struct timeval\* timeout);

* ***nfds*:** si passa il descrittore di socket maggiore tra quelli presenti nei successivi tre set, + 1 (ricordiamo infatti che un descrittore è un numero intero);
* ***readfds*:** si passa un puntatore al set di descrittori di socket da controllare in lettura (NULL se non si vuole controllare alcun socket in lettura);
* ***writefds:*** si passa un puntatore al set di descrittori di socket da controllare in scrittura (NULL se non si vuole controllare alcun socket in scrittura);
* ***exceptfds:*** si passa un puntatore al set di descrittori di socket da controllare per le eccezioni (a noi non interessa, quindi passeremo NULL).
* ***timeout:*** si passa un puntatore ad una struttura di tipo *timeval*, contenente un intervallo di timeout (NULL se non si vuole alcun intervallo di timeout).

La primitiva lascia in questi set solo i descrittori pronti (chiaramente, nel set puntato da *readfds* lascerà solo i descrittori pronti in lettura, così come nel set puntato da *writefds* lascerà solo i descrittori pronti in scrittura. Come *exceptfds* non passeremo nulla dunque non ci interessa) e restituisce il numero di descrittori lasciati. Restituisce -1 in caso di errore.

La primitiva è bloccante: se nei tre set non c’è alcun descrittore pronto, il processo si blocca finché non ne diventa pronto almeno uno o finché non scade l’intervallo di timeout indicato.

La struttura *timeval* è fatta così:

struct timeval{

long tv\_sec; // conterrà i secondi dell’intervallo

long tv\_usec; // conterrà i microsecondi dell’intervallo

};

Ora possiamo vedere in cosa consiste l’I/O multiplexing. Supponiamo che un processo voglia controllare un certo numero di socket in lettura:

1. Come prima cosa, il processo mette in un set i descrittori dei socket da controllare in lettura;
2. A questo punto, chiama la select(), passando tra i vari argomenti un puntatore a questo set;
3. La primitiva lascerà nel set passato solo i descrittori pronti, e così il processo sa su quali socket può operare, senza il rischio che possa bloccarsi (essendo, appunto, pronti).

Visto poi che la select() fa modifiche al set, è bene farne una copia prima di chiamare la select(), così se il processo vuole controllare nuovamente gli stessi socket non deve ricostruire il set, ma ce l’ha già.

Vediamo ora un esempio di server che fa uso dell’I/O multiplexing:

#include …

int main () {

**fd\_set** **master**;

**fd\_set read\_fds**;

**int sdmax**;

int ret, listener, new\_sd, addrlen, nbytes, i;

char buf[1024];

struct sockaddr\_in sv\_addr, cl\_addr;

**FD\_ZERO(&master);**

**FD\_ZERO(&read\_fds);**

listener = socket(AF\_INET, SOCK\_STREAM, 0);

sv\_addr.sin\_family = AF\_INET;

sv\_addr.sin\_addr.s\_addr = INADDR\_ANY;

sv\_addr.sin\_port = htons(20000);

ret = bind(listener, (struct sockaddr\*)&sv\_addr, sizeof(sv\_addr));

ret = listen(listener, 10);

**FD\_SET(listener, &master);**

**sdmax = listener;**

while(1) {

read\_fds = master;

select(sdmax + 1, &read\_fds, NULL, NULL, NULL);

for (i = 0; i <= sdmax; i++){

if (**FD\_ISSET**(i, &read\_fds)){

if (i == listener){

addrlen = sizeof(cl\_addr);

new\_sd = accept(listener, (struct sockaddr\*)&cl\_addr, &addrlen);

**FD\_SET**(new\_sd, &master);

if (new\_sd > sdmax){

sdmax = new\_sd;

}

}

else{

nbytes = recv(i, buf, sizeof(buf), 0);

…

close(i);

**FD\_CLR**(i, &master);

}

}

}

}

return 0;

}

Supponiamo che questo server abbia bisogno di controllare dei socket solo in lettura. Dichiaro innanzitutto varie variabili, tra cui:

* ***master***, che sarà il set principale, contenente i descrittori dei socket che si vogliono controllare in lettura;
* ***read\_fds***, che sarà il set copia del set *master* e su cui lavorerà la select();
* ***sdmax****,* che conterrà il descrittore di socket maggiore tra quelli presenti nel set *master.*

Fatto ciò, azzero preventivamente i due set (per evitare che la zona di memoria riservata ai set contenga valori a caso che possono dare problemi).

Una volta creato e inizializzato il socket di ascolto, va messo tra quelli da controllare in lettura (per sapere quando è arrivata o meno una richiesta di connessione). Metto allora il suo descrittore nel set *master* e assegno il valore di questo descrittore alla variabile *sdmax*, visto che al momento è il descrittore di socket maggiore tra quelli presenti nel set *master* (essendo l’unico descrittore di socket presente nel set).

A questo punto, entro in un ciclo while infinito. Come prima cosa, copio il set *master* nel set *read\_fds*, ed è su questo set che chiamo la select(). Questa lascerà in *read\_fds* i descrittori pronti in lettura, ma non ci dice quali sono questi. L’unico modo per saperlo è fare un altro ciclo, dove prima vedo se il descrittore 0 è presente in questo set; se non c’è, vedo se è presente il descrittore 1; se non c’è vedo se è presente il descrittore 2… e continuo nel caso peggiore fino al descrittore *sdmax*, che era il descrittore maggiore tra quelli inizialmente presenti.

Una volta trovato un descrittore *i* che è stato lasciato nel set *read\_fds*, e dunque un descrittore pronto in lettura tra quelli da controllare, distinguo due casi:

INADDR\_ANY

* Se è il descrittore del socket di ascolto, significa che gli è arrivata una richiesta di connessione. Chiamo allora la *accept()* per accettarla e metto il descrittore del socket di comunicazione creato dalla primitiva nel set *master* (dunque aggiungo questo socket di comunicazione tra quelli da monitorare in lettura). Se poi il valore di questo descrittore è > *sdmax*, lo assegno ad *sdmax* (visto che questa variabile deve contenere il descrittore di socket maggiore tra quelli presenti nel set *master*);
* Altrimenti, significa che è il descrittore di un socket di comunicazione e che è arrivato qualcosa nel suo buffer di ricezione. Chiamo allora la *recv()* per fare la ricezione di ciò che è arrivato, ci faccio varie elaborazioni e supponendo non debba comunicare ulteriormente, chiudo il socket e tolgo il suo descrittore dal set *master* (visto che non lo devo più controllare, avendolo chiuso). Notare che potrei togliere proprio il descrittore *i == sdmax*, dunque *in teoria* dovrei aggiornare il valore di questa variabile. In pratica, affinché tutto funzioni, l’importante è che *sdmax* abbia un valore ≥ del descrittore di socket maggiore tra quelli presenti nel set *master.*

Qualunque sia il caso, una volta gestito si passa alle prossime iterazioni del for, per vedere se ci sono altri descrittori pronti in lettura ed agire di conseguenza.

Terminata l’ultima iterazione del for, si torna all’inizio del corpo del while, ripetendolo da capo, e si continua così all’infinito (notare che, ad ogni iterazione del while, il set *master* contenente i descrittori da controllare in lettura può avere più/meno descrittori rispetto all’iterazione precedente, visto che durante questa possono essere state servite richieste o accettata una nuova).

**NB:** anche lo *stdin* è un socket, il cui descrittore è *0* e che è pronto in lettura quando c’è *qualcosa* (e cioè, quando si è scritto *qualcosa* da tastiera e si è premuto *invio*). Questo può essere utile nel progetto.

**SOCKET UDP** *02-12-22*

Finora ci siamo concentrati sui socket di tipo SOCK\_STREAM, e cioè socket che garantiscono una comunicazione in cui verrà usato TCP come protocollo di trasporto. Vediamo ora i **socket di tipo SOCK\_DGRAM,** e cioè socket che garantiscono una comunicazione in cui verrà usato UDP come protocollo di trasporto.

Consideriamo un’applicazione client-server costituita da un processo client e un processo server, in esecuzione su host diversi, e supponiamo che vogliano comunicare usando il protocollo di trasporto UDP. Come prima cosa, allora, entrambi i processi creano il proprio socket di tipo SOCK\_DGRAM, in questo modo:

socket(AF\_INET, SOCK\_DGRAM, 0);

Il server assegna poi un indirizzo al proprio socket usando la *bind()* e lo pubblicizza al client. A questo punto, visto che il protocollo UDP non richiede che venga prima instaurata una connessione tra i due processi che vogliono comunicare tra loro, client e server possono già iniziare a scambiarsi messaggi, usando le primitive **sendto()** e **recvfrom()**.

**PRIMITIVA SENDTO()**

Permette a un processo di inviare un messaggio attraverso un socket a quello di indirizzo specificato:

ssize\_t sendto(int sockfd, const void\* buf, size\_t len, int flags, const struct sockaddr\* dest\_addr, socklen\_t addrlen);

* ***sockfd*:** si passa il descrittore del socket da usare per l’invio;
* ***buf*:** si passa un puntatore al buffer contenente il messaggio da inviare;
* ***len:*** si passa la dimensione in byte del messaggio da inviare;
* ***flags:*** serve per indicare opzioni aggiuntive. Per non indicarne nessuna, si passa *0*;
* ***dest\_addr:*** si passa un puntatore alla struttura di tipo *sockaddr\_in* contenente l’indirizzo del socket destinatario. Visto però che la funzione accetta solo *sockaddr\**, va fatto un cast;
* ***addrlen:*** si passa la dimensione in byte della struttura di tipo *sockaddr\_in* del punto precedente.

La primitiva restituisce il numero di byte inviati, e cioè quanti byte del messaggio è riuscita a copiare nel buffer di invio del socket da usare per l’invio. In caso di errore, restituisce -1.

La primitiva è bloccante: il processo si blocca finché non viene inviato almeno un byte del messaggio.

**PRIMITIVA RECVFROM()**

Permette a un processo di ricevere un messaggio arrivato su un socket:

ssize\_t recvfrom(int sockfd, const void\* buf, size\_t len, int flags, struct sockaddr\* src\_addr, socklen\_t\* addrlen);

* ***sockfd*:** si passa il descrittore del socket;
* ***buf*:** si passa un puntatore al buffer in cui salvare il messaggio da ricevere;
* ***len:*** si passa la dimensione in byte del messaggio da ricevere;
* ***flags:*** serve per indicare opzioni aggiuntive. Per non indicarne nessuna, si passa *0*.
* ***src\_addr:*** si passa un puntatore a una struttura di tipo *sockaddr\** in cui verrà salvato l’indirizzo del socket mittente. Visto però che la funzione accetta solo *sockaddr\**, va fatto un cast;
* ***addrlen:*** si passa un puntatore a una variabile contenente la dimensione in byte della struttura di tipo *sockaddr\_in* del punto precedente.

La primitiva restituisce il numero di byte ricevuti, e cioè quanti byte del messaggio è riuscita a copiare dal buffer di ricezione del socket. In caso di errore, restituisce -1.

La primitiva è bloccante: il processo si ferma finché non viene ricevuto almeno un byte del messaggio

Vediamo un esempio di codice del server…

#include …

int main () {

int ret, sd, len, addrlen;

char buf[BUFSIZE];

struct sockaddr\_in **sv\_addr**, cl\_addr;

sd = socket(AF\_INET, **SOCK\_DGRAM**, 0);

memset(&sv\_addr, 0, sizeof(sv\_addr)

sv\_addr.sin\_family = AF\_INET;

sv\_addr.sin\_addr.s\_addr = INADDR\_ANY;

sv\_addr.sin\_port = htons(20000);

ret = bind(sd, (struct sockaddr\*)&sv\_addr, sizeof(sv\_addr));

while(1) {

addrlen = sizeof(cl\_addr);

len = **recvfrom**(sd, buf, BUFSIZE, 0,

(struct sockaddr\*)&cl\_addr, &addrlen);

… // elaborazioni varie

}

return 0;

}

Dichiaro varie variabili, tra cui **cl\_addr,** in cui salverò l’indirizzo del socket UDP usato dal client per mandargli un messaggio.

Fatto ciò, creo il socket UDP del server, passando ***SOCK\_DGRAM***, come campo *type*.

Una volta assegnato un indirizzo al socket, posso già aspettare di ricevere messaggi dal client su questo socket.

… e codice del client:

#include …

int main () {

int ret, sd, len;

char buf[BUFSIZE];

struct sockaddr\_in **sv\_addr**;

sd = socket(AF\_INET, **SOCK\_DGRAM**, 0);

memset(&sv\_addr, 0, sizeof(sv\_addr)

sv\_addr.sin\_family = AF\_INET;

sv\_addr.sin\_addr.s\_addr = INADDR\_ANY;

sv\_addr.sin\_port = htons(20000);

while(1) {

len = **sendto**(sd, buf, BUFSIZE, 0,

(struct sockaddr\*)&sv\_addr, sizeof(sv\_addr));

… // elaborazioni varie

}

return 0;

}

Dichiaro varie variabili, tra cui **sv\_addr,** in cui salvo l’indirizzo del socket UDP del server a cui manderò un messaggio.

Fatto ciò, creo il socket UDP del client, passando ***SOCK\_DGRAM***, come campo *type*.

A questo punto, posso già iniziare a usare questo socket per mandare messaggi a quello del server.

**SOCKET UDP “CONNESSO”**

UDP non richiede che venga prima instaurata una connessione tra i due processi che vogliono comunicare, dunque è necessario che questi:

* Nella *sendto()*, specifichino ogni volta l’indirizzo del socket UDP destinatario del messaggio(visto che potrebbero mandarlo a chiunque);
* Nella *recvfrom()*, specifichino in quale variabile salvare l’indirizzo del socket da cui hanno ricevuto un messaggio (visto che potrebbero riceverlo da chiunque).

Il client, tuttavia, userà il proprio socket UDP solo per scambiare messaggi col server. Quello che può fare allora è chiamare la ***connect()*** sul proprio socket UDP, passando come campo *addr* un puntatore ad una struttura di tipo *sockaddr\_in* contenente l’indirizzo del socket UDP del server. In questo modo, il socket del client invierà/riceverà messaggi solo al/dal socket del server, dunque è inutile che specifichi/si salvi l’indirizzo del socket remoto, essendo quello del socket UDP del server. Infatti, una volta chiamata la *connect()*, è possibile per il client usare le primitive *send()* e *recv()* (che, per l’appunto, non richiedono che venga specificato/salvato l’indirizzo del socket remoto).

**NB:** usando la *connect()*su un socket UDP non viene stabilita alcuna connessione tra il socket UDP locale e quello remoto! Il protocollo di trasporto è sempre infatti UDP, che è *connectionless*!

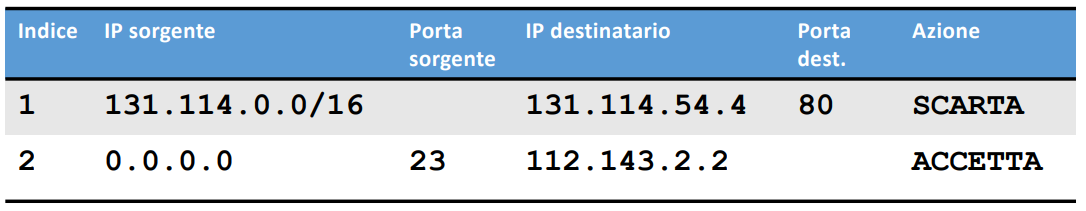
**FIREWALL** *12-12-22*

Un **firewall** è un software che permette di proteggere una sottorete dal resto di Internet, facendo sì che solo alcuni pacchetti siano in grado di entrare/uscire dalla sottorete, bloccando invece gli altri. La teoria l’abbiamo già vista con Peppe Anastasio; noi vedremo più in dettaglio i firewall di tipo ***stateless packet filters*** e come configurare quello presente su UNIX.

**STATELESS PACKET FILTERS**

Un firewall di tipo **stateless packet filter**, senza tenere conto di quali sono le connessioni TCP attive o gli scambi UDP in corso, opera sul singolo pacchetto che gli arriva, analizzando il suo header IP e TCP/UDP.  
In particolare, sfrutta una propria **tabella di regole**, dove ogni regola è costituita da:

* Alcuni ***criteri***;
* L’***azione*** da intraprendere quando il firewall riceve un pacchetto che soddisfa tali criteri:
  + *ACCEPT*: far passare il pacchetto;
  + *DROP*: bloccare il pacchetto.

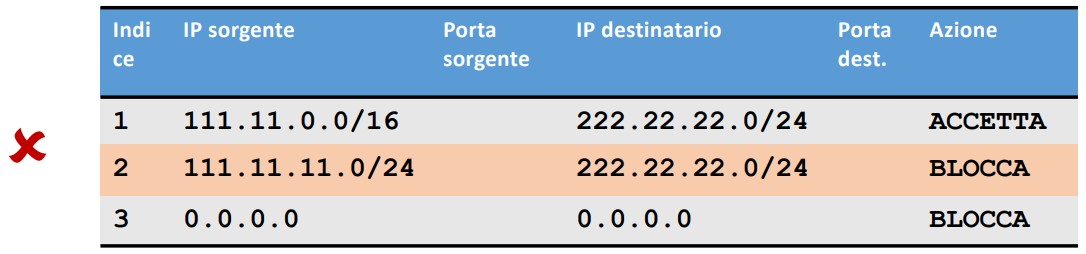


Quando un pacchetto arriva al firewall, questo inizia a consultare la propria tabella di regole in ordine crescente di indice (quindi dalla prima fino all’ultima), e non appena trova un’entrata i cui criteri corrispondono a quelli del pacchetto, applica l’azione relativa a tale entrata, facendo così passare o meno tale pacchetto. Dopodiché, il firewall passa ad analizzare altri pacchetti.

L’ordine delle regole è dunque importante. Supponiamo infatti di avere una rete locale di indirizzo 22.22.0.0/16 e di voler configurare un firewall di tipo stateless packet filter per:

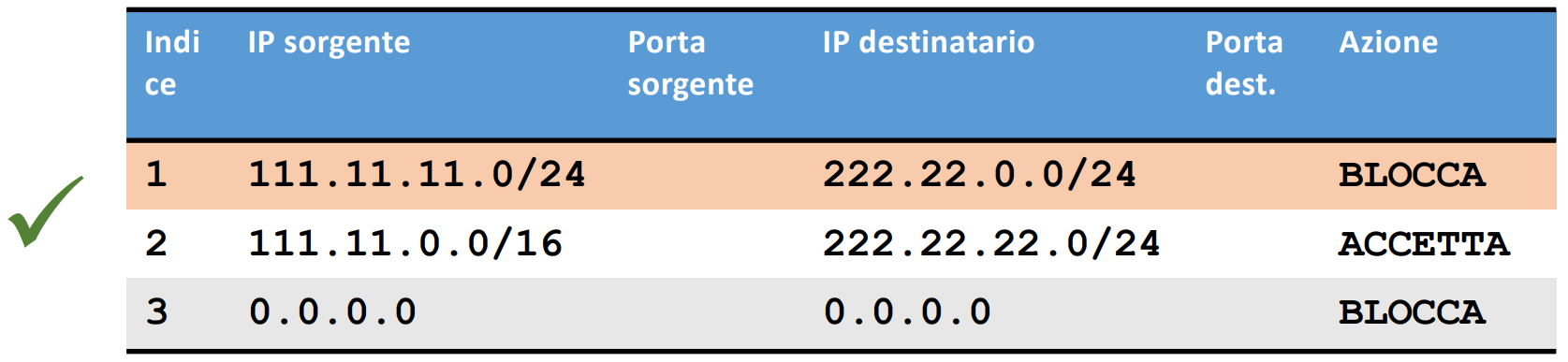
* Impedire l'accesso a Internet dall'interno della rete;
* Consentire alla sottorete esterna 111.11.0.0/16 di accedere alla sottorete locale 222.22.22.0/24;
* Impedire alla sottorete esterna 111.11.11.0/24 di accedere alla sottorete locale 222.22.22.0/24.

Questo qui è un ordine **errato**:



Infatti, se arrivasse un pacchetto proveniente dalla sottorete esterna 111.11.11.0/24 diretto alla sottorete locale 222.22.22.0/24, questo erroneamente verrebbe fatto passare, perché il firewall vedrebbe soddisfatta e applicherebbe la prima regola. Il problema infatti è che 111.11.0.0/16 contiene anche 111.11.11.0/24!

La soluzione quindi è che, se ci sono due (o più) regole dove i criteri di una rientrano nei criteri dell’altra, bisogna prima mettere quella più particolare, e poi quella più generale. Questo è dunque l’ordine **corretto:**

****

**STATELESS PACKET FILTER DI UNIX**

Unix ha a disposizione uno stateless packet filter che sfrutta una tabella di regole chiamata **filter.** Questa si divide in tre sezioni, chiamate **chain**:

* **INPUT:** contiene le regole usate per filtrare pacchetti in ingresso destinati a processi locali;
* **OUTPUT:** contiene le regole usate per filtrare pacchetti in uscita da processi locali;
* **FORWARD:** contiene le regole usate per filtrare pacchetti in transito.

Ogni chain ha poi la sua **policy,** che definisce ciò che viene fatto nel caso in cui i criteri di un pacchetto ricevuto dal firewall non corrispondessero a quelli di nessuna regola presente nella chain che lo riguarda.

Si avrà dunque:

* **Policy ACCEPT**: il pacchetto viene fatto passare. Si dice in questo caso che il firewall è **esclusivo** (si “esclude” dal problema, non fregandosene niente, e quindi lo fa passare). E’ un approccio più comodo, ma meno sicuro, perché se ci si è dimenticati di mettere una regola per bloccare certi pacchetti, questi potrebbero passare per via della policy.
* **Policy DROP**: il pacchetto viene buttato. Si dice in questo caso che il firewall è **inclusivo** (si “include” nel problema, preoccupandosi, e quindi non lo fa passare). E’ un approccio più sicuro ma meno comodo, perché se ci si è dimenticati di mettere una regola per far passare certi pacchetti, questi potrebbero non passare per via della policy.

Solo l’utente root può visualizzare e modificare la tabella *filter*, usando il programma **iptables**. Vediamo alcuni comandi utili:

List

* **iptables [-t *table*] -L [*chain*]:** visualizza le regole presenti nella *chain* indicata della *table* indicata.

Append

* **iptables [-t *table*] -A *chain* *rule-specification*:** aggiunge la regola che ha la *rule-specification* indicata in fondo alla *chain* indicata della *table* indicata;

Insert

* **iptables [-t *table*] -I *chain* [*num*] *rule-specification*:** aggiunge la regola che ha la *rule-specification* indicata alla posizione di indice *num* nella *chain* indicata della *table* indicata. Se non viene indicato alcun *num,* viene assunto *num == 1* e dunque la regola viene aggiunta all’inizio;

Delete

* **iptables [-t *table*] -D *chain* *rule-specification*:** rimuove la regola che ha la *rule-specification* indicata dalla *chain* indicata della *table* indicata;

Delete

* **iptables [-t *table*] -D *chain* *num*:** rimuove la regola nella posizione di indice *num* dalla *chain* indicata della *table* indicata.

Flush

* **iptables [-t *table*] -F [*chain*]:** rimuove tutte le regole dalla *chain* indicata della *table* indicata.

Policy

* **iptables [-t *table*] -P [*chain*] *target*:** imposta la policy *target* per la *chain* indicata della *table* indicata.

Laddove il parametro *table* è opzionale, se non viene specificato, il comando riguarderà la tabella *filter*; laddove il parametro *chain* è opzionale, se non viene specificato, il comando riguarderà tutte le chain della tabella scelta.

Le regole impostate usando il comando iptables (qualunque sia la chain della tabella su cui si opera) sono subito operative. L’unico problema è che, riavviando la macchina, vengono perse. E’ necessario dunque, una volta impostate, salvarle su un *file*, in questo modo:

**iptables-save > *file***

e poi, ogni volta che si riavvia la macchina, caricarle dal *file*, in questo modo:

**iptables-restore < *file***

*Esempi:*

iptables -A OUTPUT -p tcp -d 10.0.5.4 --dport 80 -j DROP: aggiunge in fondo alla chain OUTPUT della tabella *filter* (non si è specificata la *table*, quindi viene scelta questa) una regola che scarti tutti i pacchetti TCP in uscita da processi locali e destinati alla porta 80 (HTTP) dell’host 10.0.5.4;

iptables -A INPUT -p udp -s 121.0.0.0/16 -j ACCEPT: aggiunge in fondo alla chain INPUT della tabella *filter* (non si è specificata la *table*, quindi viene scelta questa) una regola che lasci passare tutti i pacchetti UDP in ingresso destinati a processi locali e provenienti dalla sottorete 121.0.0.0/16;

iptables -A INPUT -p icmp -i eth0 -j DROP: aggiunge in fondo alla catena INPUT della tabella *filter* (non si è specificata la *table*, quindi viene scelta questa) una regola che scarti tutti i pacchetti ICMP in ingresso destinati a processi locali e arrivati sull’interfaccia d’ingresso eth0.

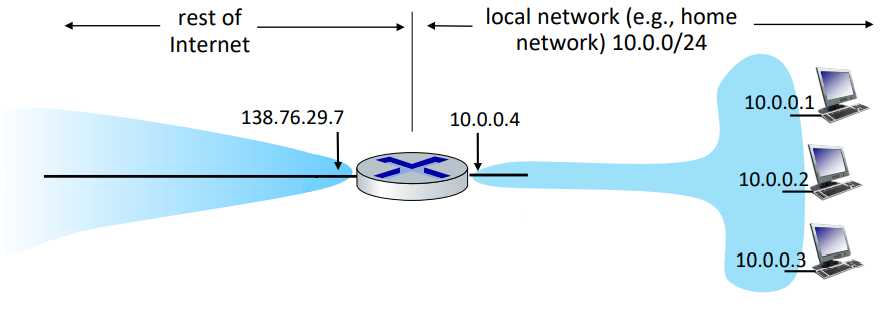
Visualizzando ora la tabella *filter*, avremo questo:

Immagine che contiene testo

Descrizione generata automaticamente

**NAT E PAT** *13-12-22*

Lo spazio di indirizzamento IPv4 è a soli 32 bit, e questo ha portato presto ad una scarsità di indirizzi IP.   
Un modo per gestirla è attraverso il **NAT**: data una sottorete in cui è presente come router di frontiera un router NAT, L’ISP assegna un indirizzo IP solo a tale router, detto **indirizzo** **IP pubblico**, unico al mondo. Tutti i nodi connessi alla sottorete avranno invece degli **indirizzi IP privati**, attraverso cui sono identificati univocamente all’interno della sottorete (anche il router quindi avrà un indirizzo IP privato, essendo connesso alla sottorete).



Supponiamo ora che un client che gira su un host che si trova in una sottorete nattata (e cioè una sottorete in cui c’è un router NAT) voglia comunicare con un server che gira su un altro host che si trova in un’altra sottorete. La comunicazione avviene in questo modo:

1. L’host client prepara un pacchetto in cui mette, tra le varie cose:
   1. come indirizzo IP sorgente, il proprio indirizzo IP privato;
   2. come numero di porta sorgente, il numero di porta del client.
2. Il router NAT, ricevuto il pacchetto, sostituisce l’indirizzo IP sorgente con il proprio indirizzo IP pubblico. Fatto ciò, inoltra il pacchetto verso l’esterno, che arriverà così al server;
3. Il server, ricevuto il pacchetto, risponde con un altro pacchetto in cui mette, tra le varie cose:
   1. come indirizzo IP destinatario, quello che nel pacchetto ricevuto prima era l’indirizzo IP sorgente (pensando sia l’indirizzo IP dell’host su cui gira il processo che l’aveva contattato), e quindi l’indirizzo IP pubblico del router NAT;
   2. come numero di porta destinataria, quello che nel pacchetto ricevuto prima era il numero di porta sorgente (pensando sia il numero di porta del processo che l’aveva contattato, che lo è effettivamente), e quindi il numero di porta del client.
4. Il router NAT, ricevuto il pacchetto, sostituisce l’indirizzo IP destinatario (che è il proprio indirizzo IP) con l’indirizzo IP privato dell’host client. Fatto ciò, inoltra il pacchetto nella sottorete, che arriverà così al client.

C’è un problema nel punto 4: se nella sottorete nattata ci fossero più host, come fa il router NAT a capire quale indirizzo IP privato mettere al posto dell’indirizzo IP destinatario scritto nel pacchetto (e cioè, come fa a capire a chi effettivamente è destinato il pacchetto)?

Se i processi che girano su questi host avessero tutti numeri di porta diversi, il router NAT potrebbe usare il numero di porta destinataria scritta nel pacchetto per capire anche qual è l’host cui tale pacchetto è destinato, sostituendo così l’indirizzo IP destinatario con l’indirizzo IP privato di tale host. Tuttavia, se per esempio il numero di porta destinatario scritto nel pacchetto fosse ‘69’ e nella sottorete ci fossero due host su cui gira un processo che ha tale numero di porta, il router NAT non saprebbe dire a quale dei due host tale pacchetto è destinato.

La soluzione allora è estendere il NAT con il **PAT.** Per capire cos’è, consideriamo nuovamente l’esempio di prima, vedendo come avviene la comunicazione nel caso in cui l’host client si trovi in una sottorete naptata (e cioè una sottorete in cui c’è un router NAPT):

1. L’host client prepara un pacchetto in cui mette, tra le varie cose:
   1. come indirizzo IP sorgente, il proprio indirizzo IP privato;
   2. come numero di porta sorgente, il numero di porta del client.
2. Il router NAPT, ricevuto il pacchetto, sostituisce:
   1. l’indirizzo IP sorgente con il proprio indirizzo IP pubblico;
   2. Il numero di porta sorgente con un altro numero di porta, detto ***numero di porta esterna***. Questo è un numero diverso per ogni coppia {IP sorgente – porta sorgente}, così che possa essere usato per identificare univocamente un processo (nel nostro caso, il client) tra tutti i processi che girano su tutti gli host della sottorete;

Fatto ciò, inoltra il pacchetto verso l’esterno, che arriverà così al server;

1. Il server, ricevuto il pacchetto, risponde con un altro pacchetto in cui mette, tra le varie cose:
   1. come indirizzo IP destinatario, quello che nel pacchetto ricevuto prima era l’indirizzo IP sorgente (pensando sia l’indirizzo IP dell’host su cui gira il processo che l’aveva contattato), e quindi l’indirizzo IP pubblico del router NAT;
   2. come numero di porta destinataria, quello che nel pacchetto ricevuto prima era il numero di porta sorgente (pensando sia il numero di porta del processo che l’aveva contattato), e quindi il numero di porta esterna scelto per il client dal router NAPT;
2. Il router NAPT, ricevuto il pacchetto, guarda il numero di porta esterna e capisce esattamente verso quale processo (e quindi verso quale host) è destinato, e cioè verso il client (che gira sull’host client). Sostituisce allora:
   1. l’indirizzo IP destinatario (che è il proprio indirizzo IP) con l’indirizzo IP privato dell’host client.
   2. Il numero di porta destinataria (che è il numero di porta esterna scelto prima) con il numero di porta del client.

Fatto ciò, inoltra il pacchetto nella sottorete, che arriverà così al client.

**NAPT DI UNIX**

Unix ha a disposizione un NAPT che sfrutta una tabella di regole chiamata **nat.** Questa si divide in tre chain:

* **PREROUTING:** contiene delle regole usate per cambiare indirizzo IP/numero di porta destinataria (operazione detta *DNAT*) dei pacchetti in ingresso;
* **OUTPUT:** contiene delle regole usate per cambiare indirizzo IP/numero di porta destinataria dei pacchetti in uscita dai processi locali, prima del routing (non vediamo in dettaglio);
* **POSTROUTING:** contiene delle regole usate per cambiare indirizzo IP/numero di porta sorgente (operazione detta *SNAT*) dei pacchetti in uscita;

Solo l’utente root può visualizzare e modificare la tabella *nat*, usando il programma **iptables** (e i comandi sono simili a quelli visti prima).

*Esempi:*

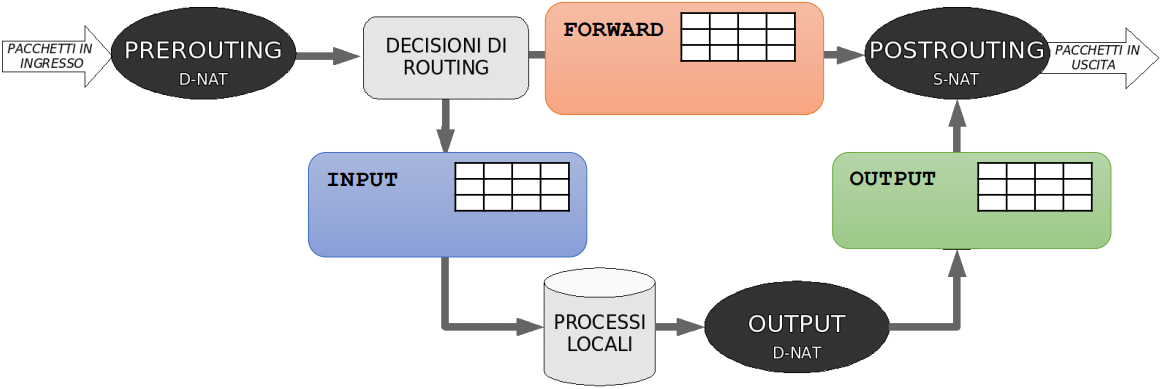
iptables -t nat -A POSTROUTING -s 192.168.0.2 -j SNAT --to-source 151.162.50.1: aggiunge in fondo alla chain POSTROUTING della tabella *nat* una regola che fa SNAT per tutti i pacchetti in uscita provenienti dall’host 192.168.0.2, impostando l’IP sorgente a 151.162.50.1;

iptables -t nat -A POSTROUTING -s 192.168.0.0/24 -j SNAT --to-source 151.162.50.1:4001-4100: aggiunge in fondo alla chain POSTROUTING della tabella *nat* una regola che fa SNAT per tutti i pacchetti in uscita provenienti dalla sottorete 192.168.0.0/24, impostando l’IP sorgente a 151.162.50.1 e la porta sorgente ad un numero appartenente all’insieme {4001,…,4100}.

iptables -t nat -A PREROUTING -d 151.162.50.2 -j DNAT --to 192.168.0.2: aggiunge in fondo alla chain PREROUTING della tabella *nat* una regola che fa DNAT per tutti i pacchetti in ingresso destinati a 151.162.50.2, impostando l’IP destinatario a 192.168.0.2;

iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to 192.168.0.55:80: aggiunge in fondo alla chain PREROUTING della tabella *nat* una regola che fa DNAT per tutti i pacchetti TCP in ingresso destinati alla porta 80, impostando l’IP destinatario a 192.168.0.55 e la porta destinataria a 80.

Questo è l’ordine con cui vengono applicate le varie chain delle tabelle *filter* e *nat*:



Quel *decisioni di routing* è un modulo presente nei router che viene usato per decidere qual è il prossimo nodo cui il pacchetto ricevuto va inoltrato (chiaramente negli host non sarà presente, visto che non si occupano di inoltrare pacchetti, così come non verrà mai applicata la chain *forward*). Le chain PREROUTING e POSTROUTING prendono dunque il nome dal fatto che vengono applicate rispettivamente prima e dopo il routing.

Si può poi notare che le chain della tabella *filter* vengono applicate dopo la chain PREROUTING e prima della chain POSTROUTING. In questo modo, il firewall opera vedendo indirizzi IP e numeri di porta “reali”, e non quelli modificati dal NAPT.

**APACHE** *14-12-22*

**Apache** è un server web HTTP. Tutti i comandi che vedremo possono essere usati solo dall’utente root.   
Il primo è il seguente:

control

**apache2ctl *argomento***

Dove ***argomento***può essere:

* **start**: avvia Apache;
* **stop**: arresta Apache;
* **restart**: riavvia Apache;
* **status**: mostra informazioni sullo stato attuale di Apache;
* **configtest:** controlla il file di configurazione di Apache per vedere se ci sono errori di sintassi.

In alternativa si può usare questo comando:

**service apache2 *argomento***

Dove ***argomento*** può essere:

* **start**: avvia Apache;
* **stop**: arresta Apache;
* **restart**: riavvia Apache;
* **reload**: ricarica la configurazione impostata sul file di configurazione di Apache. Infatti, questa configurazione viene caricata solo una volta all’avvio di Apache, dunque se poi si facessero modifiche, l’unico modo per renderle effettive è ricaricare la configurazione, riavviando Apache oppure usando questo comando.

Mentre Apache è in esecuzione, aprendo un browser e accedendo al sito <http://localhost/> verrà mostrata la pagina web */var/www/html/index.html*.

**CONFIGURAZIONE**

La directory principale dei file di configurazione di Apache è */etc/apache2,* e il file di configurazione principale è */etc/apache2/apache2.conf*. Questo, a sua volta, recupera parti di configurazione da altri file, attraverso la direttiva ***Include***.

Alcuni file contenenti parti di configurazione si trovano in */etc/apache2/conf-available.* Questi possono essere abilitati o disabilitati usando rispettivamente i comandi:

enable conf

**a2enconf *nome\_file***

disable conf

**a2disconf *nome\_file***

Per ogni file abilitato presente in */etc/apache2/conf-available* viene poi creato un soft link (un riferimento ad esso) in */etc/apache2/conf-enabled*, e il file di configurazione principale è impostato per includere tutti i file presenti in questa directory.

Esistono poi altri file, detti *moduli*, contenenti parti di configurazione più complesse. Questi si trovano in */etc/apache2/mods-available* e possono essere abilitati o disabilitati usando rispettivamente i comandi:

enable mod

**a2enmod *nome\_file***

disable mod

**a2dismod *nome\_file***

Per ogni file abilitato presente in */etc/apache2/mods-available* viene poi creato un soft link (un riferimento ad esso) in */etc/apache2/mods-enabled*, e il file di configurazione principale è impostato per includere tutti i file presenti in questa directory.

**DIRETTIVE GLOBALI**

Le **direttive globali** specificano opzioni valide per l’intero server Apache. Sono tipicamente presenti nel file di configurazione principale */etc/apache2/apache2.conf*, e le più importanti sono:

* **ServerRoot *path*:** specifica la directory principale dei file di configurazione di Apache. Su Debian, viene automaticamente impostata su */etc/apache2* non appena si avvia il server usando il comando *apache2ctl start* (per questo motivo, nel file di configurazione principale */etc/apache2/apache2.conf*, questa direttiva è commentata).
* **KeepAlive *on/off*:** specifica se Apache offrirà o meno connessioni TCP persistenti;
* **KeepAliveTimeout *seconds*:** nel caso in cui fossero offerte connessioni TCP persistenti, specifica per quanti secondi Apache attenderà la successiva richiesta dal client su una stessa connessione prima di chiuderla;
* **Listen *port*:** specifica la porta su cui Apache si metterà in ascolto di richieste di connessione.   
  E’ obbligatoria: se non è presente, il server non parte. Questa direttiva è presente nel file */etc/apache2/ports.conf*, e il file di configurazione principale */etc/apache2/apache2.conf* è impostato per includerlo;
* **ErrorLog *path*:** specifica il file in cui verrà salvato il log degli errori che si sono verificati durante l’esecuzione di Apache.

**DIRETTIVE PER I SITI WEB: VIRTUAL HOSTING**

Nell’approccio standard, si ha un server web in esecuzione su un host che ha un certo indirizzo IP e ospita un unico sito web. Un utente, in questo modo, accede al sito indicando semplicemente l’indirizzo IP dell’host.

Un approccio del genere chiaramente causerebbe un forte spreco di risorse, visto che servirebbe un host per ogni sito web. Per questo motivo, molti server web offrono un approccio basato sul **virtual hosting:** l’host su cui gira il server web può ospitare più siti web, e un client accede ad un certo sito indicando, oltre all’indirizzo IP dell’host, anche il nome simbolico del sito stesso.

Anche Apache offre un approccio basato sul virtual hosting. In particolare, i file di configurazione di tutti i siti web che dovrà gestire vanno messi nella directory */etc/apache2/sites-available,* e possono essere abilitati o disabilitati (rendendo così i corrispondenti siti raggiungibili o meno dall’esterno) usando rispettivamente i comandi:

enable site

**a2ensite *nome\_file***

disable site

**a2dissite *nome\_file***

Per ogni file abilitato presente in */etc/apache2/sites-available* viene poi creato un soft link (un riferimento ad esso) in */etc/apache2/sites-enabled*, e il file di configurazione principale è impostato per includere tutti i file presenti in questa directory.

Il file di configurazione di un sito web ha il seguente formato:

Immagine che contiene testo

Descrizione generata automaticamente

Qui sono presenti varie **direttive per i siti web,** e cioè direttive che specificano opzioni valide solo per il particolare sito web. Vediamone alcune:

* **VirtualHost *IP*:*port* :** è una cosiddetta *direttiva contenitore*, e cioè contiene al suo interno più direttive che riguardano il particolare sito web. Come valore, va messo l’indirizzo IP e il numero di porta destinataria che si vuole un client inserisca per accedere a tale sito web.
* **ServerName** ***name***: specifica il nome simbolico del sito web;
* **DocumentRoot** ***path***: specifica la directory dei file del sito web.

**NB:** se si abilitano/disabilitano file di configurazione mentre Apache è in esecuzione (siano essi presenti in */etc/apache2/conf-available, /etc/apache2/mods-available* o */etc/apache2/sites-available)*, occorre ricaricare la configurazione per rendere le modifiche effettive. Il file di configurazione principale infatti include tutti i soft link dei file abilitati, e quindi cambiando i file abilitati, cambiano anche i soft link che il file di configurazione principale include, e quindi cambia il contenuto del file di configurazione principale stesso. Visto però che la configurazione viene caricata una volta sola all’avvio di Apache, per rendere effettive le modifiche è necessario ricaricarla.

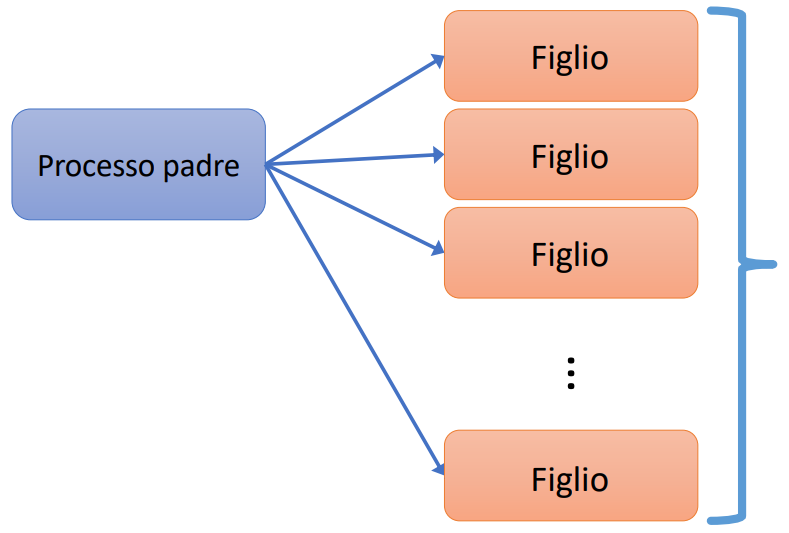
**MULTI-PROCESSING MODULES**

Apache presenta **tre multi-processing modules (MPM),** e cioè tre modi di gestire le richieste che gli arrivano da parte dei client. Vediamoli.

**MPM PREFORK**

All’avvio di Apache, il processo fa del *preforking*, creando in anticipo un certo numero di processi figli che si occuperanno di accettare e gestire nuove richieste di connessione.

Il processo padre, invece, si occuperà di creare o terminare figli, per vari motivi (*es.* se vede che al momento ci sono pochi figli disponibili a servire richieste da parte dei client, ne crea altri. Se poi i figli disponibili dovessero essere troppi, ne termina alcuni. Inoltre, dopo che un figlio ha servito un certo numero di connessioni, lo termina, così se si fossero verificati dei memory leak accidentali, vengono recuperate tutte le risorse consumate).



I principali vantaggi di gestire le richieste in questo modo sono i seguenti:

* **Overhead ridotto**: grazie al preforking, il grosso del lavoro viene fatto all’avvio, quando Apache ancora non sta gestendo richieste da parte di client. Se poi fosse necessario creare altri processi figli, in generale ne vengono creati molti di meno per volta di quanti ne sono stati creati all’inizio, dunque il tempo di attesa è limitato;
* **Stabilità**: se dovesse crashare un processo figlio, al massimo verrebbe interrotta la connessione che lui aveva instaurato con un client, dunque il danno è limitato.

I principali svantaggi, invece, sono i seguenti:

* **Grossa occupazione di memoria**: la RAM si riempirebbe di tutti questi cloni del processo padre;
* **Difficoltà nel regolare le impostazioni**: bisogna configurare il server in modo che non si abbiano mai né troppi figli (così da non sprecare risorse inutilmente), né troppo pochi (così da dover chiamare il meno possibile la *fork()* per crearne altri, limitando il tempo di attesa).

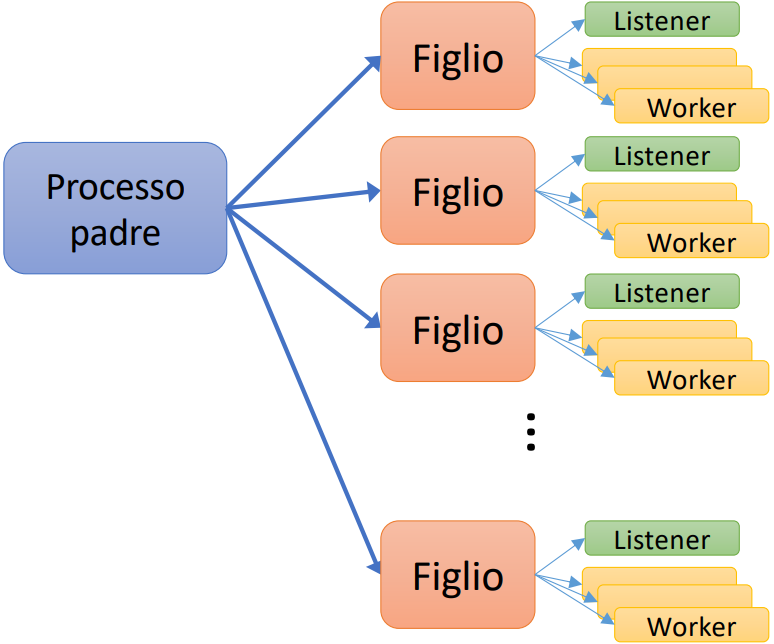
**MPM WORKER**

All’avvio di Apache, il processo fa del *preforking*, creando in anticipo un certo numero di processi figli.

Ognuno di questi, a sua volta, crea:

* Un thread *listener*, che si occuperà di accettare nuove richieste di connessione;
* Vari thread *worker*, che si occuperanno di gestire le richieste di connessione accettate dal listener.

Il processo padre, come nell’MPM precedente, si occuperà di creare o terminare figli, per vari motivi (*es.* se vede che al momento ci sono pochi thread disponibili a servire richieste da parte dei client, crea altri figli, che creeranno a loro volta i thread necessari. Se poi i thread disponibili dovessero essere troppi, termina alcuni figli, e così verranno terminati anche i rispettivi thread. Inoltre, dopo che un figlio – tramite i suoi thread –   
ha servito un certo numero di connessioni, lo termina, così se si fossero verificati dei memory leak accidentali, vengono recuperate tutte le risorse consumate).



I principali vantaggi di gestire le richieste in questo modo sono i seguenti:

* **Overhead ulteriormente ridotto**: adesso il processo padre può creare molti meno processi figli, visto che poi ogni processo figlio sfrutterà i thread per servire le richieste;
* **Risparmio di memoria:** essendo necessari molti meno processi figli, la RAM sarà più libera.

I principali svantaggi, invece, sono i seguenti:

* **Problemi di compatibilità:** alcuni moduli Apache non supportano i thread;
* **Stabilità ridotta:** i thread condividono tra loro parte della memoria, dunque se uno di questi dovesse fare qualcosa di sbagliato, l’errore potrebbe ripercuotersi anche su tutti gli altri;
* **Difficoltà nel regolare le impostazioni**: bisogna fare in modo che non si abbiano mai né troppi thread (così da non sprecare risorse inutilmente), né troppo pochi (così da dover chiamare il meno possibile la *fork()* per creare altri figli che creeranno i thread necessari, limitando il tempo di attesa).

**MPM EVENT**

E’ un’evoluzione del MPM worker, che presenta la seguenti novità:

1. Supponiamo che un worker W1 sia connesso ad un client C1 che tarda a inviare una richiesta. Invece di attendere, W1 restituisce il controllo del socket che usava per comunicare con C1 al listener, e passa a servire un altro client C2. Quando poi C1 invierà una richiesta, il listener assegnerà il controllo del socket usato per comunicare con C1 ad un altro worker libero W2.
2. Supponiamo che un worker W1 stia servendo un client C1 con una connessione lenta, a tal punto che il buffer di invio del socket usato per comunicare con C1 diventa pieno. Invece di attendere, W1 restituisce il controllo di tale socket al listener e passa a gestire un altro client. Non appena sarà di nuovo scrivibile, il listener assegnerà il controllo del socket usato per comunicare con C1 ad un altro worker libero W2.

**HIERARCHICAL ROUTING**

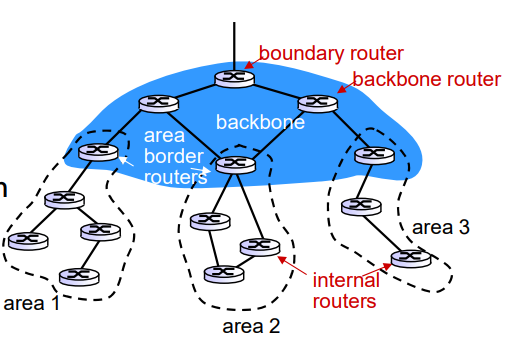
Si basa su una visione secondo la quale i router non sono tutti uguali: ci sono infatti più di 600 milioni di destinazioni su Internet, e non è possibile che le routing table di ogni router memorizzino tutte queste informazioni. Internet allora viene partizionata in *parti* più piccoli, così che ogni router memorizzi e pubblicizzi meno informazioni. Ogni *parte* prende il nome **autonomous systems** **(AS),** che è un insieme di router dove:

* i router interni useranno un certo stesso algoritmo di routing **intra-AS**;
* i router di frontiera useranno un certo stesso algoritmo di routing **inter-AS.**

**ALGORITMI INTRA-AS: OSPF**

Supponiamo che un router interno R1 di AS1 riceva un pacchetto destinato al di fuori di AS1. Dovrebbe prima mandarlo ad un router di frontiera di AS1, ma come fa a sapere qual è il percorso di router migliore per raggiungere un qualsiasi router di frontiera? A tale scopo, ogni router interno usa degli **algoritmi intra-AS**.

Il più famoso è **Open Shortest Path First (OSPF),** che sfrutta Dijkstra. In particolare, questo algoritmo vede l’AS divisa in tante aree (quelle propriamente chiamate aree + la backbone), e viene applicato Dijkstra su ognuna di queste aree (infatti, Dijkstra per essere applicato richiede la conoscenza della topologia di tutta la rete. Invece di applicarlo allora su tutta la AS, che può essere molto vasta, viene applicato su ognuna di queste aree, che sono di dimensione più limitata).



**ALGORITMI INTER-AS: BGP**

Grazie agli algoritmi intra-AS, il router R1 sa qual è il percorso migliore per mandare il pacchetto ad un qualsiasi router presente in AS1, e quindi ad un qualsiasi router di frontiera. Come fa adesso a sapere a quale tra questi router deve effettivamente mandare questo pacchetto, così che arrivi all’AS esterna desiderata seguendo il percorso migliore? A tale scopo, ogni router di frontiera usa degli algoritmi **inter-AS**.

Il più famoso è **Border Gateway Protocol (BGP)**, che è costituito da due parti:

* **eBGP**: usato da ogni router di frontiera di AS1 per sapere dagli AS vicini a quali altri AS portano;
* **iBGP**: usato da ogni router di frontiera di AS1 per propagare questa informazione a tutti i router interni di AS1, così che sappiano quali sono gli AS raggiungibili attraverso tale router di frontiera.

In particolare, ad ogni router interno, un router di frontiera dà queste informazioni:

* **AS-PATH**: considerato un certo AS esterno, indica quali sono gli AS che verranno attraversati per arrivare a tale AS esterno;
* **NEXT-HOP**: indica, per lo specifico router interno, qual è il prossimo router di AS1 cui lo specifico router deve mandare il pacchetto, così che questo pacchetto possa arrivare effettivamente al router di frontiera.

A questo punto, R1, oltre a sapere qual è il percorso migliore per raggiungere un qualsiasi router di frontiera, sa anche quali sono i router di frontiera che portano all’AS esterno desiderato.

Adesso, date le varie informazioni che ha ricevuto dai router di frontiera che portano all’AS esterno desiderato, R1 manderà il pacchetto verso il router di frontiera che gli ha mandato l’informazione che presenta l’AS-PATH con meno AS (e cioè, manderà il pacchetto verso il percorso che attraverserà il minor numero di AS esterni. Non è detto quindi che sia effettivamente il percorso migliore, visto che magari potrebbe esserci un percorso di costo minore che attraversa più AS esterni).