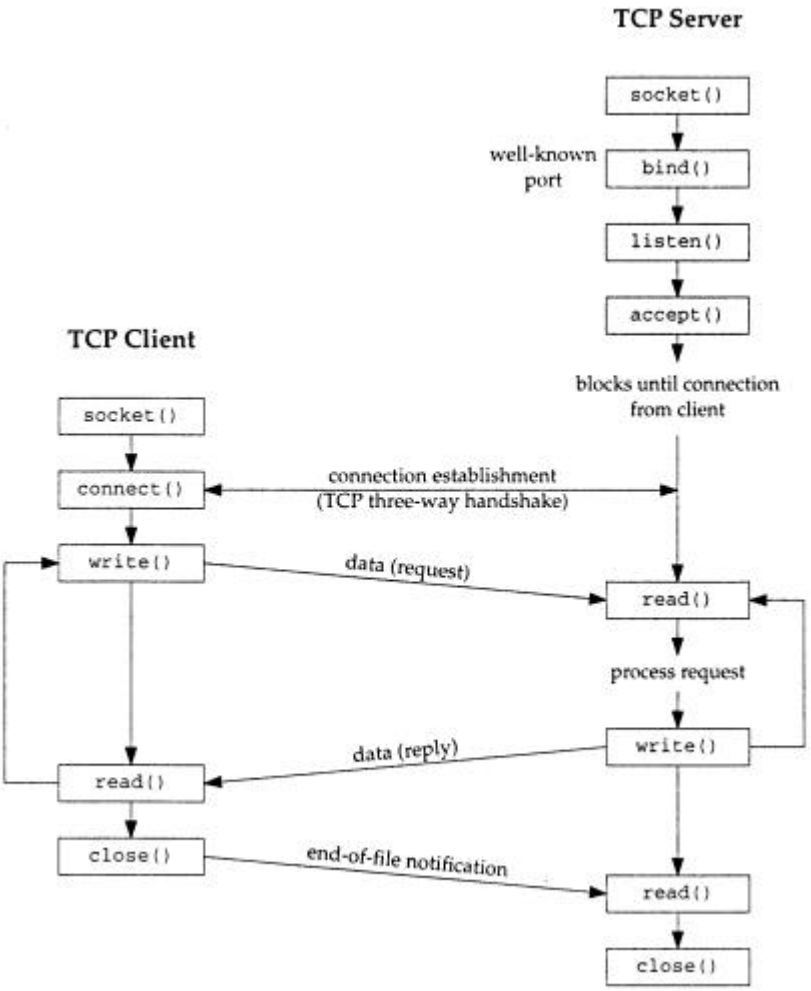
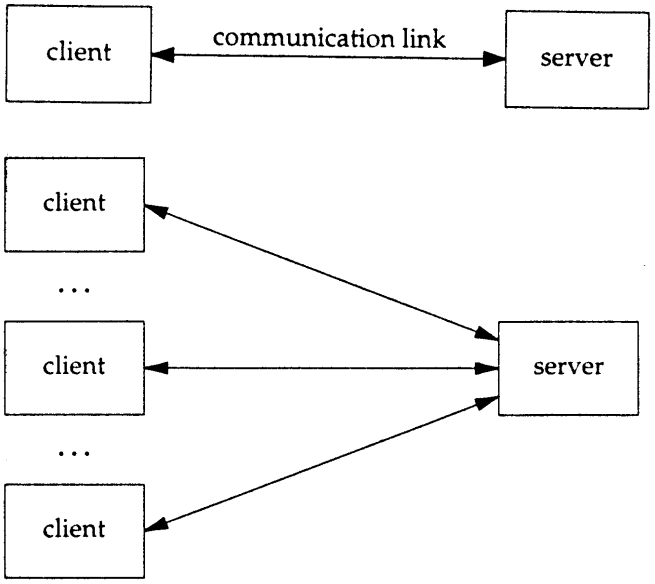
Socket per TCP: Fondamenti



Network Applications

Molte applicazioni di rete sono formate da due programmi distinti (che lavorano su due diversi host) uno detto server ed uno detto client. Il server si mette in attesa di una richiesta da servire, il client effettua tale richiesta.



Tipicamente il client comunica con un solo server, mentre un server usualmente comunica con più client contemporaneamente (su connessioni diverse nel caso tcp). Inoltre spesso client e server sono processi utente, mentre i protocolli della suite TCP/IP fanno solitamente parte del sistema operativo. Nel seguito faremo riferimento al termine IP nel senso di IPv4.

Unix Standards

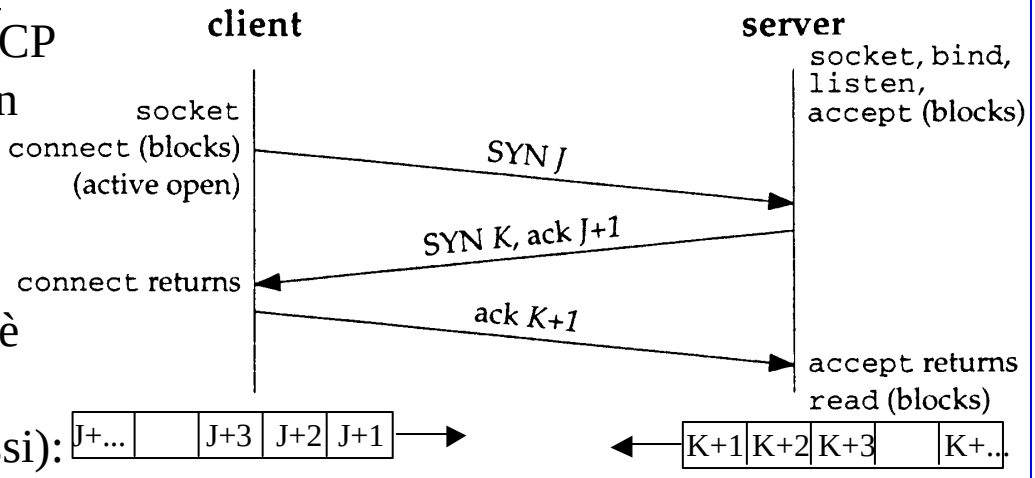
Posix = Portable Operating System Interface è una famiglia di standard (vedi http://www.pasc.org/standing/sd11.html) sviluppata da IEEE e adottata da ISO. Posix comprende **IEEE Std 1003.1 (1996)** (una raccolta di alcune specifiche precedenti) che contiene al suo interno una parte detta “Part1: System Application Program Interface (API)” che specifica l’interfaccia C per le chiamate di sistema del kernel Unix, relative a processi (fork, exec, signal, timer, user ID, gruppi), files e directory (I/O function), I/O da terminale, password, le estensioni per il realtime, execution scheduling, semaphores, shared memory, clock, message queues. In particolare comprende **IEEE Std 1003.1g: Protocol Independent Interface (PII)** che è lo standard per l’interfaccia di programmazione delle reti, e definisce due standard chiamati **DNI** (**Detailed Network Interfaces**):

1) **DNI/Socket** basato sulle API socket del 4.4BSD, di cui ci occuperemo

2) DNI/XTI, basato sulle specifiche XPG4 del consorzio X/Open

Setup delle ConnessioniTCP

Una connessione TCP viene instaurata con le seguenti fasi, che formano il **Three-Way Handshake** (perchè formato da almeno 3 pacchetti trasmessi):



1) il **server** si predispone ad **accettare una richiesta di connessione**, mediante le chiamate a socket, bind, listen e infine **accept** che realizza una apertura passiva (passive open) cioè senza trasmissione di dati.

2) il client effettua le chiamate a socket, bind ed infine alla **connect** che realizza una apertura attiva (active open) mediante la spedizione di un **segmento** TCP detto **SYN** segment (synchronize) in cui è **settato ad 1 il flag syn**, a zero il flag ack, e che trasporta un **numero di sequenza iniziale** (J) che è il numero di sequenza iniziale dei **dati che il client vuole mandare al server**. Il segmento contiene un header TCP con i numeri di porta ed eventuali opzioni su cui accordarsi, e di solito non contiene dati. Il segmento viene incapsulato in un datagram IP.

3) Il server deve rispondere al segmento SYN del client spedendogli **un segmento SYN** (**flag syn settato ad 1**) con il numero di sequenza iniziale (K) dei **dati che il server vuole mandare al client** in quella connessione. Il segmento presenta inoltre nel campo **Ack number il valore J+1** che indica che si aspetta di ricevere J+1, e presenta il **flag ack settato ad 1**, per validare il campo Ack number.

4) il client, ricevendo il SYN del server con l'Ack numer J+1 sa che la sua richiesta di connessione è stata accettata, e dal sequence number ricevuto K capisce che i dati del server inizieranno da K+1, quindi risponde con un segmento ACK (**flag syn settato a zero** e **flag ack settato a 1**) con **Ack number K+1**, e termina la connect.

5) al ricevimento dell'ACK K+1 il server termina la accept.

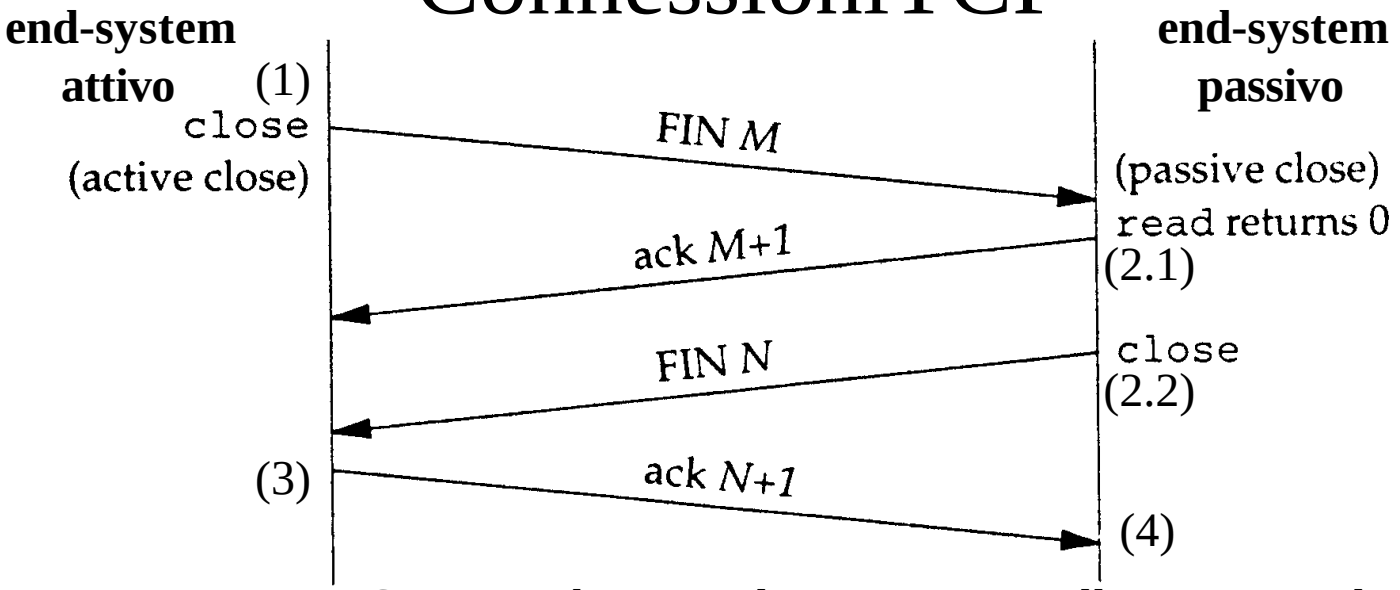
Opzioni TCP nel Setup

Ogni segmento di tipo SYN può contenere delle opzioni, che servono a stabilire alcune caratteristiche della connessione che si sta instaurando. Tra le più importanti ricordiamo:

1) **MSS options**: con questa opzione il TCP che manda il proprio SYN annuncia il maximum segment size, la più grande dimensione di segmento che è in grado di ricevere. L'opzione TCP_MAXSEG resa disponibile dall'interfaccia socket, consente di settare questa opzione.

2) **Windows scale options**: la finestra scorrevole più grande che due TCP possono concordare è 65535, perchè il campo Window Size occupa 16 bit. Per situazioni in cui il collegamento è a larghissima banda ma con grande ritardo di trasmissione (es. via satellite) una dimensione di finestra più grande rende più veloce la trasmissione di grandi quantità di dati. Per identificare finestre più grandi nell'header TCP, si setta questa opzione che indica di considerare il campo Window Size dopo averlo shiftato a sinistra di un numero di posizione compreso tra 0 e 14, in modo da moltiplicare la Window Size di un fattore fino a 2 elevato alla 14, ovvero in modo da raggiungere valori dell'ordine del GigaByte. L'opzione SO_RECVBUF resa disponibile dall'interfaccia socket, consente di settare questa opzione.

Terminazione delle ConnessioniTCP (1)



Una connessione TCP viene chiusa mediante un protocollo composto da **quattro** messaggi trasmessi:

1) una delle applicazioni su un end-system (chiamiamola **attiva**) effettua la chiusura attiva (active close) chiamando la funzione **close()** che **spedisce un segmento FIN** (flag FIN settato a 1), con un numero di sequenza M pari all'ultimo dei byte trasmessi in precedenza più uno. Con ciò si indica che viene trasmesso un ulteriore dato, che è il FIN stesso. Per convenzione il FIN è pensato avere dimensione pari ad 1 byte, quindi l'end-system attivo si aspetta di ricevere per il FIN un ACK con Ack Number pari a M+1.

2) l'end system che riceve il FIN (chiamiamolo **passivo**) effettua la chiusura passiva (passive close) all'insaputa dell'applicazione.

2.1) Per prima cosa il modulo TCP del **passivo spedisce all'end-system attivo un segmento ACK** con Ack number pari a M+1, come riscontro per il FIN ricevuto.

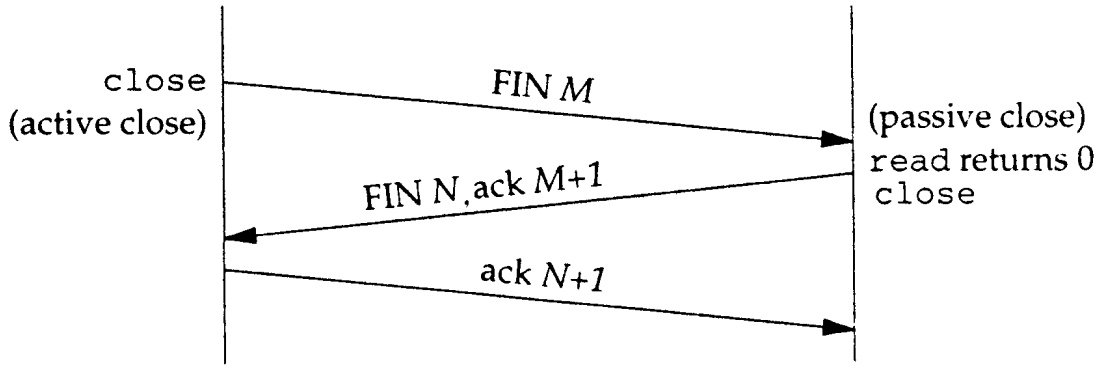
2.2) Poi il TCP passivo trasmette **all'applicazione** padrona di quella connessione il segnale FIN, sotto forma di **end-of-file** che viene accodato ai dati non ancora letti dall'applicazione. Poiche la ricezione del FIN significa che non si riceverà nessun altro dato, con l'end-of-file il TCP comunica all'applicazione che lo stream di input è chiuso.

Terminazione delle ConnessioniTCP (2)

3) Quando **l'applicazione del passivo finalmente legge dal buffer l'end-of-file** (con una read() che restituisce 0), deve effettuare per quel socket la chiamata alla funzione **close().** La close() ordina al modulo TCP di **inviare** a sua volta **all'end-system attivo un segmento FIN**, col numero di sequenza (N) del FIN, cioè l'ultimo byte trasmesso più 1.
4) il modulo TCP dell'attivo, quando riceve il FIN spedisce un ACK con Ack number N+1, cioè il numero di sequenza del successivo al FIN, cioè il FIN più uno, poichè 1 è per convenzione la dimensione del FIN. Terminato questo passo viene conclusa anche la funzione close() dell'attivo.

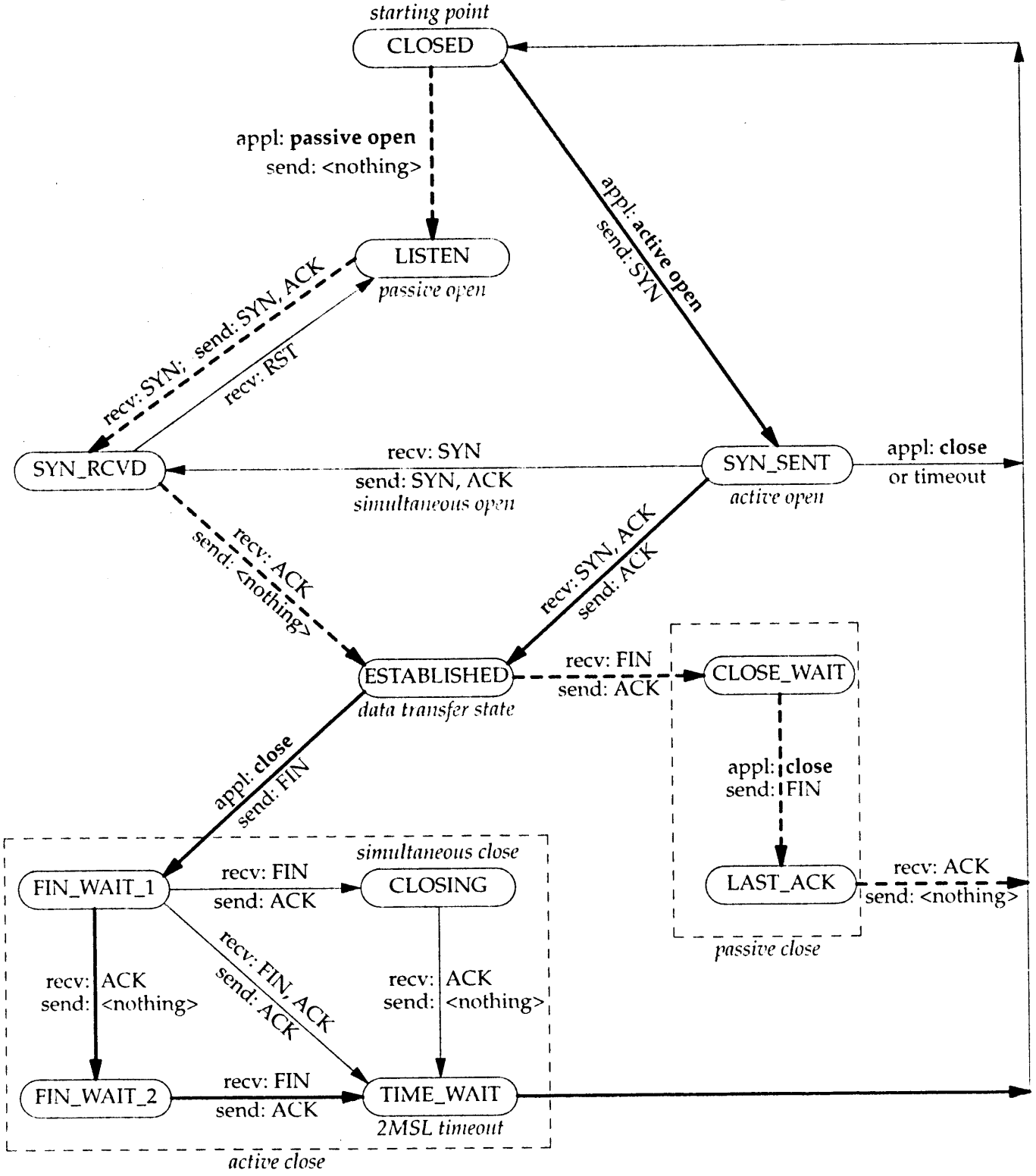
• Chiusura attiva o passiva non dipendono dall'essere client o server, ma solo da chi per primo effettua la chiamata alla funzione close().

• Notare che i due segmenti dal passivo all'attivo degli step 2.1 e 2.2 (ACK M+1 e FIN N rispettivamente) potrebbero essere combinati in un solo messaggio a seconda del comportamento del passivo.



• Un'ulteriore variazione, di carattere opposta alla precedente, è che tra gli step 2.1 e 2.2 **il passivo** ha ancora la possibilità di inviare dei dati verso l'attivo, perchè al momento dello step 2.1 è stata effettuata, mediante il FIN, una chiusura del flusso solo nella direzione dall'attivo al passivo, chiusura che viene detta half-close.

TCP State Transition Diagram (1)



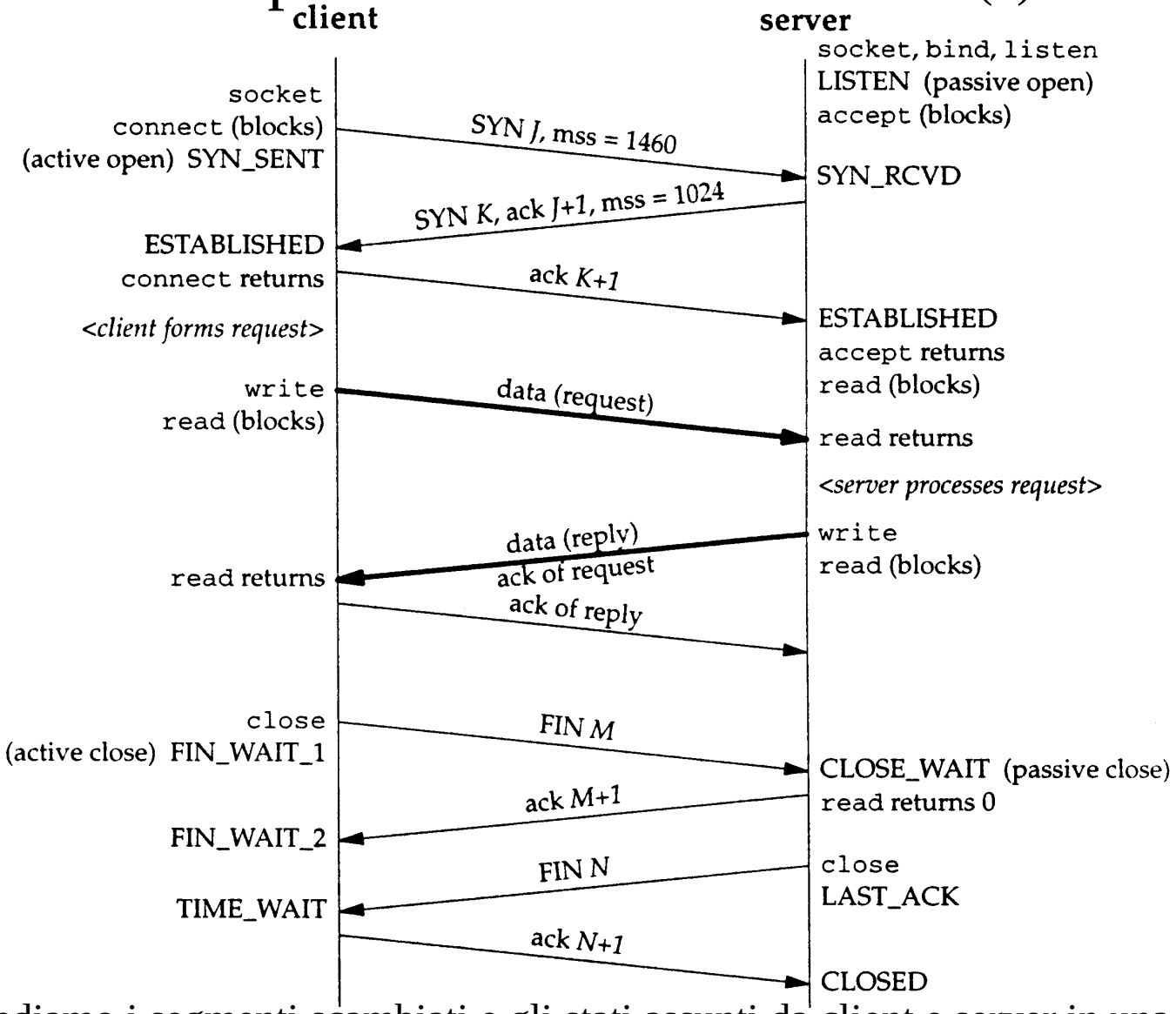
indicate normal transitions for client
indicate normal transitions for server
appl: *indicate state transitions taken when application issues operation*
recv: *indicate state transitions taken when segment received*
send: *indicate what is sent for this transition*

TCP State Transition Diagram (2)

Le operazioni di instaurazione e terminazione di una connessione TCP sono specificate dal precedente diagramma di transizione degli stati. Da ogni stato una o più frecce uscenti individuano una condizione che fa uscire da quello stato e passare in un altro stato. Tale condizione può essere la ricezione di un segmento, un'operazione effettuata dall'applicazione proprietaria del socket oppure lo scadere di un timeout.
In corrispondenza di una transizione di stato può essere effettuata un'operazione come ad es. la trasmissione di un segmento.
Ad es. se ci troviamo nello stato ESTABLISHED, in cui una connessione è stata instaurata tra i due end system, se l'applicazione effettua una chiusura attiva chiamando la close(), il TCP spedisce un segmento di FIN e passa nello stato FIN_WAIT aspettando una risposta. Se invece sempre dallo stato ESTABLISHED il TCP riceve un segmento FIN dall'altro end-system allora parte con la chiusura passiva, risponde con un ACK e si porta nello stato CLOSE_WAIT.

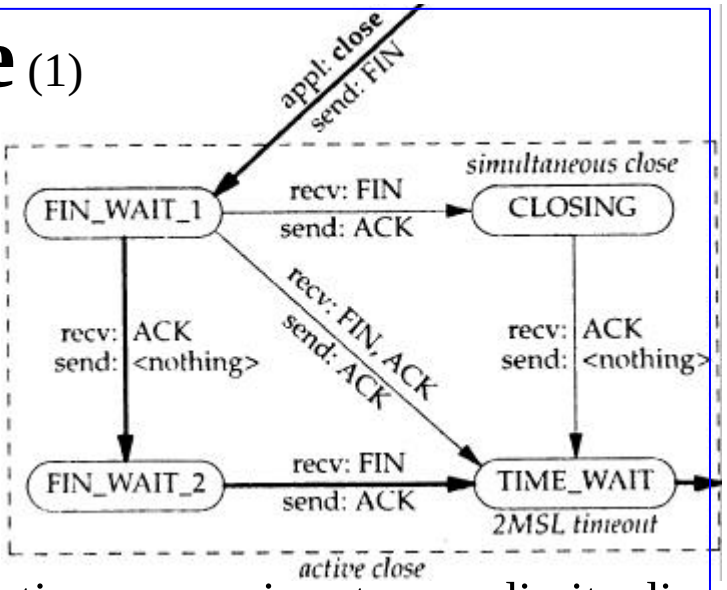
Gli stati qui illustrati sono utilizzati dall'applicazione netstat per identificare la situazione corrente di ciascun socket TCP.

Esempio di connessione TCP (1)



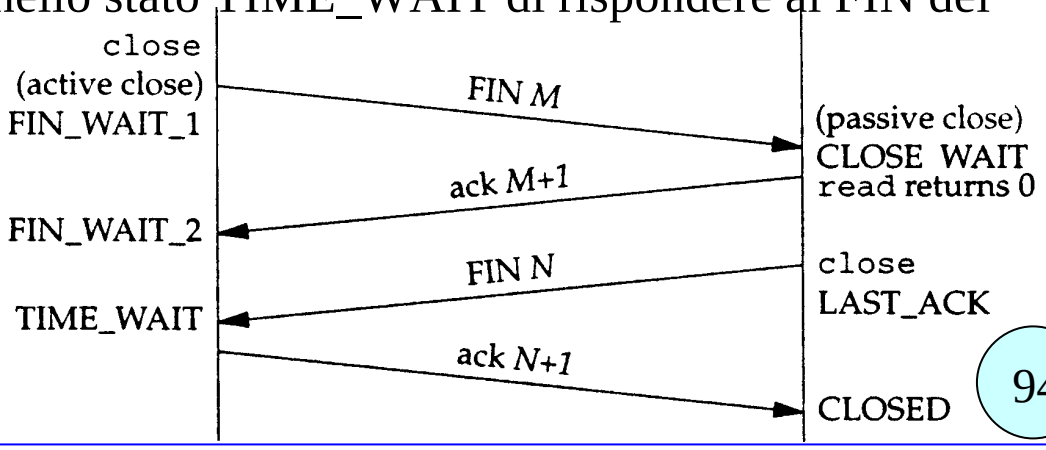
Vediamo i segmenti scambiati e gli stati assunti da client e server in una connessione TCP in cui il client chiede un servizio ed il server risponde. Il client inizia e specifica l'opzione Maximum Segment Size di 1460 byte, il server risponde e specifica una diversa richiesta di MSS di 1024. La MSS può essere diversa nelle due direzioni. Stabilita la connessione il client spedisce una richiesta al server nei dati di un solo segmento. Il server risponde spedendo la risposta nei dati di un solo segmento. Notare che, per diminuire il numero di segmenti scambiati, assieme alla risposta il server spedisce nel segmento anche l'ACK per il segmento ricevuto. Tale tecnica, detta **piggybacking**, viene utilizzata quando il ritardo nella risposta è inferiore ai 200 msec. Infine vengono utilizzati quattro segmenti per effettuare la terminazione della connessione.

TIME_WAIT state (1)



Durante la fase di chiusura della connessione, l'end system che effettua la chiusura attiva passa nello stato detto **TIME_WAIT**. **Il periodo di tempo durante il quale l'end system rimane nello stato TIME_WAIT è il doppio del MSL** (Maximum Segment Lifetime = massimo tempo di vita di un segmento) e viene detto **2MSL**. La durata dell'MSL è una scelta dell'implementazione del TCP. Il valore raccomandato in RFC 1122 è di 2 minuti, ma alcune implementazioni preferiscono 30 secondi. Quindi la durata del TIME_WAIT state varierà da 1 a 4 minuti.
Il MSL approssima il massimo tempo che un datagram IP (contenente un segmento TCP) **può rimanere in vita in internet a causa del TTL** inizialmente settato a 255.

Lo stato TIME_WAIT serve a due scopi:
1) Il primo scopo è riuscire a terminare correttamente la connessione TCP anche se il segmento finale **ACK N+1 inviato dal terminatore attivo** (che è nello stato TIME_WAIT) al passivo **viene perso**. Infatti se tale ACK viene perso il passivo ripete la trasmissione del segmento FIN N fino ad ottenere risposta. Il terminatore attivo quindi attende nello stato TIME_WAIT di rispondere al FIN del terminatore passivo.



TIME_WAIT state (2)

2) Il secondo scopo del TIME_WAIT è di impedire che segmenti duplicati dalla rete danneggino l'instaurazione di nuove connessioni con stessi indirizzi IP e di porta locali e remoti.

Una causa della duplicazione dei segmenti TCP è data dalla possibilità di avere anomalie nel routing. In particolare, se un router va in crash o se il link tra due router diventa down, è necessario un certo tempo affinche i router si organizzino e mediante gli algoritmi di routing stabiliscano percorsi alternativi. Durante questo periodo di transizione i pacchetti IP possono ritrovarsi ad essere instradati su dei loop, da A a B e di nuovo ad A, perchè i router non si sono ancora coordinati. Se durante questo periodo scade un timeout per il riscontro, il TCP assume che il segmento non sia stato ricevuto e lo ritrasmette, e il segmento ritrasmesso può seguire il nuovo giusto percorso, se i router si sono già coordinati. Se i router si coordinano prima che scada il TTL dei datagram IP che erano nel loop, anche i segmenti "dispersi" riusciranno ad arrivare a destinazione, generando così un duplicato per quel segmento. Il segmento originale, che si era perso, viene detto *lost duplicate* o *wandering duplicate.*

Ora, **se una connessione è terminata, e se ne instaura una nuova esattamente tra gli stessi hosts e le stesse porte**, **gli eventuali segmenti duplicati della vecchia connessione possono essere interpretati come segmenti della nuova connessione, e falsare le comunicazioni**. Per ovviare a questo problema il **TCP non instaura una connessione (una nuova connessione con stessi IP e stesse porte) che attualmente si trova nello stato di TIME_WAIT**, e **la durata del TIME_WAIT** (doppia del tempo di vita di ogni segmento) **impedisce che segmenti duplicati della vecchia connessione sopravvivano fino ad interessare una nuova connessione instaurata dopo la fine del TIME_WAIT**.

Socket Address Structures (1)

Cominciamo la descrizione delle Socket API (Application program Interface) dalla descrizione delle **strutture usate per trasferire indirizzi** dall'applicazione al kernel (nelle funzioni bind, connect, sendto) e dal kernel alle applicazioni (nelle funzioni accept, recvfrom, getsockname e getpeername).

• I dati definiti per Posix.1g sono quelli della seguente tabella:

int8_t	signed 8-bit integer <sys/types.h>	
uint8_t	unsigned 8-bit integer	<sys/types.h >
int16_t	signed 16-bit integer	<sys/types.h >
uint16_t	unsigned 16-bit integer	<sys/types.h>
int32_t	signed 32-bit integer	<sys/types.h>
uint32_t	unsigned 32-bit integer	<sys/types.h>
sa_family_t	famiglia di indirizzi socket	<sys/socket.h>
	AF_INET per IPv4, AF_INET6 per IPv6,	
	AF_LOCAL per indir. locali unix (per pipe ecc..)	
socklen_t	lunghezza della struttura che contiene l'indirizzo, di solito è un uint32_t	<sys/socket.h>
in_addr_t	indirizzo IPv4, = uint32	<netinet/in.h>
in_port_t	porta TCP o UDP, = uint16	<netinet/in.h>

• Poichè i socket devono fornire un'interfaccia per diverse famiglie di protocolli (IPv4, IPv6 e Unix), e poichè tali strutture vengono passate per puntatore, le funzioni di libreria presentano un argomento che è il **puntatore alla generica struttura** (**struct sockaddr***), ma essendo diversa la struttura passata a seconda della famiglia di indirizzi usata, l'argomento passato deve essere convertito mediante il cast alla struttura (struct sockaddr*), ad es:

```
struct sockaddr_in server;  /* IPv4 socket address structure */
memset ( &server, 0, sizeof(server) ); /* azzero tutta la struttura */
... riempimento dei dati della struttura server ...
bind ( socketfd, (struct sockaddr *)&server, sizeof(server) );
```

Socket Address Structures (2)

La generica struttura dell'indirizzo è dunque cosi definita:

```
struct sockaddr {
        uint8_t         sa_len;
        sa_family_t     sa_family;
        char            sa_data[14];
        };
```

La famiglia di indirizzi Ipv4 (sa_family=AF_INET) usa la struttura:

```
struct sockaddr_in {
   uint8_t          sin_len;       /* lunghezza struttura */
   sa_family_t      sin_family;  /* = AF_INET */
   in_port_t        sin_port;    /*16-bit TCP UDP port, network byte ordered */
   struct in_addr   sin_addr;   /* 32-bit IPv4 address, network byte ordered */
   char             sin_zero[8];    /* unused */
};
```

con

```
struct in_addr {                /* e' una struttura per ragioni storiche */
        in_addr_t   s_addr ;    /* 32-bit IPv4 address network byte ordered */
        };
```

• sa_len e sa_family si sovrappongono perfettamente a sin_len e sin_family rispettivamente, permettendo di leggere la costante di tipo sa_family_t e di capire che tipo di struttura si sta utilizzando.

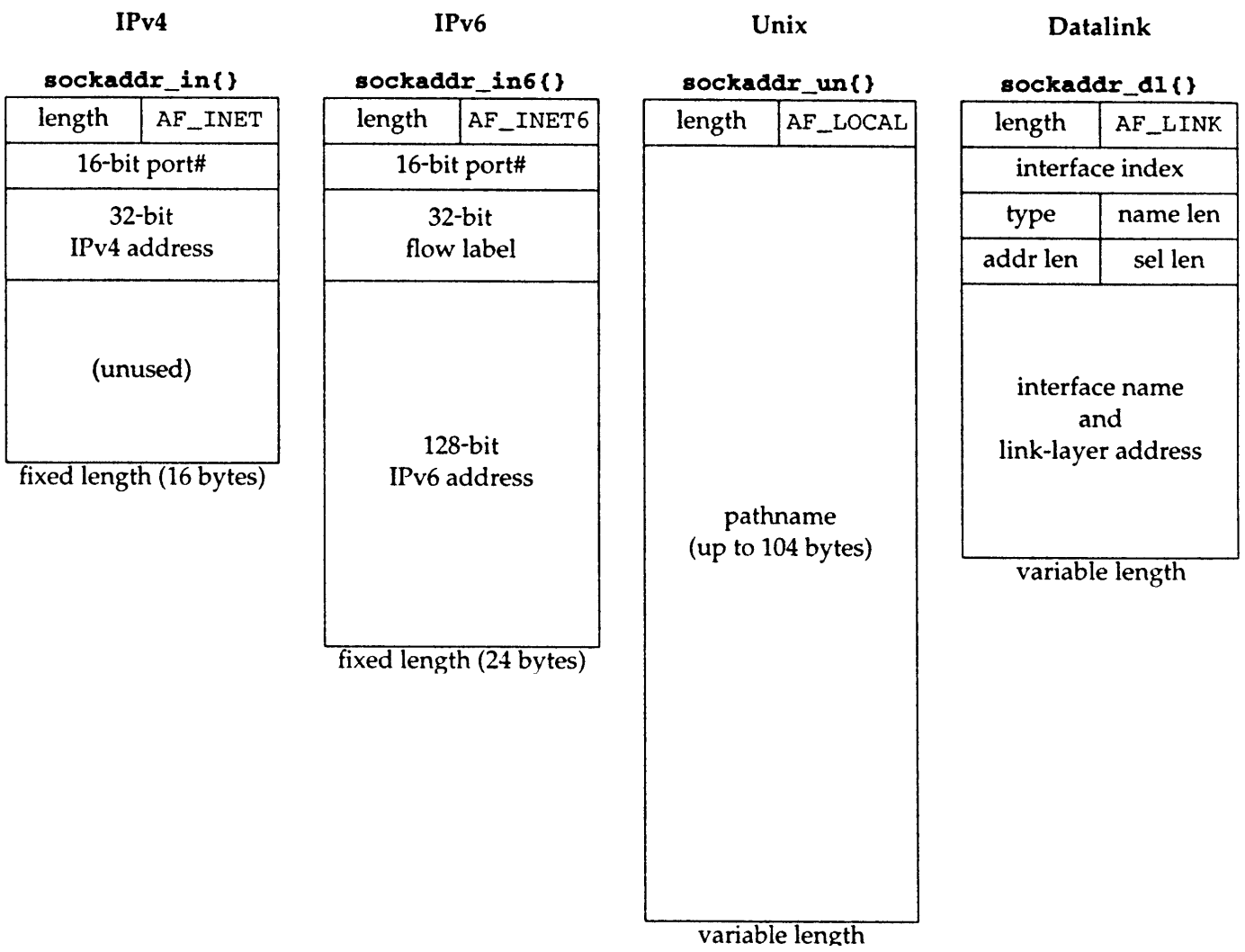
• il campo sin_len non è richiesto espressamente da Posix.1g, e anche quando è presente non è necessario settarlo, se non per applicazioni di routing, in quanto le principali funzioni in cui si passano indirizzi prevedono già un argomento in cui si passa (o riceve) la lunghezza della struttura indirizzo.

• Il campo sin_zero non è usato, ma va sempre settato tutto a zero prima di passare una struttura che lo contiene. Di più, **per convenzione, bisogna sempre settare TUTTA la struttura indirizzo tutta a zero prima di riempire i vari campi, usando la funzione memset().**

• **memset (&server, 0, sizeof(server));**

Socket Address Structure (3)

Confrontiamo alcune delle strutture usate per gli indirizzi:



Funzioni di Ordinamento dei Byte

Poichè alcuni campi delle strutture di indirizzo (i numeri di porta o gli indirizzi IPv4 ad esempio) devono essere memorizzati secondo l'ordine per i bytes stabilito per la rete (network byte order), prima di assegnare alla struttura un valore di porta (16-bit) o un indirizzo IPv4 (32-bit) è necessario convertirlo dall'ordine dei byte per l'host all'ordine per la rete, utilizzando delle funzioni di conversione, i cui prototipi sono definiti nell'include <netinet/in.h>:

uint16_t **htons** (uint16_t host16bitvalue); /* Host TO Network Short */
uint32_t **htonl** (uint32_t host32bitvalue); /* Host TO Network Long */

Viceversa, per convertire il valore di una porta o di un indirizzo IPv4, preso da una struttura di indirizzo, in un valore intero secondo l'ordinamento dell'host si devono utilizzare le funzioni:

uint16_t **ntohs** (uint16_t net16bitvalue); /* Network TO Host Short */
uint32_t **ntohl** (uint32_t net32bitvalue); /* Network TO Host Long */

Se l'ordinamento dell'host è corrispondente all'ordinamento di rete, queste funzioni sono implementate con delle macro nulle, cioè non modificano il dato.

Funzioni di Manipolazione dei Byte

Vediamo solo le funzioni portabili ovunque perche sono ANSI C.

void ***memset** (void *dest, int c, size_t n_bytes);
setta al valore c un numero len di byte a partire da dest

void *memcpy (void *dest, const void *src, size_t n_bytes);
copia n_bytes byte da src a dest, problemi se c'e' sovrapposizione, nel caso usare memmove. Resituisce dest.

void ***memcmp** (const void ptr1, const void *ptr2, size_t n_bytes);
confronta due vettori di n_bytes ciascuno, restituisce 0 se sono uguali, diverso da zero se diversi.

Funzioni di Conversione di Indirizzi IP dalla forma dotted-decimal ASCII string alla forma 32-bit network byte ordered

Queste funzioni sono definite in <arpa/inet.h>
Le funzioni **inet_aton** e **inet_addr** convertono gli indirizzi IP da una forma di stringa di caratteri ASCII decimali separati da punti del tipo “255.255.255.255”, nella forma di interi a 32-bit ordinati secondo l’ordinamento di rete.

int **inet_aton** (const char *str, struct in_addr *addrptr);

scrive nella locazione puntata da addrptr il valore a 32-bit, nell’ordine di rete, ottenuto dalla conversione della stringa zero-terminata puntata da str. Restituisce zero in caso di errore, 1 se tutto va bene.

in_addr_t inet_addr (const char *str); **NON VA USATA**

restituisce il valore a 32-bit, nell’ordine di rete, ottenuto dalla conversione della stringa zero-terminata puntata da str.
In caso di errori restituisce INADDR_NONE, e questo è un casino, perchè INADDR_NONE è un intero a 32 bit di tutti 1, che sarebbe ottenuto come risultato della chiamata di inet_addr passandogli la stringa “255.255.255.255” che è l’indirizzo valido di broadcast.
Per evitare confusione non deve essere usata.

Infine c’e’ una funzione che effettua la conversione inversa, da interi a 32-bit network ordered verso stringhe ASCII decimali separate da punti.

char ***inet_ntoa** (struct in_addr addr);

scrive in una locazione di memoria statica (di cui restituisce un puntatore) la stringa ASCII null-terminata di caratteri decimali separati da punti corrispondeni all’indirizzo IP a 32-bit, nell’ordine di rete, contenuto nella struttura addr (che stranamente non è un puntatore). Occhio, questa funzione non è rientrante, perchè memorizza il risultato in una locazione statica.

I/O su Socket TCP (1)

I socket TCP, una volta che la connessione TCP sia stata instaurata, sono accedibili come se fossero dei file, mediante un descrittore di file (un intero) ottenuto tramite una socket() o una accept() o una connect(). Con questo descrittore è possibile effettuare letture tramite la funzione read, che restituisce i byte letti dal flusso in entrata, e scritture tramite la funzione write, che spedisce i byte costituendo il flusso in uscita.

ssize_t **read** (int fd, void *buf, size_t count);

cerca di leggere count byte dal file descriptor fd, scrivendoli nel buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata la lettura e viene restituito il numero di byte letti. Se viene restituito zero significa end-of-file (fine stream). Se viene restituito -1 è accaduto un errore e viene settato la variabile globale errno definita in <errno.h>.

La funzione read presenta una particolarità quando è applicata ad un socket. Può accadere che la read() restituisca meno byte di quanti richiesti, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la read (richiedendo il numero dei byte mancanti) fino ad ottenerli tutti. Oss.: ssize_t è definito in <unistd.h> ed è un long.

ssize_t **write** (int fd, const void *buf, size_t count);

cerca di scrivere fino a count byte sul file descriptor fd, leggendoli dal buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata la scrittura e viene restituito il numero di byte scritti. Se viene restituito -1 è accaduto un errore e viene settato errno.

Analogamente alla read() anche la write presenta una particolarità quando è applicata ad un socket. Può accadere che la write() scriva meno byte di quanto richiesto, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la write (con i soli byte mancanti) fino a scriverli tutti.

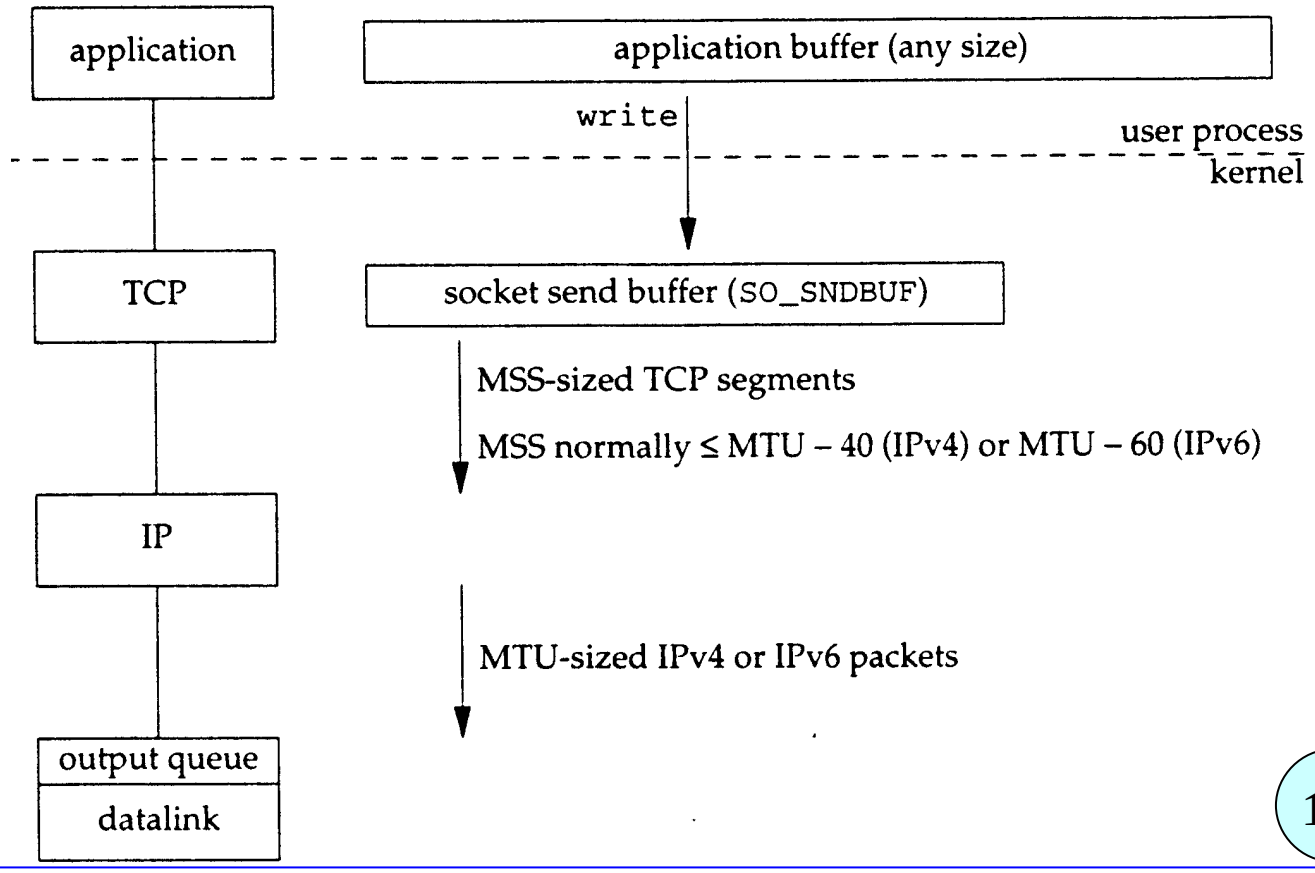
I/O su Socket TCP: (2) TCP Output

Ogni socket TCP possiede un buffer per l'output (send buffer) in cui vengono collocati temporaneamente i dati che dovranno essere trasmessi mediante la connessione instaurata. La dimensione di questo buffer può essere configurata mediante un'opzione SO_SNDBUF.

Quando un'applicazione chiama write() per n bite sul socket TCP, il kernel cerca di copiare n byte dal buffer dell'appl. al buffer del socket.

Se il buffer del socket e' più piccolo di n byte, oppure è già parzialmente occupato da dati non ancora trasmessi e non c'è spazio sufficiente, verranno copiati solo nc<n byte, e verrà restituito dalla write il numero nc di byte copiati.

Se il socket ha le impostazioni di default, cioè è di tipo bloccante, la fne della routine write ci dice che sono stati scritti sul buffer del socket quegli nc byte, e possiamo quindi riutilizzare le prime nc posizioni del buffer dell'applicazione. Ciò non significa affatto che già i dati siano stati trasmessi all'altro end-system.



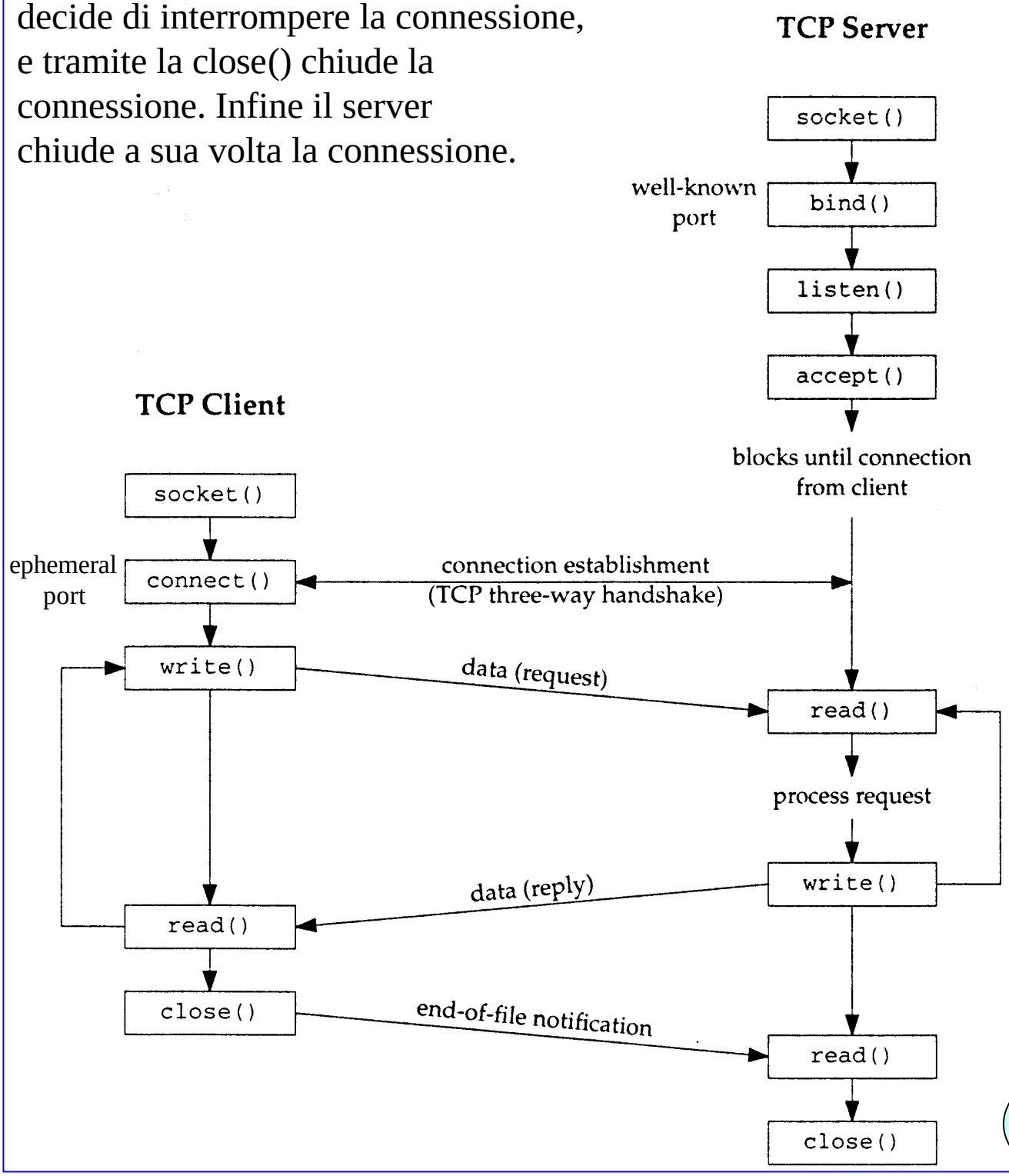
I/O su Socket TCP (3)

Per semplificarsi la vita ed usare delle funzioni per l'I/O con i socket che si comportano esattamente come le read() e write() su file, si può scrivere due proprie funzioni read**n**() e write**n**() che effettuano un loop di letture/scritture fino a leggere/scrivere tutti gli n byte come richiesto, o incontrare l'end-of-file o riscontrare un errore.

```
ssize_t  readn (int fd, void *buf, size_t n)
{   size_t  nleft;     ssize_t  nread;  char   *ptr;
    ptr = buf;  nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)   nread = 0;    /* and call read() again */
            else                          return(-1);
        }
        else if (nread == 0)
                break;    /* EOF, esce */
        nleft -= nread;   ptr   += nread;
    }
    return(n - nleft);                /* return >= 0 */
}
ssize_t  writen (int fd, const void *buf, size_t n)
{   size_t  nleft;     ssize_t  nread;  char   *ptr;
    ptr = buf;  nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)   nwritten = 0;   /* and call write() again*/
            else                          return(-1);       /* error */
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

Interazioni tra Client e Server TCP

Per primo viene fatto partire il server, poi viene fatto partire il client che chiede la connessione al server e la connessione viene instaurata. Nell'esempio (ma non è obbligatorio) il client spedisce una richiesta al server, questo risponde trasmettendo alcuni dati. Questa trasmissione bidirezionale continua fino a che uno dei due (il client nell'esempio) decide di interrompere la connessione, e tramite la close() chiude la connessione. Infine il server chiude a sua volta la connessione.



funzione **socket**()

La prima azione per fare dell'I/O da rete è la chiamata alla funziona socket() specificando il tipo di protocollo di comunicazione da utilizzare (TCP con IPv4, UDP con IPv6, Unix domain stream protocol per usare le pipe).

#include <sys/socket.h>

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

restituisce un descrittore di socket maggiore o uguale a zero, oppure -1 in caso di errore, e setta errno.

L'argomento family specifica la famiglia di protocolli da utilizzare.

family	descrizione
AF_INET	IPv4 protocol
AF_INET6	IPv6 protocol
AF_LOCAL	Unix domain protocols (ex AF_UNIX)
AF_ROUTE	Routing socket
AF_ROUTE	Key socket (sicurezza in IPv6)

L'argomento type specifica quale tipo di protocollo vogliamo utilizzare all'interno della famiglia di protocolli specificata da family.

type	descrizione
SOCK_STREAM	socket di tipo stream (connesso affidabile)
SOCK_DGRAM	socket di tipo datagram
SOCK_DRAW	socket di tipo raw (livello network)

L'argomento protocol di solito è settato a 0, tranne che nel caso dei socket raw.

Non tutte le combinazioni di family e type sono valide. Quelle valide selezionano un protocollo che verrà utilizzato.

	AF_INET	AF_INET6	AF_LOCAL	AF_KEY AF_ROUTE
SOCK_STREAM	TCP	TCP	esiste	
SOCK_DGRAM	UDP	UDP	esiste	
SOCK_DRAW	IPv4	IPv6		esiste

funzione **connect**()

La funzione connect() è usata dal client TCP per stabilire la connessione con un server TCP.

#include <sys/socket.h>

int **connect** (int socketfd, const struct sockaddr *servaddr,
socklen_t addrlen);

restituisce 0 se la connessione viene stabilita, -1 in caso di errore.

• L'argomento **socketfd** è un descrittore socket ottenuto da una chiamata alla funzione socket().

• L'argomento **servaddr** come visto in precedenza è in realtà per IPv4 un puntatore alla struttura sockaddr_in, e **deve specificare l'indirizzo IP e il numero di porta del server da connettere**.

• L'argomento **addrlen** specifica la dimensione della struttura dati che contiene l'indirizzo del server servaddr, viene di solito assegnata mediante la sizeof(servaddr).

• Il client non deve di solito specificare il proprio indirizzo IP e la propria porta, perchè queste informazioni non servono a nessuno. Quindi può chiedere al sistema operativo di assegnargli una porta TCP qualsiasi, e come indirizzo IP l'indirizzo della sua interfaccia di rete, o dell'interfaccia di rete usata se ne ha più di una. Quindi NON SERVE la chiamata alla bind() prima della connect().

• Nel caso di connessione TCP la connect inizia il protocollo three way handshake spedendo un segmento SYN. La funzione termina o quando la connessione è stabilita o in caso di errore.

• In caso di errore la connect restituisce -1 e la variabile errno è settata a:

- ETIMEDOUT nessuna risposta al segmento SYN
- ECONNREFUSED il server risponde con un segmento RST (reset) ad indicare che nessun processo server è in attesa (stato LISTEN) su quella porta
- EHOSTUNREACH o ENETUNREACH host non raggiungibile
- ed altri ancora.

funzione **bind()** (1)

La funzione bind() collega al socket un indirizzo locale. Per TCP e UDP ciò significa assegnare un indirizzo IP ed una porta a 16-bit.

#include <sys/socket.h>

int **bind** (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);

restituisce 0 se tutto OK, -1 in caso di errore.

• L'argomento **sockfd** è un descrittore ottenuto da una socket().

• L'argomento **myaddr** è un puntatore alla struttura sockaddr_in, e specifica l'eventuale indirizzo IP **locale** e l'eventuale numero di porta **locale** a cui il sistema operativo deve collegare il socket.

• L'argomento **addrlen** specifica la dimensione della struttura myaddr.

• L'applicazione può collegarsi o no ad una porta.

  • Di solito il server si collega ad una porta nota (well know port). Fa eccezione il meccanismo delle RPC.

  • I client di solito non si collegano ad una porta con la bind.

  • In caso non venga effettuato il collegamento con una porta, il kernel effettua autonomamente il collegamento con una porta qualsiasi (ephemeral port) al momento della connect (per il client) o della listen (per il server).

• L'applicazione può specificare (con la bind) per il socket un indirizzo IP di un'interfaccia dell'host stesso.

  • Per un TCP client ciò significa assegnare il source IP address che verrà inserito negli IP datagram, spediti dal socket.

  • Per un TCP server ciò significa che verranno accettate solo le connessioni per i client che chiedono di connettersi proprio a quell'IP address.

  • Se il TCP client non fa la bind() o non specifica un IP address nella bind(), il kernel sceglie come source IP address, nel momento in cui il socket si connette, quello della interfaccia di rete usata.

  • Se il server non fa il bind con un IP address, il kernel assegna al socket come indirizzo IP locale quello contenuto nell'IP destination address del datagram IP che contiene il SYN segment ricevuto

funzione **bind()** (2)

Chiamando la bind() si può specificare o no l'indirizzp IP e la porta, assegnando valori ai due campi **sin_addr** e **sin_port** della struttura sockaddr_in passata alla bind come secondo argomento.
A seconda del valore otteniamo risultati diversi, che sono qui elencati, nella tabella che si riferisce solo al caso:

IP_address sin_addr	port sin_port	Risultato
wildcard	0	il kernel sceglie IP address e porta
wildcard	nonzero	il kernel sceglie IP address, porta fissata
Local IP Address	0	IP address fissato, kernel sceglie la porta
Local IP Address	non zero	IP address e porta fissati dal processo

Specificando il numero di porta 0 il kernel sceglie collega il socket ad un numero di porta temporaneo nel momento in cui la bind() è chiamata.
Specificando la wildcard (mediante la costante **INADDR_ANY** per IPv4) il kernel non sceglie l'indirizzo IP locale fino a che o il socket è connesso (se TCP) o viene inviato il primo datagram per quel socket (se UDP).

L'assegnazione viene fatta con le istruzioni:
struct sockaddr_in localaddr;
localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
localaddr.sin_port = htons(port_number);

Se con la bind si lascia al kernel la scelta di IP address locale o port number locale, una volta che il kernel avrà scelto, si potrà sapere quale IP address e quale port number è stato scelto mediante la funzione **getsockname()**.

funzione **listen**()

La funzione listen **è chiamata solo dal TCP server** e esegue due azioni:
1) ordina al kernel di far passare il socket dallo stato iniziale CLOSED allo stato LISTEN, e di accettare richieste di inizio connessione per quel socket, accodandole in delle code del kernel.
2) specifica al kernel quante richieste di inizio connessione può accodare al massimo per quel socket.

#include <sys/socket.h>
int **listen** (int socketfd, int backlog);
restituisce 0 se tutto OK, -1 in caso di errore.

• L'argomento **socketfd** è un descrittore ottenuto da una socket().
• L'argomento **backlog** è un intero che specifica quante richieste di inizio connessione (sia connessioni non ancora stabilite, cioè che non hanno ancora raggiunto lo stato ESTABLISHED, sia connessioni stabilite) il kernel può mantenere in attesa nelle sue code.
• Quando un segmento SYN arriva da un client, se il TCP verifica che c'è un socket per quella richiesta, crea una nuova entry in una **coda delle connessioni incomplete**, e risponde con il suo FIN+ACK secondo il 3-way handshake. L'entry rimane nella coda fino a che il 3-way è terminato o scade il timeout.
• Quando il 3-way termina normalmente, la connessione viene instaurata, e la entry viene spostata in **una coda delle connessioni completate**.
• Quando il server chiama la accept, la prima delle entry nella **coda delle connessioni completate** viene consegnata alla accept() che ne restituisce l'indice come risultato, ovvero restituisce un nuovo socket che identifica la nuova connessione.
• Se quando il server chiama la accept(), la coda delle connessioni completate è vuota, la accept resta in attesa.
• L'argomento backlog specifica il numero totale di entry dei due tipi di code.
• Solitamente si usa 5, per http daemon si usano valori molto grandi.

funzione **accept**()

La funzione accept **è chiamata solo dal TCP server** e restituisce la prima entry nella **coda delle connessioni già completate** per quel socket. Se la coda è vuota la accept resta in attesa.

```
#include <sys/socket.h>
int accept (int socketfd, struct sockaddr *cli_addr,
                socklen_t *ptraddrlen);
```

restituisce un descrittore socket >=0 se tutto OK, -1 in caso di errore.

L'argomento **socketfd** è un descrittore ottenuto da una socket() e in seguito processato da bind() e listen(). E' il cosiddetto **listening socket**, ovvero il socket che si occupa di insturare le connessioni con i client che lo richiedono, secondo le impostazioni definite dalla bind() e dalla listen(). Tale listening socket viene utilizzato per accedere alla coda delle connessioni instaurate come visto per la listen().

• L'argomento **cli_addr** è un puntatore alla struttura sockaddr_in, su cui la funzione accept scrive l'indirizzo IP **del client** e il numero di porta **del client**, con cui è stata instaurata la connessione a cui si riferisce il socket che viene restituito come risultato .

• L'argomento **ptraddrlen** è un puntatore alla dimensione della struttura cli_addr che viene restituita.

Se accept termina correttamente restituisce un nuovo descrittore di socket che è il **connected socket**, cioè si riferisce ad una connessione instaurata con un certo client secondo le regole del listening socket socketfd passato come input. Il connected socket verrà utilizzato per scambiare i dati nella nuova connessione.

Il **listening socket** socketfd (il primo argomento) mantiene anche dopo la accept le impostazioni originali, e può essere riutilizzato in una nuova accept per farsi affidare dal kernel una nuova connessione.