Il modello Client-Server e le Interfacce Socket

S. Gianelloni e-mail (giannel@ux4sns.sns.it)
U. Parrini e-mail (parrini@ux4sns.sns.it)
G.A. Romano e-mail (g.romano@cnuce.cnr.it)

CNUCE C96-017

Pisa, 25 Luglio 1996

La sempre maggior espansione delle reti geografiche di questi ultimi anni, ed il loro conseguente successo (Internet), ha creato una crescente richiesta, nel campo della ricerca prima ed in quello commerciale successivamente, di poter accedere a vari tipi di risorse presenti fisicamente in qualsiasi parte del mondo.

Sono ormai noti a tutti i diversi strumenti che attualmente forniscono accesso ai servizi Internet come ad esempio FTP (File Tranfer Protocol), WAIS (Wide Area Information Servers), Telnet, WWW (World Wide Web) E-Mail ect; questi sono tutti applicativi standardizzati basati sul modello client-server ed ormai diventati strumenti utilizzati quotidianamente da milioni di utenti.

Il seguente manuale ha lo scopo di fornire i principi base di progetto per la realizzazione di nuovi applicativi client-server sulla rete Internet che utilizzano il protocollo di comunicazione TCP per gli ambienti operativi Unix e MS-Windows.

Gran parte degli argomenti presenti in questa pubblicazione sono stati trattati nella tesi di Laurea in Scienze dell'Informazione, presso l'Università di Pisa, del 19 Luglio '96 il cui titolo è "Un Server Windows per l'accesso ad un sistema di information retrieval (CDS/ISIS) da Server WAIS su piattaforma Unix.".

I concetti trattati riguardano fondamentalmente l'interazione di programmi applicativi tramite la rete Internet (IPC InterProcess Communication) e sono principalmente:

- Il modello funzionale.
- Le caratteristiche del protocollo di comunicazione TCP.
- Le interfacce di comunicazione negli ambienti UNIX e Windows.

In questo testo vengono quindi esaminati i concetti fondamentali del modello client-server, le proprietà del protocollo di comunicazione TCP e le interfacce tra i programmi applicativi ed il software del protocollo nei sistemi operativi UNIX e Windows, infine nella parte conclusiva vengono forniti esempi di schemi di programmi per la realizzazione di moduli di rete (client e server) nel sistema UNIX e Windows.

1.1 Modello funzionale Client-Server in rete

Il modello predominante per avere un meccanismo di comunicazione tra processi in rete è il paradigma client-server. La scelta di questo modello di comunicazione è dovuta al fatto che il paradigma client-server è un'estensione conveniente e naturale della comunicazione fra processi in una singola macchina ed è quindi facile realizzare dei programmi che impiegano tale modello anche per interagire su rete.

Con il termine Server possiamo definire un qualsiasi programma in esecuzione che offre un servizio che possa essere raggiunto attraverso la rete, quindi il server accetta le richieste che arrivano sulla rete, esegue il suo servizio e riporta i risultati al richiedente client. Alcuni tipici esempi di servizi forniti da un server possono essere:

- ritorno dell'ora del giorno (daytime)
- terminale remoto (telnet)
- trasferimento file (ftp)
- posta elettronica (e-mail)
- ricerca di termini su database (wais)
- collegamenti e consultazione di documenti (www)

Con il termine Client si definisce un programma in esecuzione che diventa tale quando invia una richiesta ad un server ed attende una risposta.

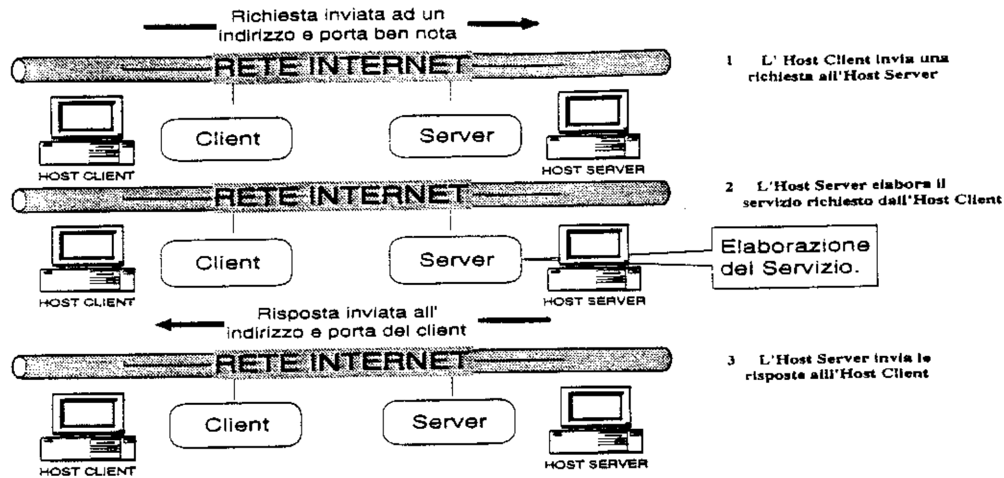


Figura 1 Schema funzionale client-server su rete

I passi che in modo generale i due processi svolgono sono:
Per quando riguarda il Server, il quale viene sempre avviato per primo rispetto al client, abbiamo:

Step 1 - Apre un canale di comunicazione ed informa l'host locale della sua disponibilità ad accettare le richieste del client a qualche indirizzo ben noto che ha stabilito.
Step 2 - Attende che la richiesta di un client arrivi all'indirizzo ben noto.
Step 3 - Esegue il suo servizio e trasmette la risposta.
Step 4 - Torna al passo 2 cioè ad aspettare un'altra richiesta di un client.

Il processo Client svolge invece un insieme diverso di azioni e viene avviato sempre dopo che è stato avviato il server:

Step 1 - Apre un canale di comunicazione e si connette ad uno specifico indirizzo ben note (cioè al server che vuole contattare).
Step 2 - Invia al server i messaggi di richiesta di servizio e riceve la risposta.
Step 3 - Chiude il canale di comunicazione e termina.

1.2 Classificazione dei tipi di server

I servers sono molto più difficili da realizzare rispetto ai clients perchè devono, quando il sistema operativo lo permette, gestire più richieste concorrentemente.
La possibilità o meno di gestire più richieste concorrentemente permette di definire tre tipi di server:

- **Server Iterativi**
- **Server Concorrenti** e
- **Server Apparentemente Concorrenti.**

Si usa il termine di ***Server Iterativo*** per descrivere l'implementazione di un server che elabora una richiesta alla volta; questo è tipicamente un server sequenziale poiché dopo aver accettato una richiesta, il server prepara una risposta e la trasmette prima di tornare a vedere se è arrivata un'altra richiesta.
In questa soluzione si ipotizza implicitamente che il sistema operativo accodi le richieste che arrivano per un server mentre lo stesso è occupato, e che la coda di attesa non diventi troppo lunga perchè il server ha soltanto una ridotta quantità di lavoro da svolgere.

I servers iterativi sono tipicamente usati in tutte quelle applicazioni in cui si sa in anticipo sia il tempo necessario per gestire ciascuna richiesta e che tale tempo è minimo.

I ***Servers Concorrenti*** sono quelli che permettono di gestire più richieste concorrentemente , anche se l'elaborazione della singola richiesta richiedesse un tempo considerevole; in questo tipo di soluzione è fondamentale che il sistema operativo sottostante sia un multi-tasking preemptive e che sia possibile a livello implementativo la creazione dinamica dei processi.

Servers di questo tipo sono utilizzati in tutte quelle applicazioni nelle quali la quantità di lavoro necessaria per gestire una richiesta è incognito, per cui il server avvia un altro processo per gestire ciascuna richiesta.

Normalmente, i server concorrenti sono costituiti da due parti: un singolo processo master che è responsabile di accettare nuove richieste, ed un insieme di processi slaves che hanno la responsabilità di gestire le singole richieste.

Poiché il master avvia uno slave per ogni nuova richiesta che arriva, l'elaborazione procede concorrentemente; pertanto le richieste che richiedono poco tempo per essere completate possono finire prima di quelle che richiedono più tempo, indipendentemente dall'ordine in cui sono state avviate.

Per esempio, si supponga che il primo client che ha contattato un file server richieda un grande trasferimento di file per cui occorrono parecchi minuti.

Se un secondo client contatta il server per richiedere un trasferimento per cui occorrono solo pochi secondi, il secondo trasferimento può essere avviato e completare mentre il primo è ancore in corso.

Infine, quando un'implementazione concorrente non è possibile (cioè quando il sistema operativo non è un vero multi-tasking e non esiste la possibilità di creazione dinamica dei processi) oppure quando il server deve eseguire delle elaborazioni che richiedono tempi molto piccoli, oppure quando l'overhead dovuto alla creazione dinamica di nuovi processi è tale da sconsigliarne l'utilizzazione, allora in tutte queste situazioni è possibile implementare un ***Server Apparentemente Concorrente*** il quale è costituito da un singolo processo che usa il meccanismo dell' I/O asincrono per permettere il simultaneo uso di più canali di comunicazione.

In quest'ultimo tipo di server si è usato il termine concorrente poiché da un punto di vista del client, il server sembra che comunichi con più clients concorrentemente, il punto fondamentale è che in questa classificazione il termine server apparentemente concorrente significa che il server gestisce più richieste simultaneamente e non che la sottostante implementazione usa la concorrenza multipla tra processi.

La classificazione dei server in base alla loro capacità di gestire più richieste o più canali di comunicazione concorrentemente si può estendere definendo altri due tipi di server:

- **server multiprotocollo**
- **server multiservizio**

Un processo server in grado di gestire richieste per un servizio attraverso più protocolli di comunicazione viene definito ***server multiprotocollo***. Un tipico esempio è quello relativo al servizio Daytime del server inetd di Unix il quale gestisce le richieste sia attraverso il protocollo UDP sia attraverso il protocollo TCP.

La motivazione principale per utilizzare un server multiprotocollo emerge quando per entrambi i protocolli viene utilizzato lo stesso programma per elaborare il servizio richiesto; in questo caso, rispetto ad un'implementazione che prevede la creazione di un processo per ogni protocollo, abbiamo un minor dispendio delle risorse.

I servers multiprotocollo possono implementarsi seguendo uno schema concorrente o apparentemente concorrente.

Un ***server multiservizio*** è un particolare tipo di server in grado di gestire servizi multipli. Un esempio tipico di tale server è sempre il demone inetd di Unix (il quale oltre ad essere multiprotocollo è anche multiservizio) gestisce alcuni dei servizi standard di internet quali Time, Echo, Ping ed altri.

Gli stessi vantaggi che hanno motivato la scelta di progetto di un server multiprotocollo si hanno anche nel caso di server multiservizio, ed anche per questo tipo di server possiamo avere una versione concorrente o quella basata sull'I/O asincrono.

Infine possiamo ulteriormente classificare i servers in base alla loro capacità o meno di mantenere informazioni sullo stato delle interazioni con i clients (*state information*); server che non mantengono informazioni di stato sono chiamati **stateless servers** mentre quelli che mantengono informazioni di stato sono chiamati **stateful servers**.

1.3 Protocolli connection-oriented e connectionless

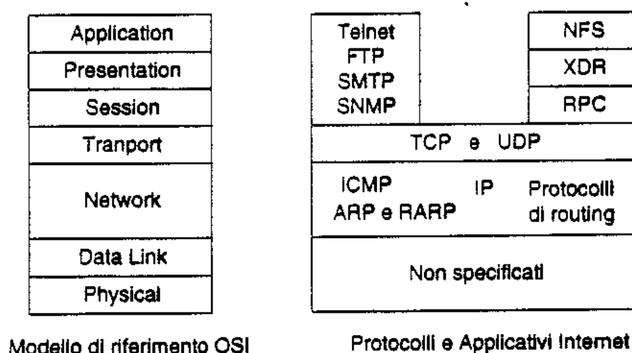
Per la realizzazione di un applicativo distribuito il primo passo che il progettista deve compiere è quello relativo alla scelta del protocollo di trasporto che un client usa per accedere al servizio che il server intende offrire.

Nella suite di protocolli Internet, il TCP fornisce un servizio di trasporto connection-oriented, mentre l'UDP fornisce un servizio connectionless. Pertanto, clients e servers che usano il protocollo TCP si definiscono rispettivamente *client* e *server connection-oriented* mentre quelli che usano il protocollo UDP si definiscono *client* e *server connectionless.*

Anche a livello di protocollo applicativo (cioè in quello strato di software al di sopra dei protocolli di trasporto che è responsabile delle regole e convenzioni in base alle quali client e server interagiscono) è importante scegliere tra una implementazione connectionless o una implementazione connection-oriented, poiché un protocollo applicativo destinato ad usare un servizio di trasporto connection-oriented può operare scorrettamente o inefficacemente quando usa un servizio di trasporto connectionless.

Prima di passare a descrivere le caratteristiche dei due protocolli vediamo molto brevemente (non è lo scopo di questo manuale) come è fatta l'architettura di rete TCP/IP, in particolare "paragoniamo" l'Internet Protocol Suite con il modello di riferimento OSI (Open System Interconnection).

Nella figura sottostante solo schematizzati i principali protocolli presenti nell' Internet.



Modello di riferimento OSI   Protocolli e Applicativi Internet

I primi due livelli dell'OSI si chiamano physical e data link, e l'Internet Protocol Suite non cerca di specificare questi due livelli ma parte direttamente dal livello tre che è il livello di network in cui propone il protocollo IP; IP è il protocollo centrale nel livello network dell'architettura di rete TCP-IP ma non è l'unico protocollo che opera a questo livello perché abbiamo altri protocolli come ARP e RARP, un altro protocollo importante che si chiama ICMP e poi abbiamo dei protocolli di Routing ossia protocolli per il calcolo delle tabelle di instradamento.

A livello transport l'architettura di rete TCP-IP offre due soluzioni alternative: o il TCP oppure il protocollo UDP; quando diciamo due soluzioni alternative significa che alcuni applicativi preferiscono usare il TCP altri applicativi preferiscono usare l'UDP cioé sono due protocolli di trasporto alternativi.

In TCP-IP non c'è una netta distinzione tra livelli session presentation e application ma, nella maggioranza dei casi, ci sono in qualche modo degli applicativi che si interfacciano direttamente al livello TCP e IP; questi applicativi vanno dal classico Telnet all'FTP all'SMTP (Simple Mail Tranfer Protocol) applicativo della posta elettronica, all'SNMP (Simple Network Management Protocol) il quale è un protocollo che ci serve per fare gestione della rete stessa cioè gestione degli apparati. Oltre a questi applicativi storici ci sono state delle estensioni la più rilevante delle quali è quella che riguarda NFS (Network File System) la quale si appoggia su una sorta di livello presentation per rentersi indipendente dalla rappresentazone dell'informazione ed è XDR (External Data Rappresentation) e poi si appoggia per stabilire la comunicazione su una libreria che si chiama RPC (Remote Procedure Call).

Chiaramente, sia il servizio connectionless che il servizio connection-oriented presentano vantaggi e svantaggi.

Il principale vantaggio che abbiamo nel sviluppare un applicativo di rete che utilizza un servizio connection-oriented sta nella facilità di implementazione, infatti in questo caso sia il client che il server non si devono preoccupare di gestire il problema dei pacchetti persi e di quelli arrivati fuori ordine poiché è lo stesso protocollo che li gestisce automaticamente.

Con questo tipo di servizio client e server gestiscono e usano le connessioni.

Il server, ad esempio, accetta le connessioni in arrivo e poi spedisce tutte le informazioni attraverso la connessione, cioè riceve richieste dal client e spedisce i risultati, alla fine il server chiude la connessione dopo che le interazioni con il client sono terminate.

Nel tempo in cui una connessione rimane aperta, il TCP fornisce tutta l'affidabilità di cui abbiamo bisogno, cioè ritrasmette i dati perduti, verifica che i dati arrivino senza errori e riordina i pacchetti se necessario.

Quando un client manda una richiesta, il TCP o consegna la richiesta in modo affidabile oppure informa il client che la connessione è stata interrotta, lo stessa cosa avviene con il server il quale delega al TCP la responsabilità di consegnare le risposte o di informarlo che la connessione è interrotta.

Clients e servers connection-oriented hanno anche degli svantaggi, infatti questo tipo di servizio richiede un socket separato per ciascuna connessione, mentre il servizio connectionless permette la comunicazione con hosts multipli sullo stesso socket. Per semplici applicazioni, l'overhead necessario per stabilire e terminare una connessione attraverso l'handshake a tre vie del protocollo TCP è notevolmente più dispendiosa rispetto al servizio connectionless.

Un'ulteriore svantaggio del servizio connection-oriented consiste nel fatto che le possibili cadute del client non consentono al server di recuperare tutte le risorse (socket, buffer, ect) che erano state allocate per instaurare la comunicazione; questo problema nel caso di ripetute cadute del client porterebbero ad una terminazione del server per mancanza di risorse.

Infine, poiché TCP offre una comunicazione di tipo *point-to-point*, non è possibile supportare una comunicazione *broadcast* o *multicast*, così ogni server che accetta o risponde a comunicazioni multicast deve essere necessariamente connectionless.

Poiché il protocollo UDP non supporta la consegna affidabile, il trasporto connectionless costringe il protocollo applicativo a provvedere all'affidabilità (quando richiesta), attraverso una complessa e complicata tecnica conosciuta come ritrasmissione adattabile (adaptive retransmission) basata su timeout e ritrasmissione.

In base al servizio di trasporto utilizzato (connectionless o connection-oriented) ed in base al tipo di gestione delle richieste provenienti dai clients possiamo avere le seguenti sei potenziali combinazioni:

Per ciascun tipo di server, attraverso uno pseudolinguaggio, è possibile definire un algoritmo che descrive i passi svolti dal server evidenziando i processi ed i sockets coinvolti nella comunicazione.

	Server Iterativo	Server Concorrente	Server Apparentemente concorrente (I/O asincrono)
Servizio connetion-oriented	Server iterativo connection-oriented	Server concorrente connection-oriented	Server apparentemente concorrente connection-oriented
Servizio connetionless	Server iterativo connectionless	Server concorrente connectionless	Server apparentemente concorrente connectionless

Figura 2

1.4 Proprietà del servizio di consegna affidabile del TCP

Prima di descrivere gli algoritmi utilizzati dai vari tipi di server è opportuno riassumere le caratteristiche e le proprietà del servizio di consegna affidabile del protocollo TCP e chiarire alcuni concetti chiave del protocollo TCP quali: porte, connessioni, punti terminali, aperture passive e attive.

In primo luogo vediamo alcune delle caratteristiche più rilevanti di questo protocollo il quale è attualmente considerato lo standard de facto dei protocolli connessi; questo protocollo, anche se fa parte costituzionalmente della suite dei protocolli Internet (TCP/IP), è un protocollo indipendente, di applicabilità generale, che può essere adattato per l'impiego con altri sistemi di consegna.

Per esempio, poiché il TCP fa pochissime ipotesi sulla rete sottostante, è possibile utilizzarlo in una singola rete, come un Ethernet (LAN), come in un'internet complessa (WAN o MAN); in effetti, il TCP è diventato così popolare che uno dei protocolli di sistema aperti dell'ISO (International Organization for Standardization), il TP-4, è un suo derivato.

Il Transport Control Protocol è utilizzato da applicativi che richiedono la trasmissione affidabile dell'informazione, alcuni dei principali applicativi che utilizzano il TCP sono ad esempio:

Telnet	Terminale remoto
FTP	File Transfer Protocol
SMTP	Simple Mail Transport Protocol
WWW	World Wide Web
WAIS	Wide Area Information Servers

Attraverso questo protocollo connection-oriented quando un nodo (host) deve comunicare con un altro nodo crea un **circuito virtuale** e come tutti i protocolli di tipo connesso il modello con connessione prevede una fase preliminare in cui viene stabilita una connessione stabile tra i due corrispondenti (circuito virtuale), una fase di scambio informazioni e una fase di abbattimento della connessione.

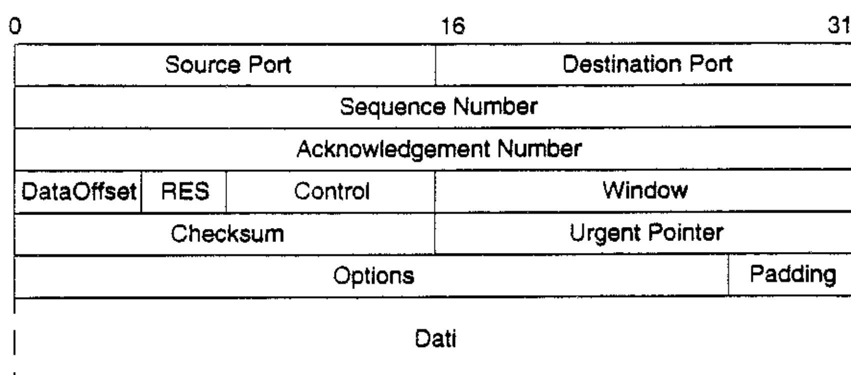
Al circuito virtuale è appunto associato un protocollo di trasporto che oltre ad essere connesso è *full-duplex* (cioè è in grado di scambiare le informazioni contemporaneamente dal nodo A al nodo B e da nodo B al nodo A), ha meccanismi di acknowledge (ossia ha dei meccanismi con cui A da l'acknowledge delle informazioni che riceve da B e B da l'acknowledge delle informazioni corrette che riceve da A) e ha

meccanismi di controllo del flusso (meccanismi in cui "tara" la velocità con cui i dati vengono scambiati sulla rete rallentando eventualmente le trasmissioni troppo veloci che possono creare degli intasamenti quindi creare problemi di flusso dell'informazione sulla rete).

Il TCP segmenta e riassembla i dati secondo le sue necessità ossia non c'è nessuna corrispondenza tra la segmentazione che fa un nodo TCP con il mittente e il numero di segmentazioni che fa il nodo TCP con il destinatario, entrambi (mittente e destinatario) gestiscono il protocollo ma i dati che scambiano con l'applicativo è in numero variabile di pacchetti non correlato tra sender e receiver (il TCP non garantisce nessuna relazione tra il numero di read e quello di write con gli applicativi mittente e destinatario).

Questo protocollo come la maggior parte dei protocolli connessi è un protocollo con *sliding windows, timeout* e *ritrasmissione*; sliding windows significa che non dobbiamo immediatamente fornire l'acknoledge di un dato trasmesso ma esiste una certa quantità di dati che possono essere "fuori" cioè che possono essere in attesa di acknoledge, mano mano i dati di cui forniamo gli acknoledge vengono tolti dalla possibile coda di ritrasmissione perché senz'altro arrivati a destinazione e la finestra (window) viene spostata. Se ad un certo punto non forniamo più acknoledge c'è un certo meccanismo di timeout e si innesca la ritrasmissione tipicamente la ritrasmissione dell'intera window; questo implica che dobbiamo fissare a priori la window, inoltre tale dimensione dovrà essere espressa in byte ed non in segmenti.

Se vediamo il pacchetto TCP questo ha un campo "window" il quale indica quanti byte possono ancora essere trasmessi prima di un ack, in tale pacchetto come mostrato nella figura sottostante sono presenti:



una *souce port* e una *destination port* le quali sono le porte per effettuare il multiplexing tra gli applicativi, ha il *sequence number* del pacchetto ed un *numero di acknowledgement* per piggybacking dei pacchetti che viaggiano nella direzione opposta, ha il campo *window* che indica quanti byte possono essere ancora ricevuti senza causare intasamento su questo nodo, ha una *checksum* che serve per controllare l'intregrità del pacchetto, ha il campo *Data Offset* il quale specifica la lunghezza del segmento, il campo *RES* è riservato all'uso futuro, il campo *Control* è formato da sei bit denominati bit di codice per determinare lo scopo e il contenuto del segmento, il campo *checksum* contiene una checksum che server a verificare l'integrità dei dati come pure l'intestazione del TCP, l'*Urgent Pointer* indica che nel pacchetto ci sono uno o più byte urgenti, dove tipicamente i byte urgenti sono dei byte associati a degli eventi asincroni e per poter gestire questi byte (detti anche dati fuori banda) il TCP consente al trasmettitore di specificare i dati come "urgenti" nel senso che il programma ricevente deve essere avvisato del loro arrivo il più rapidamente possibile e indipendentemente dalla loro posizione nello stream, infine abbiamo il campo *Options* il quale è impiegato dal software del TCP per negoziare col software del TCP all'altra estremità la dimensione massima del segmento che è disposto ad ricevere (il campo *Padding* è un campo di riempimento).

Ora illustriamo l'interfaccia tra i programmi applicativi ed il software del TCP/IP attraverso le seguenti cinque caratteristiche sotto riportate:

- **Orientamento allo stream**

  Quando due programmi applicativi (processi di utente) trasferiscono grandi volumi di dati, tali dati sono immaginati come stream (sequenze, o serie) di bit, suddivisi in byte. Il servizio di consegna dello stream nella macchina di destinazione passa al ricevitore esattamente la medesima sequenza di byte che il trasmettitore ha passato al suddetto servizio nella macchina di provenienza, abbiamo quindi un ordinamento sequenziale dei dati trasmessi.

- **Connessione di circuito virtuale**

  Un servizio orientato alla connessione richiede tre passi:

  ◊ attivazione della connessione
  ◊ trasferimento dei dati
  ◊ terminazione della connessione

  Inizialmente i due programmi applicativi stabiliscono un'interconnessione logica cioè prima di poter iniziare il trasferimento, i programmi applicativi trasmittente e ricevente devono interagire coi rispettivi sistemi operativi, informandoli del desiderio di un trasferimento di stream.

  I moduli software del protocollo nei due sistemi operativi comunicano inviando dei messaggi attraverso la rete, verificando che il trasferimento è autorizzato e che entrambe le parti sono pronte.

  Una volta che tutti i dettagli sono stati definiti, i moduli del protocollo informano i programmi applicativi che una connessione è stata stabilita e che il trasferimento può iniziare.

  Durante il trasferimento, il software del protocollo nelle due macchine mantiene la comunicazione per verificare che i dati siano ricevuti correttamente.

  Se la comunicazione fallisse per un qualsiasi motivo (ad esempio, per un guasto nell'hardware della rete lungo il percorso fra le due macchine), entrambe le macchine rivelerebbero il fallimento e ne informerebbero gli appropriati programmi applicativi.

  Infine due programmi che impiegano il TCP per comunicare possono terminare la conversazione chiudendo la connessione; al suo interno il protocollo impiega un meccanismo con riscontri per chiudere le connessioni.

- **Trasferimento bufferizzato**

  I programmi applicativi inviano uno stream di dati attraverso un circuito virtuale passando ripetutamente byte di dati al software del protocollo.

  Quando si trasmettono i dati, ogni applicazione impiega qualsiasi dato che ritenga conveniente, che può essere anche un solo byte. All'estremità ricevente, il software del protocollo consegna i byte allo stream di dati esattamente nel medesimo ordine in cui sono stati inviati, rendendoli disponibili al programma applicativo ricevente non appena sono stati ricevuti e verificati.

  Il software del protocollo è libero di suddividere lo stream in pacchetti indipendentemente dal programma applicativo.

  Per rendere più efficiente il trasferimento e per minimizzare il traffico nella rete, di solito le implementazioni raccolgono abbastanza dati da uno stream da riempire un datagram ragionevolmente grande prima di trasmetterlo attraverso la rete. Quindi anche se il programma applicativo genera lo stream un byte alla volta, il trasferimento attraverso la rete può essere piuttosto efficiente.

  Similmente, se il programma applicativo sceglie di generare blocchi di dati estremamente grandi, il software del protocollo può scegliere di suddividere ciascun blocco in pezzi più piccoli per la trasmissione.

  Questo trasferimento bufferizzato garantisce anche che il ricevitore non sia sopraffatto dal trasmettitore, come accadrebbe se questo inviasse i dati più velocemente di quanto il ricevitore non possa elaborarli; in questo caso il protocollo esegue il controllo del flusso sui dati trasmessi sulla rete.

- **Stream non strutturata**
  Il servizio di stream del TCP non onora lo stream di dati strutturati, per esempio non c'è alcun modo in cui un'applicazione della contabilità degli stipendi possa richiedere al servizio di stream di contrassegnare i confini tra i record degli impiegati.
  I programmi applicativi che utilizzano il servizio di stream devono comprendere il contenuto dello stream e concordare sul formato di questo prima di avviare una connessione.
- **Connessione full-duplex**
  Le connessioni fornite dal servizio di stream del TCP consentono il trasferimento simultaneo in entrambe le direzioni. Tali connessioni sono dette full-duplex.
  Dal punto di vista di un processo applicativo, una connessone full-duplex consiste di due stream indipendenti che fluiscono in direzioni opposte, senza alcuna apparente interazione.
  Il servizio di stream consente ad un processo applicativo di terminare il flusso in una direzione mentre i dati continuano a fluire nella direzione opposta, rendendo half-duplex la connessione.
  Il vantaggio di una connessione full-duplex è che il sottostante software del protocollo può inviare in risposta alla provenienza le informazioni di controllo per una stream in datagram che trasportano i dati nella direzione opposta. Tale operazione, nota come "*piggy-backing*" contribuisce a ridurre il traffico nella rete.

1.4.1 Porte, connessioni e punti terminali

Il TCP consente la comunicazione simultanea di più programmi applicativi nella stessa macchina e distribuisce fra i vari programmi applicativi il traffico TCP in arrivo; questo è reso possibile poiché il TCP impiega numeri di porta di protocollo per identificare la destinazione finale all'interno di una macchina.

Quando un processo client deve contattare un processo server, il client per identificare il server desiderato usa l'indirizzo IP (associato univocamente ad ogni host) e un numero di porta.

Le porte, in sostanza, sono il mezzo con cui un programma client su un elaboratore indirizza un programma server su un altro elaboratore.

Per chiarire maggiormente questo concetto possiamo anche dire che il TCP è stato costruito sull'astrazione della connessione, in cui gli oggetti da identificare non sono più processi su una macchina remota ma connessioni virtuali di circuiti.

Le connessioni sono identificate da una coppia di punti terminali (endpoint) dove un punto terminale è una coppia di numeri (host, porta), in cui host è l'indirizzo IP di un host, mentre porta è una porta TCP in quell'host.

Per esempio, il punto terminale (192.84.155.36, 25) specifica la porta TCP 25 (destinata per la posta elettronica) nella macchina della Scuola Normale Superiore il cui indirizzo IP è 192.84.155.36.

Poiché una connessione è definita dai suoi due punti terminali, se esiste una connessione dalla macchina 192.84.155.36 alla macchina 131.114.2.42 (CNUCE-CNR di Pisa), essa potrebbe essere definita dai punti terminali

(192.84.155.36, 25) e (131.114.2.42, 1069)

nel frattempo, un'altra connessione di e-mail potrebbe essere in corso da una macchina del dipartimento di scienze dell'informazione al (192.84.155.36) e tale connessione potrebbe essere identificata dai punti terminali

(192.84.155.36, 25) e (131.114.4.36, 1085)

Questa condizione è resa possibile poiché il TCP identifica una connessione mediante una coppia di punti terminali e quindi un certo numero di porta può essere condiviso da più connessioni nella medesima macchina.

Per gli applicativi principali (Telnet, ftp, e-mail, ect) esiste una assegnazione standard ufficiale che prende il nome di *well known port* in base alla quale dei numeri di porta ben noti hanno una corrispondenza tra un certo numero e un certo applicativo.

Nella rete Internet le porte 0-1023 sono porte riservate, le restanti possono essere utilizzate per lo sviluppo di applicativi non standardizzati.

1.4.2 Aperture passive e attive

Poiché il TCP è un protocollo orientato alla connessione, prima che il traffico di dati possa attraversare un'internet, i programmi applicativi alle due estremità della connessione devono concordare sul fatto che tale connessione è desiderata.

A tal fine, il programma applicativo a un'estremità (il server) esegue una funzionalità di apertura passiva (sul socket passivo) contattando il suo sistema operativo ed indicando che accetterà una connessione in arrivo. In quell'istante, il sistema operativo assegna un numero di porta TCP alla sua estremità della connessione.

Il programma applicativo all'altra estremità (il client) deve allora contattare il suo sistema operativo utilizzando una richiesta di apertura attiva (sul socket attivo) per stabilire una connessione.

I due moduli software del TCP comunicano per stabilire e verificare una connessione, una volta che una connessione e stata creata, i programmi applicativi possono cominciare a scambiarsi i dati.

Ora descriviamo, utilizzando un linguaggio naturale, gli algoritmi utilizzati dai vari tipi di server e di client. Poiché questo manuale descrivere soltanto le interfacce sockets del protocollo di comunicazione TCP la descrizione dei vari tipi di servers viene fatta solo per servers connection-oriented.

1.5 Algoritmo di un server iterativo connection-oriented

Nel caso di un server iterativo connection-oriented abbiamo l'algoritmo schematizzato nella figura 3:

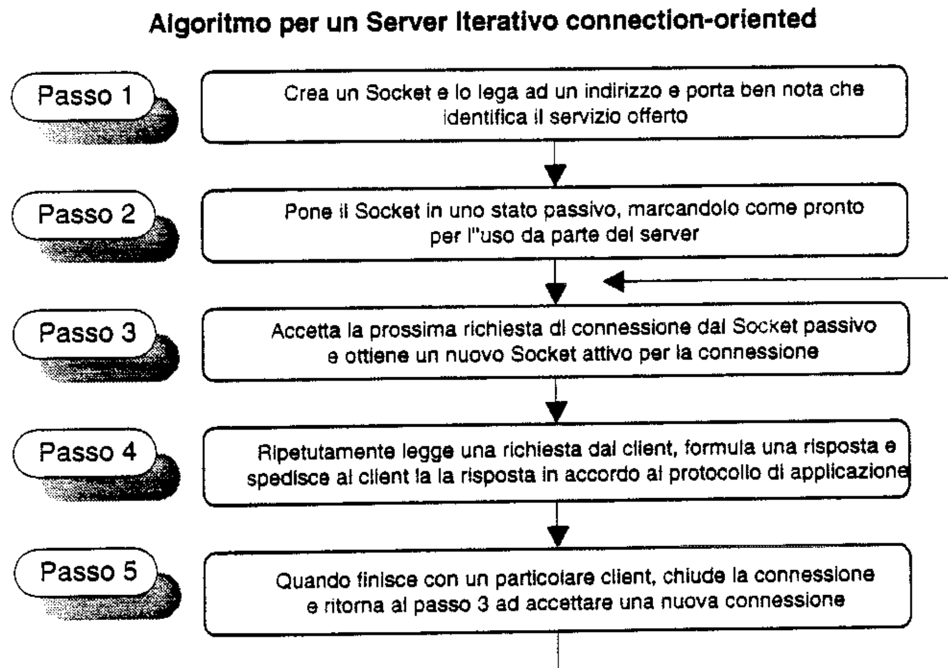


Figura 3 Algoritmo di un Server Iterativo su un protocollo connection-oriented

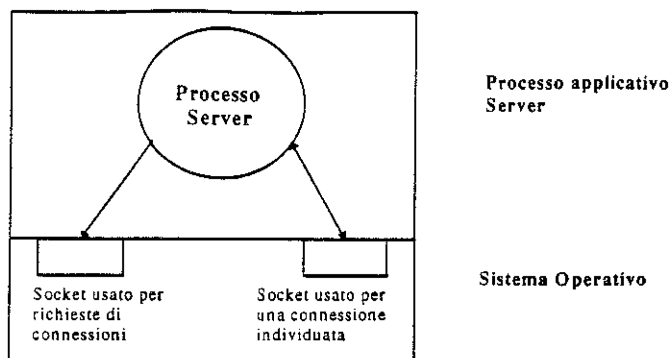


Figura 4 Schema dei processi e sockets coinvolti in un server iterativo connection-oriented

In un server iterativo connection-oriented è un singolo processo che gestisce una alla volta le connessioni dei clients, il processo itera sempre sulle connessioni usando un socket per gestire le richieste entranti (socket passivo) ed un secondo socket temporaneo (socket attivo) per gestire la comunicazione con un client, cioè aspetta su un indirizzo e porta ben nota la prossima connessione che arriva da un client, accetta la connessione, la gestisce, chiude la connessione, e poi aspetta di nuovo.

1.5 Algoritmo di un server concorrente connection-oriented

Per quanto riguarda l'algoritmo di un server concorrente connection-oriented abbiamo che il processo master accetta le connessioni entranti e crea un processo slave per gestire ciascuna di esse, una volta che lo slave ha finito sarà lui a chiudere la connessione.

Come mostrato dalla figura 5, il processo master server non comunica direttamente con i clients ma semplicemente aspetta ad un indirizzo e porta ben nota la richiesta della prossima connessione.

Quando arriva una richiesta di connessione sulla chiamata `accept`, il sistema restituirà il descrittore di socket del nuovo socket da usare per quella connessione quindi il master creerà un processo slave per gestire la connessione e permetterà allo slave di operare concorrentemente, dopodiché tornerà sulla `accept` per aspettare una nuova richiesta.

Per garantire l'utilizzo della risorsa CPU il processo master aspetta le connessioni utilizzando una chiamata `accept` bloccante , questo significa che in un server concorrente il processo master (ma questo avveniva anche nel caso di server iterativo) spende gran parte del suo tempo bloccato sulla chiamata `accept`.

Ciascun processo slave dopo aver ricevuto dal processo master il socket per la connessione individuata (nel caso del sistema operativo Unix i processi slaves erediteranno dal loro processo padre i sockets attivi poiché questi ultimi sono considerati come descrittori di file e sappiamo che nella relazione padre figlio quest'ultimo eredita dal padre tutti i descrittori attivi) interagisce con il client su quel socket, svolge il servizio richiesto, restituisce i risultati, chiude la connessione e alla fine termina.

In ogni momento il server consiste di un processo master e di zero o più processi slave.

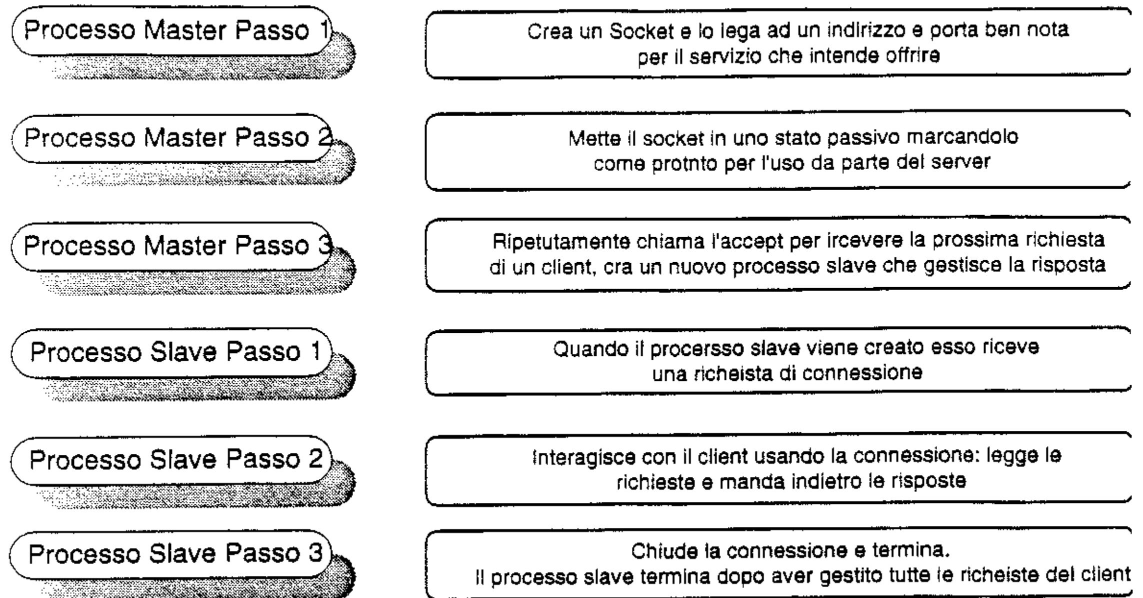


Figura 5 Algoritmo di un Server Concorrente su un protocollo connection-oriented

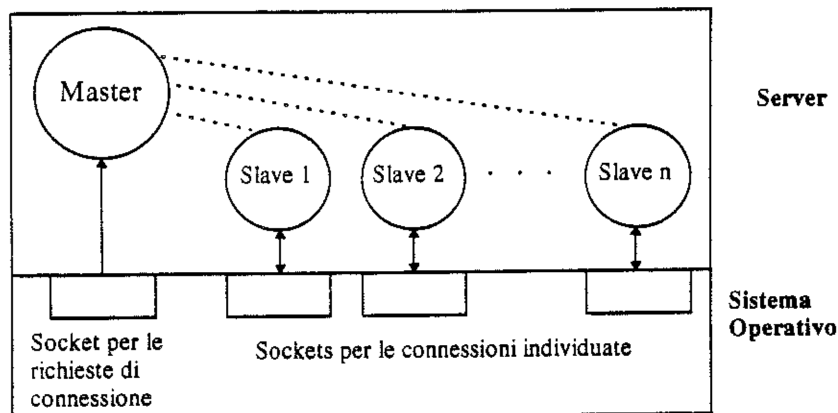


Figura 6 Schema dei processi e dei sockets coinvolti in un server concorrente connection-oriented

La ragione principale per introdurre la concorrenza in un server, sorge dal bisogno di fornire tempi di risposta migliori a clients multipli.

Una soluzione con server concorrente migliora i tempi di risposta se:

i) formulare una risposta da parte del server richiede un input-output significativo
ii) i tempi richiesti per elaborare le richieste variano enormemente
iii) se il server viene eseguito su un computer multiprocessor.

Nel primo caso, permettere al server di elaborare le richieste concorrentemente significa che esso può sovrapporre l'uso del processor e dei dispositivi periferici, questo anche se la macchina ha un sola CPU.

Mentre il processor lavora per elaborare una risposta, i dispositivi di I/O possono trasferire da o verso la memoria dati di cui il server ha bisogno per elaborare le altre richieste.

Nel secondo caso, la divisione di tempo (che è alla base dei sistemi multitasking) permette a un singolo processor di gestire le richieste che richiedono solo una piccola quantità di elaborazione senza aspettare che le richieste che richiedono più tempo finiscano.

Nel terzo caso, l'esecuzione concorrente su un computer multiprocessor permette ad un processor di elaborare una richiesta di un client mentre un altro processor elabora la richiesta di un altro client.

I servers concorrenti sono generalmente progettati in modo che le loro performance migliorino automaticamente se essi girano su un hardware che offre più risorse di elaborazione.

1.7 Algoritmo di un Server Apparentemente Concorrente connection-oriented

In alcuni casi, comunque, ha senso usare un singolo processo per gestire concorrentemente richieste provenienti da più clients.

Uno di questi casi è quando il sistema operativo (esempio Windows 3.11) non è multitasking preemptive e non offre, attraverso chiamate di sistema, la creazione dinamica di processi; questa mancanza influisce pesantemente sul progetto di servers in ambiente Windows poiché esclude a priori la possibilità di creare server concorrenti.

Altri casi possono trovarsi in quei sistemi operativi in cui la creazione di processi comporta un overhead così grande che un server non può supportare la creazione di un nuovo processo per ciascuna richiesta o ciascuna connessione.

Ma ancora più importanti sono quei casi in cui molte applicazioni clients richiedono al server la condivisione di informazioni attraverso tutte le connessioni.

Poiché i dati che arrivano da tutti i clients necessitano di informazioni residenti in una singola struttura dati condivisa, e poiché Unix BSD non permette a processi indipendenti la condivisione della memoria (Unix non è share-memory), il server non può eseguirsi con processi separati.

In Unix c'è un conflitto tra processi concorrenti che condividono la memoria e la mancanza di un supporto per la gestione di tale concorrenza.

Se non è possibile ottenere reale concorrenza tra processi che condividono la memoria, è invece possibile ottenere concorrenza apparente se il numero di richieste presenti nel server non eccede la sua capacità di gestione.

Per fare questo il server opera come un singolo processo e usa una chiamata che permette di realizzare l'I/O asincrono (Unix mette a disposizione la chiamata `select`).

Ora descriviamo come un singolo processo possa offrire apparente concorrenza a clients multipli usando un singolo processo e nel fare questo per prima cosa vediamo quando un simile approccio è realizzabile e quando è superiore ad una implementazione che usa processi multipli; per seconda cosa vediamo come in Unix un singolo processo usa le chiamate di sistema per gestire connessioni multiple concorrentemente.

Per applicazioni dove l'I/O è dominante sul costo di preparazione e risposta ad una richiesta, un server può usare l'I/O asincrono per fornire concorrenza apparente a più clients. L'idea è quella di permettere ad un singolo processo server di tenere aperte connessioni TCP per clients multipli, e avere un server che gestisce una data connessione quando i dati arrivano.

Quindi il server usa l'arrivo dei dati come attivatore di elaborazione di richieste dei clients, ossia si basa sul modello data-driven processing.

Per capire come questo approccio lavora, consideriamo il server ECHO concorrente.

Per attivare un'esecuzione concorrente, il server crea un processo slave separato per gestire ciascuna nuova connessione.

In teoria, il server dipende dal meccanismo di divisione di tempo per condividere la CPU del sistema operativo tra i processi e da questo momento anche dalle connessioni. In pratica, comunque, un ECHO server raramente dipende dalla divisione di tempo.

Se uno fosse capace di vedere da vicino l'esecuzione di un ECHO server concorrente, scoprirebbe che c'è un costante controllo sull'arrivo dei dati con relativa esecuzione del server stesso.

La ragione è in relazione al flusso di dati attraverso Internet.

I dati arrivano al server in modo "*bursts*" cioè non costantemente e non in modo continuo (non in modo "*steady stream*" cioè come flusso continuo e costante) poiché la sottostante rete Internet consegna i dati in pacchetti discreti.

I clients si aggiungono a questo comportamento (*bursts*) se cercano di spedire blocchi di dati tali che ciascun corrispondente segmento TCP è adatto a un singolo IP datagram.

Nel server ciascun processo slave spende la maggior parte del suo tempo bloccato su una chiamata read aspettando il prossimo datagram in arrivo.

Una volta che i dati arrivano, la chiamata read ritorna e il processo slave esegue il servizio richiesto.

Lo slave chiama la write per mandare i dati indietro al client, e poi può chiama nuovamente la read per aspettare altri dati oppure terminare.

Una CPU che può gestire un carico di molti clients senza (slowing down) rallentare deve avere un'esecuzione sufficientemente veloce per completare il ciclo di lettura e scrittura prima che i dati arrivino ad un altro slave.

Certamente, se il carico diventa così grande che la CPU non può processare una richiesta prima che un'altra richiesta arrivi, allora il timesharing prende il controllo.

Il sistema operativo commuta il processor tra tutti gli slaves che hanno dati da processare.

Per semplici servizi che richiedono poca elaborazione per ciascuna richiesta, i cambiamenti sono così alti che l'esecuzione sarà guidata dai dati in arrivo (*demand driven*).

Capire la naturale sequenza del comportamento di un server concorrente permette di capire come un singolo processo può eseguire gli stessi compiti. Immaginiamo un singolo processo server che ha connessioni TCP aperte per molti clients, in questa situazione il processo rimarrebbe bloccato aspettando l'arrivo dei dati.

Quando i dati arrivano su una connessione, il processo si sveglia, gestisce la richiesta, e manda una risposta. Esso poi si blocca nuovamente, aspettando che altri dati arrivino su altre connessioni

Quando la CPU è così veloce che è in grado di soddisfare il carico presente sul server, la versione con un singolo processo gestisce le richieste meglio della versione con processi multipli.

Infatti, poiché l'implementazione con un singolo processo richiede meno commutazione di contesto tra i processi, essa può essere capace di gestire un carico leggermente più alto di una implementazione che usa processi multipli.

La chiave per programmare un server costituito da un singolo processo risiede nell'uso di meccanismi di I/O asincroni attraverso primitive che il sistema operativo mette a disposizione dell'utente.

Un server crea un socket per ciascuna delle connessioni che esso deve gestire, e poi chiama la select per aspettare che i dati arrivino da ciascuno di loro, infatti, poiché la `select` può aspettare per l'I/O su tutti i possibili sockets, esso può allo stesso tempo aspettare nuove connessioni.

I passi che esegue un server apparentemente concorrente su un protocollo connection-oriented sono i seguenti:

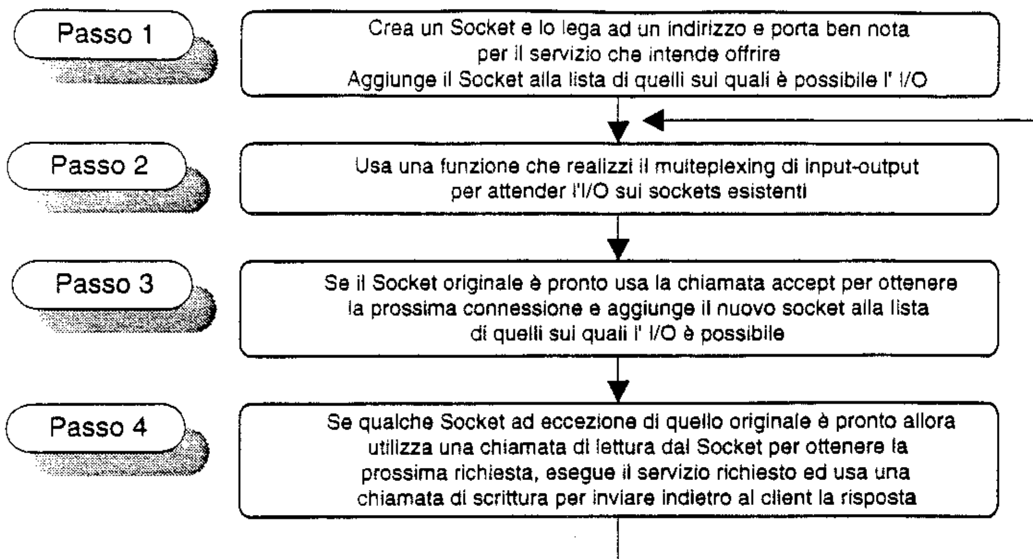


Figura 7 Algoritmo di un Server Apparentemente Concorrente su un protocollo connection-oriented

In questo caso un singolo processo server gestisce tutti i sockets cioè quello passivo e quelli attivi.

In sostanza, un singolo processo server deve svolgere le mansioni di entrambi i processi master e slaves che avevamo nel caso di server concorrente.

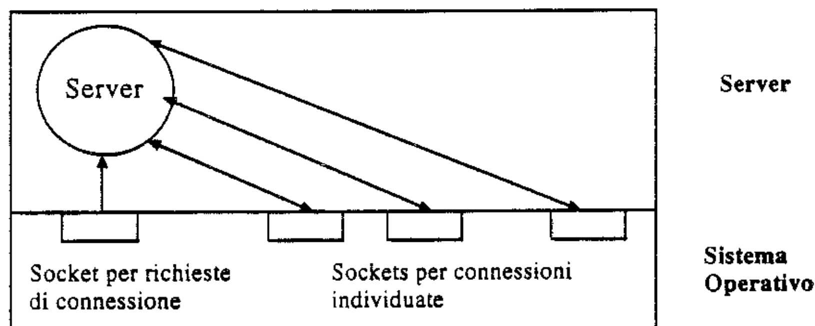


Figura 8 Schema dei processi e dei sockets coinvolti in un server apparentemente concorrente connection-oriented

Esso mantiene un certo insieme di sockets di cui uno di questo insieme (socket passivo) è legato ad una porta ben nota sulla quale il master accetta le connessioni, mentre tutti gli altri socket sono attivi e ciascuno di loro corrisponde ad una connessione sulla quale uno slave gestisce le richieste.

In Unix attraverso la chiamata `select` o la `poll` è possibile ordinare al kernel di attendere uno qualsiasi di vari eventi (ad esempio la presenza di dati da leggere sul socket) e di risvegliare il processo solo quando si verifica uno di tali eventi.

1.8 Caratteristiche di un server connection-oriented

Nella progettazione ed implementazione di servers concorrenti e apparentemente concorrenti bisogna tenere presente un importante parametro che è il livello di concorrenza.

1.8.1 Livello di concorrenza dei server

La definizione di livello di concorrenza differisce per i due tipi di server poiché nel caso di server concorrente rappresenta il numero totale di processi slaves presenti in un certo istante, mentre nel caso di server apparentemente concorrente rappresenta il numero totale di sockets attivi.

Infatti nel primo caso il server crea un nuovo processo slave per ogni nuova richiesta in arrivo e ciascun slaves termina quando ha completato la comunicazione; invece nell'altro caso quando arriva una nuova richiesta il server alloca un nuovo socket e lo aggiunge alla lista dei socket attivi, quando poi la comunicazione ha avuto fine disalloca il socket.

Chiaramente il livello di concorrenza varia nel tempo e dipende in ogni momento sia dal numero delle richieste che giungono dai clients sia dal numero di connessioni terminate.

Poiché abbiamo dei limiti reali sia nel TCP per quanto riguarda il numero delle possibili connessioni attive sia nel sistema operativo per quanto riguarda il numero dei processi ed il numero delle risorse disponibili, durante l'implementazione di un server è opportuno specificare il massimo livello di concorrenza ossia il numero massimo di processi attivi o il numero massimo di socket attivi che il server può gestire.

Nel caso di una implementazione che prevede un massimo livello di concorrenza, quando il server raggiunge questo valore massimo e soggiunge una nuova richiesta di connessione questa sarà semplicemente scartata.

Il criterio di scelta del massimo livello di concorrenza dipende fondamentalmente dal sistema operativo e dalle risorse di cui dispone la macchina. Nel caso di server iterativo il massimo livello di concorrenza è sempre 1.

Per incrementare il livello di concorrenza la maggior parte dei server concorrenti e apparentemente concorrenti utilizza lo schema *demand-driven* (guidato dalla domanda) ossia vengono usate le richieste in arrivo per incrementare tale livello; servers che adottano questa soluzione hanno il vantaggio che non usano risorse di sistema quando non ne hanno bisogno. Nel caso di server concorrente esiste un'alternativa allo schema demand-driven ed è la preallocazione dei processi slaves.

Il progettista di un server che utilizza questo schema deve consentire al processo master la creazione di n processi slave appena la sua esecuzione ha inizio.

Ogni processo slave usa poi le funzioni disponibili nel sistema per aspettare le richieste in arrivo, quando poi arriva una nuova richiesta, uno dei processi slave in attesa inizia l'esecuzione e gestisce la richiesta, quando finisce di gestire la richiesta lo slave non termina ma invece ritorna ad aspettare un'altra richiesta.

La preallocazione dei processi slaves può abbassare sensibilmente i tempi di risposta del server poiché azzera i tempi necessari per la creazione di un processo per ogni richiesta in arrivo e permette al sistema di intercalare le attività di I/O di una richiesta con quelle di un'altra.

Per questo tipo di server concorrente abbiamo la struttura processi-socket mostrata nella figura 9.

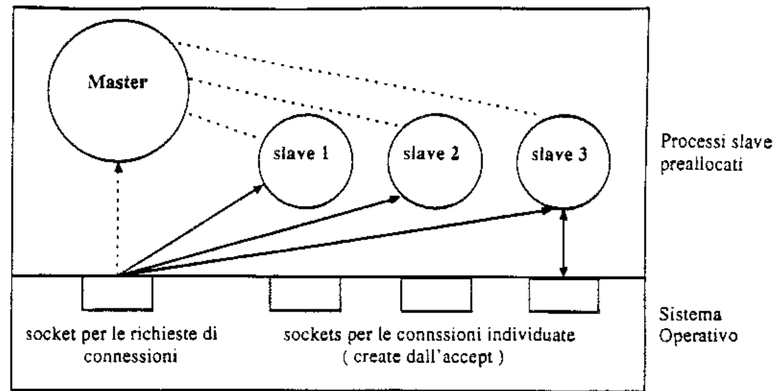


Figura 9 Struttura dei processi e dei socket coinvolti in un server concorrente connection-oriented con preallocazione dei processi

1.8.2 Deadlock dei Servers

Molte implementazioni di servers sono esposte ad un evento indesiderato che è quello del deadlook. Per capire come lo stallo può manifestarsi, consideriamo un server iterativo connection oriented.

Supponiamo una applicazione client, C, che si comporta male (male intenzionata).

Nel caso più semplice, assumiamo che C crei una connessione su un server, ma mai manda una richiesta. Il server accetterà la richiesta di connessione, ed effettuerà una chiamata di lettura per estrarre dal buffer di comunicazione la richiesta del client C.

Il processo server si bloccherà sulla chiamata di lettura aspettando una richiesta che non arriverà mai. Il deadlook del server può sorgere anche nel caso in cui i clients non consumano le risposte.

Per esempio, assumiamo che un client C stabilisca una connessione con un server, manda ad esso una sequenza di richieste , ma mai legge le risposte. Il server una volta che ha accettato le richieste ed elaborato le risposte, tenterà di spedirle al client il quale non sarà disposto ad accettarle con conseguente deadlook del server.

Possiamo anche verificare questa situazione di stallo del server esaminando il comportamento del protocollo TCP.

Infatti, in questa situazione, il software del protocollo TCP trasmette i primi byte sulla connessione che ha stabilito col client, però appena il TCP riempirà la finestra di ricezione del client fermerà la trasmissione dei dati.

Se il programma applicativo del server nel frattempo continua a generare risposte, il buffer locale TCP usato per memorizzare i dati in partenza sulla connessione diventerà pieno e il processo server si bloccherà.

In sostanza il deadlook si verifica perché il processo server si blocca quando il sistema operativo non può soddisfare una chiamata di sistema.

In particolare, una chiamata di invio bloccherà il processo che l'ha chiamata se il TCP non ha nessun spazio nel buffer locale per i dati che intende spedire; una chiamata di ricezione bloccherà il processo fintantoché il TCP riceve i dati.

Per servers concorrenti, lo stallo si verificherà solo su un singolo processo slave associato ad un particolare client se quest'ultimo fallisce nel mandare richieste o leggere risposte.

Per implementazioni di servers con un singolo processo (iterativi e parzialmente concorrenti), sarà lo stesso processo centrale ad essere bloccato, quindi non si potranno gestire altre connessioni con conseguente blocco dell'intero sistema.

Quindi un comportamento male intenzionato da parte di un client può causare deadlook su un singolo processo server se il server usa funzioni di sistema che possono bloccarsi quando comunicano con il client.

Nella implementazione di un server è quindi importante usare chiamate di sistema, per l'invio e la ricezione dei dati, non bloccanti.

Oltre alla strategia utilizzata dai servers per gestire più richieste simultaneamente, dobbiamo considerare almeno altri due parametri importanti nella progettazione di un server:
- la sicurezza e
- l'affidabilità.

1.8.3 Sicurezza dei server

In generale i servers devono realizzare o rafforzare le strategie di accesso e di protezione al servizio che intendono offrire.

Infatti, nel caso di sistemi operativi che prevedono due modalità operative da user e da superuser, (esempio Unix) molto spesso i servers sono eseguiti col più alto privilegio perchè devono ad esempio leggere files, tenere registrazioni, ed accedere a basi di dati protette o non protette, allora in tali circostanze devono farsi carico: dell'autenticazione cioè verificare l'identità del client, dell'autorizzazione cioè determinare se un dato client ha il permesso di accesso al servizio che il server fornisce, della protezione dei dati (data security nelle basi di dati) cioè garantire che il dato nella base di dati non sia involontariamente rivelato o compromesso.

La stessa situazione si riscontra nel caso di sistemi operativi che non prevedono modalità operative privilegiate (esempio Windows), infatti i servers che vogliano offrire un servizio sicuro devono farsi carico di adottare strategie di protezione e di autenticazione.

1.8.4 Affidabilità dei server

I servers in generale, oltre ad evitare lo stallo, dovendosi comportare come dei demoni (cioè come processi che si eseguono in background e che vivono da quando il sistema è boostrapped a quando il sistema è shutdown) devono proteggere se stessi sia dalle le richieste formulate scorrettamente, le quali potrebbero causare l'aborto del programma, sia dai possibili errori di rete (errori di trasmissione o errori di caduta della rete) che da errori locali (elaborazione del servizio richiesto).

Devono in sostanza assicurare: una attenta rilevazione di ogni tipo di errore ed un loro confinamento, evitare deadlook, predisporre il ripristino di uno stato consistente nel caso in cui un errore si manifestasse; devono essere in sostanza dei demoni robusti.

1.9 Algoritmo di un client iterativo connection-oriented

Le applicazioni che agiscono come client iterativo sono concettualmente le più semplici poiché gestiscono esclusivamente le interazioni con un unico server e non richiedono particolari controlli sull'andamento della comunicazione da parte dell'utente.

Se alle semplici funzionalità richieste da un processo client aggiungiamo un protocollo come il TCP il quale gestisce tutti i problemi di affidabilità e controllo del flusso, allora implementare un client che usa il TCP è molto semplice.

I passi che esegue un client iterativo su un protocollo connection-oriented sono schematicamente riassunti nella sottostante figura 10.

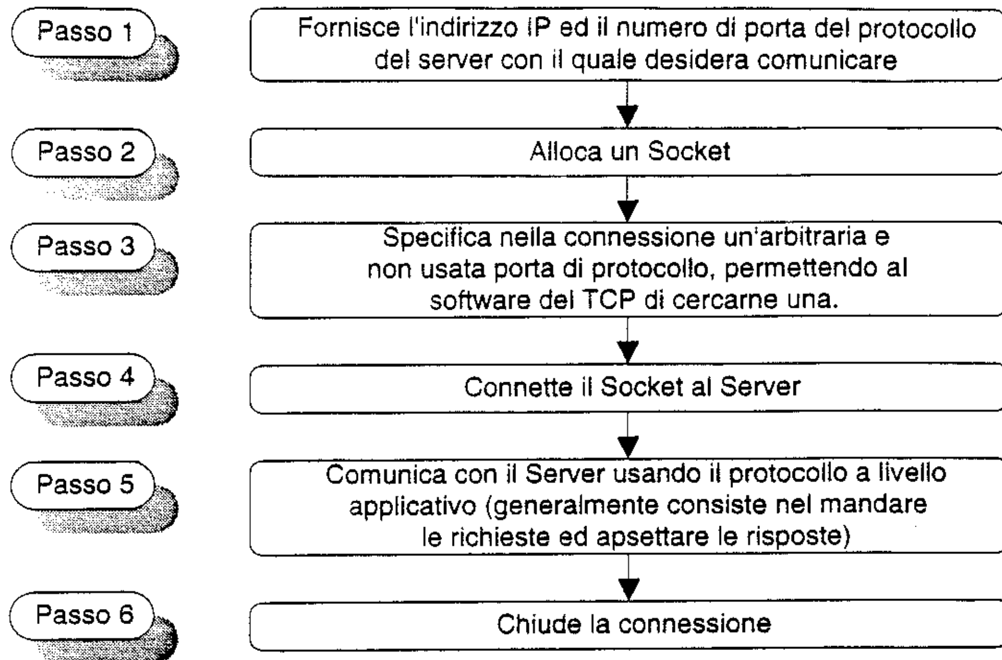


Figura 10 Algoritmo di un Client Iterativo su un protocollo connection-oriented

In un client iterativo connection-oriented abbiamo una struttura processi-socket molto semplificata con un singolo processo che gestisce un singolo socket usato per la connessione che si intendere instaurare con un unico server.

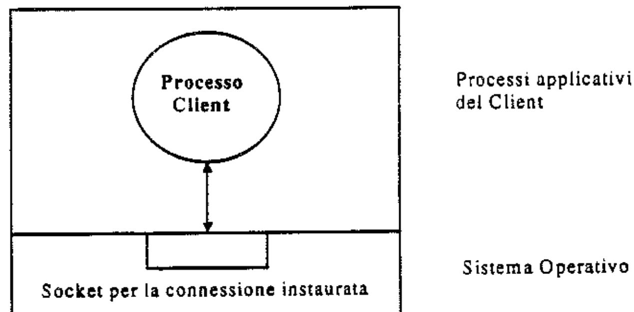


Figura 11 Struttura del processo e dei socket coinvolti in un client iterativo connection-oriented

Una soluzione con un unico processo client è adatta in tutte quelle applicazioni di rete in cui si vuole contattare un unico server per avere un determinato servizio; un classico esempio di questo tipo è quello del servizio offerto da un server su un sistema di basi di dati centralizzato in cui i dati sono mantenuti in un'unica copia e nella stessa macchina. Questo tipo di client è stato adottato per recuperare informazioni da database CDS/ISIS.

Se alla possibilità di contattare contemporaneamente più servers aggiungiamo anche la necessità di avere funzioni di controllo (sullo stato dell'esecuzione e della comunicazione) separate dalla normale elaborazione del client, come nel caso dei server, possiamo definire due tipi di client:

- client concorrente
- client che utilizza I/O asincrono

Nella trattazione dei servers abbiamo appurato che essi usano la concorrenza per due ragioni principali, la prima è che la concorrenza permette di migliorare i tempi di risposta (e quindi il throughput globale di tutti i clients), la seconda è che la concorrenza può eliminare potenziali deadlook.

In aggiunta, una implementazione di server concorrente permette ai progettisti di creare facilmente server multiprotocollo e multiservizio.

Infine, implementazioni concorrenti che usano processi multipli sono estremamente flessibili perchè esse operano bene su una varietà di piattaforme hardware (singolo processor e multiprocessor).

A prima vista sembrerebbe che i clients non possano beneficiare della concorrenza, principalmente perchè un client generalmente esegue una attività alla volta, cioè una volta che ha mandato una richiesta al server il client non può procedere fintantoché non ha ricevuto una risposta.

Inoltre, le analisi sull'efficienza ed il problema di deadlook sono meno importanti perché se un client è lento oppure cessa la sua attività allora interrompe solo se stesso e non compromette l'esecuzione di altri clients.

Malgrado l'apparenza, esistono applicazioni dove la concorrenza nel client diventa importante.

Un esempio di questo tipo si ha quando un client deve contattare più servers, infatti in questa situazione una implementazione concorrente del client permette di contattare allo stesso tempo (concorrentemente) più server per combinare i risultati che i servers restituiscono oppure per paragonare i tempi di risposta.

Altro esempio si ha quando un client manda una query ad un server che gestisce una grossa base di dati, infatti in questo caso il server potrebbe impiegare molti minuti prima di restituire i risultati e nel caso di un client iterativo questo semplicemente aspetterebbe fintantoché i risultati arrivano.

In questa situazione, un malfunzionamento del server dovuto ad uno stallo o ad errori interni avrebbe come conseguenza il blocco del client il quale attende le risposte che non arriveranno mai.

Inoltre il client non può conoscere se nel server si è verificato un stallo o un overload, se il ritardo è dovuto alla lentezza della rete sottostante oppure, nel caso in cui il server spedisse i risultati della query, non potrebbe sapere se i risultati sono stati tutti spediti.

Nel caso di un client iterativo, se una particolare risposta richiede troppo tempo e l'utente è impaziente allora l'unica cosa che quest'ultimo può fare è uccidere il processo e ritentare la query di nuovo dopo un certo tempo.

In tale situazione, la concorrenza può essere di aiuto perché un appropriato progetto di un client concorrente permette all'utente di continuare ad interagire con il client mentre lo stesso aspetta una risposta, cioè l'utente può scoprire se tutti i dati sono stati ricevuti, cercare di mandare una differente richiesta, oppure terminare la comunicazione gradualmente (*gracefully*).

Come per i server concorrenti, le maggiori implementazioni di clients concorrenti seguono due approcci base: quello concorrente in cui il client è diviso in due o più processi e ciascun processo gestisce sia funzionalità di input-output con l'utente sia la connessioni con il server; e quello concorrente asincrono in cui il client consiste di un singolo processo che usa una funzione di multiplexing per gestire eventi asincroni provenienti dall'utente e dalle connessioni con i servers.

1.9.1 Client Concorrente

In sistemi operativi (no Unix e Windows) che supportano uno schema di comunicazione basato sulla memoria condivisa, un'implementazione di client concorrente connection-oriented con processi multipli permette al client di separare l'elaborazione dell'input e dell'output tra l'utente e i servers.

Una possibile struttura processi-sockets-descrittori I/O per un client che usa un protocollo di comunicazione connection-oriented è rappresentata nella figura sottostante. Un processo di input legge dallo standard input le richieste formulate dall'utente e le manda al server sulla connessione TCP, tutto questo mentre un separato processo output riceve le risposte da un server e le scrive sullo standard output.

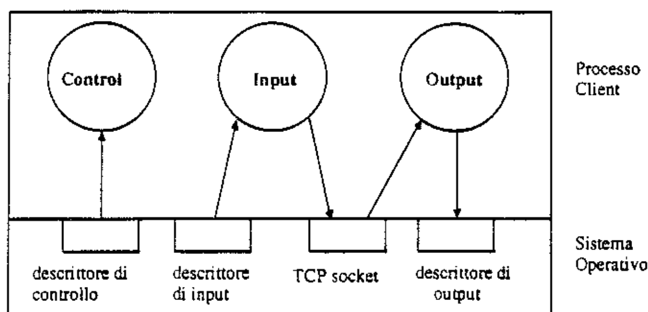


Figura 12 Struttura processi socket descrittori di I/O per un client concorrente connection-oriented

Nel frattempo un terzo processo di controllo accetta alcuni comandi dall'utente i quali controllano l'esecuzione del client.

1.9.2 Client che utilizza l'I/O asincrono

Poiché la maggior parte delle implementazioni Unix oggi esistenti non permette a processi separati la condivisione della memoria, i clients concorrenti sono progettati prevedendo un singolo processo che utilizza un meccanismo di multiplexing di input-output.

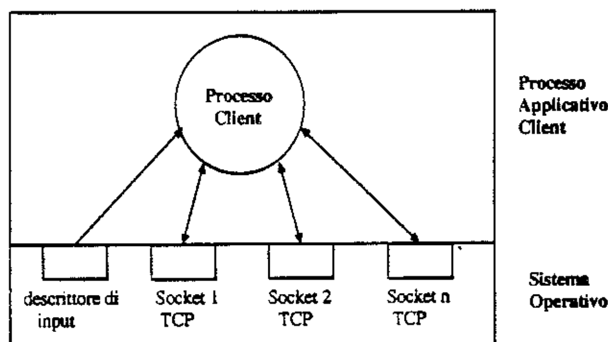


Figura 13 Struttura processi socket descrittori di input per un client che usa I/O asincrono su un protocolloconnection-oriented

Un singolo processo client usa l'I/O asincrono per gestire connessioni multiple concorrentemente. Il client crea i descrittori di sockets per le sue connessioni TCP con servers multipli. Esso può anche avere un descrittore dal quale ottenere l'input.

Il corpo del programma client consiste di un loop che usa una funzione di multiplexing per aspettare che uno dei suoi descrittori sia pronto.

Se il descrittore di input è pronto, il client legge l'input ed o lo memorizza per un uso successivo o agisce su di esso immediatamente.

Se una connessione TCP è pronta per l'output, il client prepara e spedisce una richiesta attraverso la connessione TCP; se una connessione TCP diventa pronta per l'input, il client legge la risposta che il server gli ha mandato e la gestisce.

Certamente, un client concorrente con singolo processo condivide vantaggi e svantaggi come avveniva per un server concorrente con un singolo processo.

Un vantaggio è che il client legge l'input o interagisce con il server secondo uno schema guidato da eventi; inoltre l'elaborazione locale può continuare anche se il server ritarda per un piccolo tempo, in questo modo il client continuerà a leggere ed a onorare i comandi di controllo anche se il server fallisce nel rispondere.

Uno svantaggio è che un singolo processo client può andare in deadlook se esso invoca una funzione bloccante, in tale situazione il programmatore deve essere attento nell'assicurare che il processo client non si blocchi indefinitamente aspettando un evento che mai si verificherà.

1.10 Generalità delle interfacce Sockets

L'interfaccia verso il livello di trasporto è realizzata da un insieme di system calls, funzioni e strutture dati, che normalmente viene identificata con il nome di API di networking (interfaccia di programma per le applicazioni) con i protocolli di comunicazione.

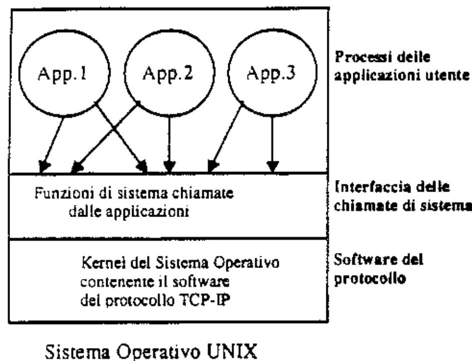


Figura 14 Interazione delle applicazioni con il software del TCP-IP nel sistema Unix

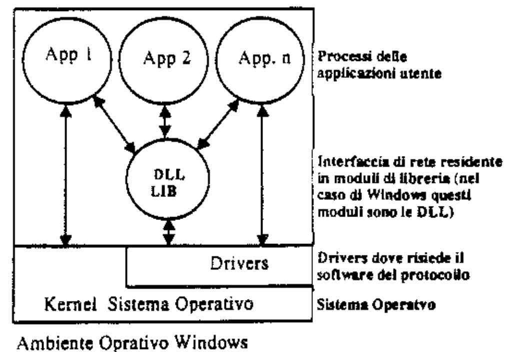


Figura 15 Interazioni delle applicazioni con il software del TCP-IP nell'ambiente Windows

Il software dei protocolli TCP-IP, nella maggior parte delle implementazioni, può risiedere o direttamente nel kernel del sistema operativo del computer (esempio Unix BSD) oppure in librerie e drivers esterni al kernel che comunque dipendono fortemente al sistema operativo per le particolari funzionalità che devono svolgere (esempi di tali interfacce sono quelle disponibili per MS-Windows e Unix system V).

Possiamo quindi affermare che, in entrambi i casi, le caratteristiche e le proprietà delle interfacce di comunicazione dipendono dal sistema operativo della macchina.

Dunque è inappropiato legare i protocolli ad una particolare interfaccia di comunicazione poiché non esiste una singola architettura d'interfaccia che funzioni bene in tutti i sistemi.

Se le interfacce dipendono dal sistema operativo è anche vero che nel caso del TCP-IP lo standard del protocollo non ha fornito accurate specifiche di come interfacciare il software applicativo con il software del protocollo.

Infatti il TCP-IP fu progettato con l'intenzione di operare tra elaboratori eterogenei (multi-vendor enviroment), quindi i progettisti del protocollo TCP-IP evitarono attentamente sia la scelta di una particolare rappresentazione interna sia di specificare l'interfaccia applicativa in termini di caratteristiche disponibili solo su un singolo sistema operativo.

Quindi, il TCP-IP non specifica i dettagli di come interfacciare il software applicativo con il software del protocollo, esso suggerisce solo le funzionalità richieste, e lascia ai progettisti di interfacce la ricercare e le scelte dei dettagli.

Avvalersi di una vaga specifica per l'interfaccia, come quella del TCP-IP, verso i protocolli di comunicazione presenta vantaggi e svantaggi.

L'aspetto positivo è la flessibilità e tolleranza dell'interfaccia, infatti questa lasca specifica ha permesso ai progettisti di implementare il TCP-IP usando sistemi operativi che variano dai semplici sistemi disponibili su personal computer ai più sofisticati sistemi per supercomputer ed inoltre (cosa molto importante) gli ha dato la possibilità di sviluppare l'interfaccia con uno schema procedurale (o memoria condivisa) oppure con uno schema message-passing (questo in base al modello che il sistema operativo supporta).

L'aspetto negativo di una lasca specifica è stato quello di aver portato alla realizzazione di interfacce con differenti dettagli per ciascun sistema operativo.

Nel momento in cui vengono aggiunte nuove interfacce che differiscono dalle interfacce esistenti , programmare applicazioni diventa più difficile e le applicazioni diventano meno portabili attraverso le macchine.

Così, mentre il progettista di interfacce è favorito da una specifica vaga, i programmatori di applicazioni desiderano una specifica ristretta poiché questo significa che le applicazioni possono essere compilate per una nuova macchina senza particolari cambiamenti.

Attualmente le due interfacce di comunicazione più utilizzate per i sistemi Unix sono i Socket di Berkeley e la TLI (Transport Layer Interface) mentre per Windows 3.11 e Windows 95 sono le Winsockets (Windows Sockets).

Le Sockets API (Application Program Interface) che prendiamo in esame in questo manuale sono le Socket BSD e le Winsockets, le prime furono sviluppate dall'università di Berkeley ed introdotte per la prima volta all'inizio degli anni 80 nel sistema Unix BSD release 3.1 per poi arrivare alla versione che prendiamo come riferimento le Sockets di BSD 4.3 la quale risale all'anno 89, le seconde furono sviluppate da un gruppo di aziende guidate dalla Microsoft a partire dall'anno 91 ed introdotte in Windows 3.11 nel 92, la versione che in questo caso prendiamo come riferimento è la Windows Sockets 1.1 del 93 .

Prendendo come riferimento il modello ISO-OSI possiamo schematicamente rappresentare le strutture di rete di Unix BSD e di Windows mettendo a confronto i livelli, il software delle interfacce il software dei protocolli, ed i protocolli stessi.

I processi utente comunicano con i protocolli di rete Internet, e quindi con altri processi su altre macchine, tramite le funzioni Socket e Winsocket che corrispondono al livello sessione di ISO-OSI, in quanto è responsabile dell'impostazione e del controllo della comunicazione.

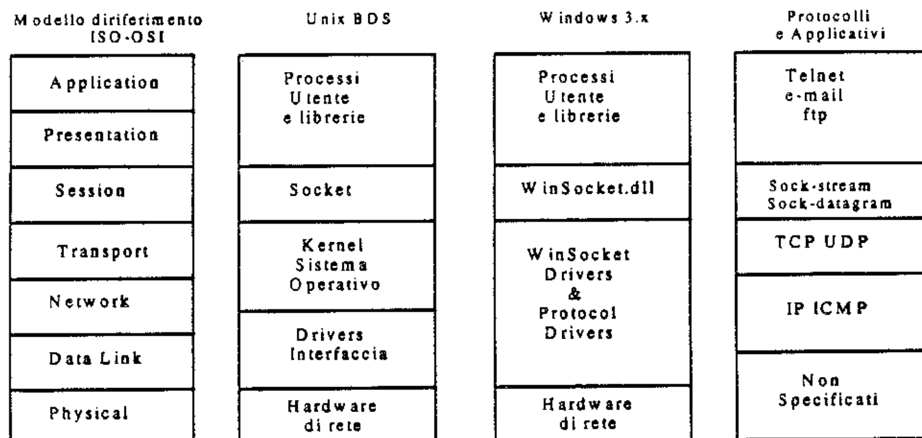


Figura 16 Relazione tra il modello ISO-OSI i protocollo TCP-IP e il software delle interfacce di comunicazione

Il software dei protocolli di comunicazione nel caso di Unix BSD risiede nel kernel del sistema operativo mentre nel caso di Windows nei drivers della WinSocket.dll.

Le DLL (librerie di collegamento dinamico) sono uno degli elementi strutturali più importanti di Windows, il loro impiego permette di fornire delle funzioni e delle risorse che possono essere condivise da tutti i programmi eseguibili.

Il termine collegamento dinamico si riferisce proprio al metodo usata da Windows per collegare una funzione in un modulo eseguibile alla funzione reale nel modulo di libreria, infatti la caratteristica più importante delle DLL è che quando si esegue il link per creare un file eseguibile dai vari moduli oggetto e librerie runtime in realtà si verifica un collegamento statico mentre il collegamento dinamico avviene al momento dell'esecuzione del modulo.

Dunque le DLL permettono di ottenere contemporaneamente due importanti risultati: completo riutilizzo del codice anche tra diversi moduli eseguibili e conseguentemente minor occupazione di memoria.

Poiché molti computers commerciali, specialmente workstations come Sun, Digital, Tektronix IBM ed altri hanno adottano il sistema operativo Unix Berkeley, le interfacce socket furono subito disponibili su molte macchine e conseguentemente sono diventate così largamente accettate da essere considerate uno standard de facto.

Anche la progettazione delle Winsocket si è basato sullo stile delle Socket BSD infatti questa interfaccia consiste di un insieme di funzioni che ricalcano il modello Socket BSD con l'aggiunta di un insieme di estensioni specifiche di Windows che permettono ai programmatori di sfruttare i vantaggi della architettura guidata da eventi (event-driven) che è propria di Windows.

I progettisti delle Socket BSD quando aggiunsero al sistema Unix i protocolli di rete decisero di fornire un'interfaccia di protocollo che consentisse ai programmatori di creare il codice del Server, che attente passivamente le connessioni, come pure il codice del Client che forma attivamente le connessioni cioè di dotare il sistema di un meccanismo di programmazione che consentisse di poter comunicare con altri processi su macchine remote attraverso una rete di comunicazione.

Un'ulteriore complessità è derivata dal fatto che i progettisti hanno voluto realizzare in un'unica interfaccia Socket un meccanismo generale che potesse gestire molte suite di protocolli di comunicazione.

Attualmente nel sistema Unix BSD 4.3 i sockets offrono ad esempio un'interfaccia a protocolli di Internet (come TCP UDP IP e ICMP), a protocolli di Xerox NS (come i protocolli SPP e IDP), a protocolli interni di Unix e a protocolli di livello di collegamento IMP.

I protocolli di Unix consentono la comunicazione tra processi residenti sulla stessa macchina, mentre i restanti vengono usati per la comunicazione tramite rete.

Nel sistema Windows i sockets offrono un'interfaccia ai soli protocolli di Internet ed in particolare solo al TCP e all' UDP.

La capacità delle interfacce Socket di poter utilizzare una certa suite di protocolli tra una famiglia di suite di protocolli viene formalizzato dicendo che i sockets dipendono dal dominio di comunicazione, dove il dominio è appunto una suite di protocolli.

Alla base di queste interfacce di comunicazione di rete c'è il concetto di Socket il quale non è altro che un punto terminale di un canale bidirezionale di comunicazione tra due processi, utilizzato per trasmettere e ricevere dati ed al quale può essere associato un nome (indirizzo IP e numero di porta).

Come per le altre forme di comunicazione fra processi, i programmi applicativi quando desiderano avere un socket richiedono al sistema operativo (o alla dll) di crearne uno ed il sistema dopo aver predisposto una certa struttura dati (area di memoria) restituisce un numero (descrittore di socket) che il programma applicativo utilizza per fare riferimento al socket appena creato.

L'applicazione può successivamente scegliere di specificare indirizzo e numero di porta locali o remoti ogni volta che utilizza il socket oppure scegliere di legare indirizzo e numero di porta locali o remoti al socket evitando di specificarli ripetutamente quando il socket viene usato per trasferire informazioni.

Una volta che sono stati specificati indirizzo e numero di porta locali e remoti, oppure che è stata creata una connessione dal socket locale al socket di destinazione, questi ultimi possono essere usati per inviare o ricevere dati.

Il socket esiste fino a quando il processo che lo ha creato lo chiude esplicitamente oppure termina. Esistono diversi tipi di sockets che rappresentano classi di servizi ovvero il tipo di socket definisce la semantica della comunicazione.

Per descrivere il tipo di servizio di comunicazione fornito dal livello trasporto (ma anche dai livelli inferiori) ad un programma applicativo esistono diversi fattori che bisogna considerare:

1) connection oriented o connectionless.
2) consegna dei dati sequenziale o non sequenziale
3) garandire o non garantisce il controllo degli errori sui dati trasmessi
4) controllo del flusso dei dati
5) byte stream oppure orientato al messaggio
6) stabilire una connessione full-duplex o half-duplex

Il primo parametro indica se i due programmi applicativi che intendono comunicare stabiliscono o meno un'interconnessione logica (circuito virtuale) prima che la comunicazione possa avere luogo, cioè abbiamo un servizio orientamento alla connessione (connection oriented) o senza connessione (connectionless).

Il secondo parametro descrive se l'ordinamento è sequenziale oppure no, cioè se i dati giungono al ricevitore nello stesso ordine in cui sono stati trasmessi.

Il terzo parametro cioè il controllo degli errori specifica se ai programmi applicativi è garantita una ricezione di dati esenti da errori .

Il quarto parametro cioè il controllo del flusso , noto anche come pacing, garantisce che il ricevitore non sia sopraffatto dal trasmettitore, come accadrebbe se inviasse i dati più velocemente di quanto il ricevitore non possa elaborare.

Il quinto parametro stabilisce se il servizio è di byte stream ossia non indica alcun confine di record alla stream (serie di dati) oppure orientato al messaggio il quale conserva i confini dei messaggi del trasmettitore per il ricevitore.

Infine il sesto parametro indica se la connessione è full-duplex cioè che il trasferimento di dati abbia luogo in entrambe le direzioni contemporaneamente fra le due entità paritarie oppure half-duplex cioè che consentono solamente un trasferimento unidirezionale dei dati.

Chiaramente un protocollo di quelli attualmente esistenti può presentare più parametri insieme, quindi in base ai servizi che un protocollo offre sono stati definiti i seguenti tipi si socket i quali caratterizzano appunto il tipo di servizio offerto dal protocollo.

Il tipo di socket in Unix BSD può essere uno dei seguenti:

socket di stream	sock_stream
socket di datagram	sock_dgram
socket sequenced packed stream	sock_seqpacket
socket raw protocol interface	sock_raw
socket delivered message	sock_rdm

Un ***socket stream*** fornisce un modello di comunicazione orientato alla connessione con un flusso di byte (dati) bidirezionale (full duplex) non strutturato, ma per il quale sono garantite l'affidabilità , la sequenza ordinata e la non duplicazione (nessun dato va perso o duplicato durante la consegna) e non ci sono limiti sulle dimensioni del flusso.

Nel dominio Internet questo tipo di socket è supportato dal protocollo TCP mentre nel dominio XNS dal protocollo SPP, inoltre questo tipo di socket è particolarmente

adatto a problemi che richiedono un grosso scambio di informazioni e tra corrispondenti stabili.

Nelle ***socket datagram*** la semantica della comunicazione permette il trasferimento di messaggi di dimensione variabile in ogni direzione, non garantisce che questi messaggi arrivino nello stesso ordine in cui sono stati inviati, oppure che non vengano duplicati, oppure che arrivino, ma la dimensione originale del messaggio viene conservata in qualsiasi datagram di arrivo.

Questo tipo è supportato nel dominio Xerox NS dal protocollo IDP mentre nel dominio Internet dal protocollo UDP, in quest'ultimo protocollo il flusso di dati è bidirezionale ma non sono garantite la sequenza, l'affidabilità e la non duplicazione; dunque un processo in ricezione su un socket datagram può trovare i messaggi duplicati o assenti ed eventualmente in un ordine diverso da come gli sono stati spediti.

Il protocollo UDP comunque garantisce un'importante proprietà, quella di permettere una comunicazione strutturata: ciò significa che ogni operazione di input output è relativa a un messaggio e non a un flusso di byte.

Inoltre il protocollo non è orientato alla connessione per cui sono le stesse operazioni di input e output che consentono l'indirizzamento corretto dei pacchetti.

Il socket datagram è adatto, dunque, ad applicativi che non prevedono un grosso scambio di informazioni tra corrispondenti e che prevedono una connessione labile e dinamica tra processi comunicanti.

La non affidabilità del tipo di socket può essere accettata con una certa tranquillità quando la rete sottostante è intrinsecamente affidabile.

Un ***socket sequenced packed stream*** ha le caratteristiche di un socket stream, ma in più garantisce il rispetto della struttura a messaggio della comunicazione; è utilizzato solo nel dominio XNS.

Le ***socket raw protocol interface*** permettono l'accesso ai meccanismi di più basso livello, cioè questa semantica di comunicazione permette l'accesso diretto a protocolli di livello network (come ICMP) senza avvalersi di alcun protocollo del livello di trasporto.(questa capacità risulta utile nello sviluppo di nuovi protocolli cioè quando si vuole costruire un particolare protocollo).

Infine le ***socket reliably delivered message*** trasferiscono messaggi garantendone l'arrivo, per il resto sono come i messaggi trasferiti usando le datagram socket; questo tipo di socket non è attualmente supportato, ed anche non tutte le combinazioni di dominio e tipo di socket sono attualmente valide.

Come per il dominio di comunicazione, anche per il tipo di socket, le interfacce tra i sistemi Unix BSD e Windows sono diverse poiché in quest'ultimo gli unici tipi di socket supportati sono attualmente i socket stream e le datagram mentre nel primo sono disponibili tutti i tipi ad eccezione dei sockets reliably delivered message.

Questa limitazione nelle WinSockets non è eccessivamente pesante poiché i possibili applicativi di rete si possono sostanzialmente suddividere in due grosse sezioni: la prima che fa uso di socket con connessione (socket stream), la seconda che fa uso di socket senza connessione (socket datagram).

socket di stream	sock_stream
socket di datagram	sock_dgram

Le specifiche del protocollo TCP-IP non permettono di definire nessun tipo di interfaccia di rete per i programmi applicativi, le sole cose che suggeriscono sono le funzionalità di cui una generica interfaccia ha bisogno.

In generale, una qualunque interfaccia di accesso ai servizi di trasporto in rete deve supportare le seguenti operazioni concettuali:

automaticamente il corretto indirizzo IP locale ed una porta di protocollo locale non utilizzata.

-listen ( ) Questa chiamata di sistema è utilizzabile soltanto con i socket stream ossia per le applicazioni che hanno selezionato un servizio connection-oriented e consente ai servers di preparare un socket per le connessioni in arrivo ponendo il socket in un modo passivo pronto cioè ad accettare le connessioni.
Quando il server chiama la listen, esso provvede anche ad informare il sistema operativo della necessità che il software del protocollo accodi più richieste simultaneamente in arrivo al socket.
I parametri da specificare sono l'identificatore di socket il quale specifica il corrispondente descrittore di socket che dovrebbe essere preparato per l'utilizzo da parte del server e la lunghezza della coda di richiesta per tale socket.
Dopo la chiamata, il sistema accoderà fino al numero specificato (il massimo valore ammesso è 5) come secondo parametro le richieste per le connessioni.
Se la coda risulterà piena quando arriva una richiesta, il sistema operativo rifiuterà la connessione scartando la richiesta.

-accept ( ) Un processo server impiega le chiamate socket, bind e listen per creare un socket, legarlo ad un endpoint locale e specificare una lunghezza della coda per le richieste di connessione.
Il server con la chiamata bind associa il socket ad un endpoint locale, ma il socket non è connesso ad una destinazione specificata, infatti la destinazione esterna deve specificare un "jolly", che consente al socket di ricevere le richieste di connessione da un cliente qualunque.
Una volta che un socket è stato predisposto per attendere le connessioni in arrivo, il server effettua la chiamata accept la quale ha come effetto il bloccaggio da parte del server finché non arriva una richiesta di connessione.
Quando arriva una richiesta di connessione oppure quando il server ha terminato di gestire una precedente richiesta e nella coda ci sono ancora richieste pendenti, l'accept estrae (con una strategia F.I.F.O) la prima richiesta dalla coda e stabilisce con essa una connessione su un nuovo socket che l'accept stessa restituisce.
Il nuovo socket verrà usato per trasferire dati solo per quella nuova connessione, quando la connessione avrà termine il nuovo socket verrà chiuso.
Il descrittore di socket restituito dall'accept fa riferimento ad un'associazione completa, cioè tutti e cinque gli elementi della quintupla relativi al nuovo socket vengono assegnati.
{*protocollo,punto terminale locale,proc. locale punto terminale remoto,proc, remoto*}

-close ( ) Una volta che un client o un server ha finito di usare un socket, può chiamare la close per terminare la connessione e deallocare il socket tentando di consegnare qualsiasi dato già accodato che dev'essere ancora inviato.
Il fatto che un socket costituisca (caso socket stream) un percorso di comunicazione full-duplex, cioè che i dati che fluiscono in una direzione sono logicamente indipendenti dai dati che viaggiano nell'altra direzione, ha spinto i progettisti dell'interfaccia socket ad introdurre una nuova chiamata di sistema la

-shutdown ( ) la quale permette una terminazione della connessione più controllata ossia la chiusura indipendente di una o entrambe le due direzioni.
Un utilizzo di questa chiamata si ha ad esempio quando il server decide di non accettare altre richieste, ultimare le operazioni in corso per poi fare abortire la comunicazione con il client attraverso la chiamata close; altra ipotesi si ha quando il server notifica al client che, terminata la trasmissione corrente, non invierà più

dati. In ogni caso questa chiamata non chiude il socket e si dovrà comunque ricorrere alla close.

Per inviare dati attraverso un socket ci sono cinque chiamate di sistema da cui scegliere:

`send()` `sendto()` `sendmsg()` `write()` `writev()`

Mentre per ricevere dati attraverso un socket esistono cinque chiamate di sistema duali alle chiamate di invio, e queste sono:

`read()` `readv()` `recv()` `recvfrom()` `recvmsg()`

La send, la write e la writev in combinazione alle rispettive chiamate di ricezione funzionano soltanto con socket connessi perchè non consentono al chiamante di specificare un indirizzo di destinazione.

Nelle chiamate send, sendto, recv, recvfrom sono richiesti ulteriori argomenti oltre a quelli standard della chiamata di sistema per la scrittura e la lettura.

Particolarmente importante è l'argomento flags il quale può valere zero oppure viene formato dall'OR di una delle seguenti costanti:

`MSG-OOB`	per inviare dati fuori banda (dati urgenti)
`MSG-PEEK`	per prelevare un messaggio in arrivo
`MSG-DONTROUTE`	per evitare l'instradamento

Il flag MSG-PEEK consente che il chiamante esamini i dati disponibili per la letture sul socket senza richiedere che il sistema scarti i dati dopo il ritorno da recv o recvfrom.

Il flag MSG-DONTROUTE consente al chiamante di richiedere che il messaggio sia inviato senza usare tabelle d'instradamento locali, consentendo al chiamante di assumere il controllo dell'instradamento.

Il flag MSG-OOB permette al trasmettitore di specificare che i messaggi dovrebbero essere inviati fuori-banda su sockets che gestiscono tale nozione.

I dati fuori banda corrispondono alla nozione di dati urgenti del protocollo TCP e sono definiti soltanto per i socket stream.

Nel ricevitore oltre a specificare (con tale flag) che si intende ricevere dati urgenti è necessario predisporre opportune chiamate che hanno il compito di intercettare il segnale che il sistema operativo invia quando il protocollo TCP trasmette la notifica dell'esistenza di dati fuori banda.

1.11.1 Utilizzo delle chiamate sockets elemetari nel client e nel server

Nonostante l'interfaccia socket di Unix BSD permette l'utilizzo di diversi tipi di socket, i possibili applicativi di rete si possono sostanzialmente suddividere in due grosse
sezioni: la prima che fa uso di socket con connessione, la seconda che fa uso di socket senza connessione.

Esistono due protocolli di convenzioni da rispettare perchè uno dei processi funga da server mentre l'altro operi come client , inoltre alcune chiamate possono essere utilizzate solo in uno dei due possibili schemi di comunicazione.

Nel modello di comunicazione senza connessione il dialogo tra i vari corrispondenti è, per così dire, occasionale per cui è nel momento della comunicazione che si specifica chi è il corrispondente.

Dopo l'operazione di creazione del socket (chiamata socket) e di bind esiste il socket ed è visibile all'esterno, esso realizza un legame rigido e permanente (fino alla sua cancellazione) in lettura, mentre è un legame labile in scrittura, nel senso che è necessario specificare a chi inviare i messaggi.

Un socket in questo caso può essere visto come una casella postale da cui il proprietario preleva messaggi senza specificare, evidentemente, i corrispondenti e nella quale può spedire i messaggi, ma specificando il destinatario.

Per consentire la comunicazione tra i processi non è necessaria alcuna ulteriore operazione preliminare oltre la socket e la bind, dopodiché si utilizzano le operazioni di input-output.

Il sistema operativo mette a disposizione cinque possibili chiamate di ingresso-uscita sui socket, queste sono: send, sendto, sendmsg, write, writev per inviare dati e corrispondentemente recv, recvfrom, recvmsg, read, readv per ricevere i dati.

Tutte le chiamate di sistema sopra specificate restituiscono come valore la lunghezza dei dati che sono stati scritti o letti.

La chiamata readv è detta "lettura a spargimento" (*scanner read*) e la writev "scrittura a raccolta" (*gather write*) poiché forniscono la capacità di leggere e scrivere da buffer non contigui.

Nel caso però di comunicazione senza connessione possiamo utilizzare soltanto le chiamate che consentono al chiamante di specificare un indirizzo di destinazione, queste sono la sendto e la sendmsg.

Lo schema delle chiamate di sistema sockets per un protocollo senza connessione è riportato nella figura 16

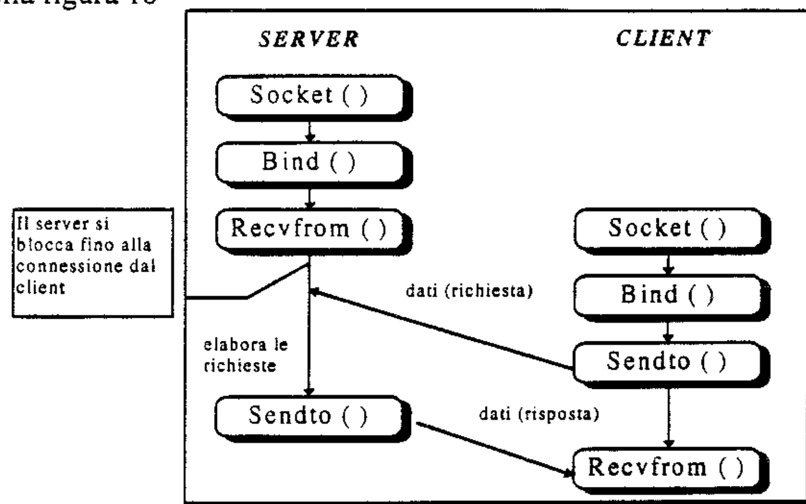


Figura 17 Schema delle chiamate socket del client e del server per un particolare protocollo connectionless

Nello schema di comunicazione con connessione viene definito un canale tra due corrispondenti che hanno la necessita di un grosso scambio di dati non strutturato e per lungo tempo.

Il modello con connessione prevede una fase preliminare in cui viene stabilita una connessione stabile tra i due corrispondenti (circuito virtuale), una fase di scambio informazioni e una fase di abbattimento della connessione.

Il server innanzitutto dovrà creare un socket, associare ad esso indirizzo e un numero di porta locali, e porre il socket in uno stato di "ascolto" (o socket passivo); cioè un socket usato da un server per aspettare che una connessione abbia inizio è chiamato passivo mentre un socket usato da un client per stabilire una connessione è chiamato attivo, quindi la sola differenza tra socket attivo e passivo consiste nel modo con cui l'applicazione usa il socket.

E' necessario introdurre un'ulteriore system call per gestire la fase di predisposizione della connessione, infatti la prima cosa da compiere consiste nell'attivare la system call listen, che predispone il processo ad accettare richieste di connessione su un socket passivo.

Quindi al socket è associata una coda di richieste in attesa e quando è segnalata una richiesta di connessione da parte di un client, il server dovrà stabilire se accettare o rigettare la richiesta.

Se accetta dovrà connettersi al client per ricevere o spedire dati attraverso un socket attivo, infine al termine del dialogo potrà chiudere il socket

Oltre a chiudere il socket con la chiamata close, è anche possibile disabilitare la ricezione o la trasmissione o entrambe utilizzando la funzione shutdown.

Ad esempio il server può decidere di non accettare altre richieste, ultimare le operazioni in corso e terminare il processo (si pensi al termine della erogazione di un servizio); in questo caso invocherà shutdown.

Altra ipotesi è: il server notifica al client che, terminata la tramissione corrente, non invierà più dati. In ogni caso questa chiamata shutdown non chiude il socket e si dovrà comunque ricorrere alla close.

Il processo client eseguirà una sequenza di operazioni analoga a quanto descritto per il server: crea un nuovo socket, lo collega con la chiamata bind ad un indirizzo locale (anche se questa operazione non è necessaria perchè qualsiasi indirizzo-porta è sufficiente e il collegamento sarà formato dalla funzione connect) e si connette con un indirizzo remoto utilizzando appunto la chiamata connect.

Il protocollo d'intesa che è stato descritto si può tradurre in pseudolinguaggio nello schema di invocazioni sotto specificato.

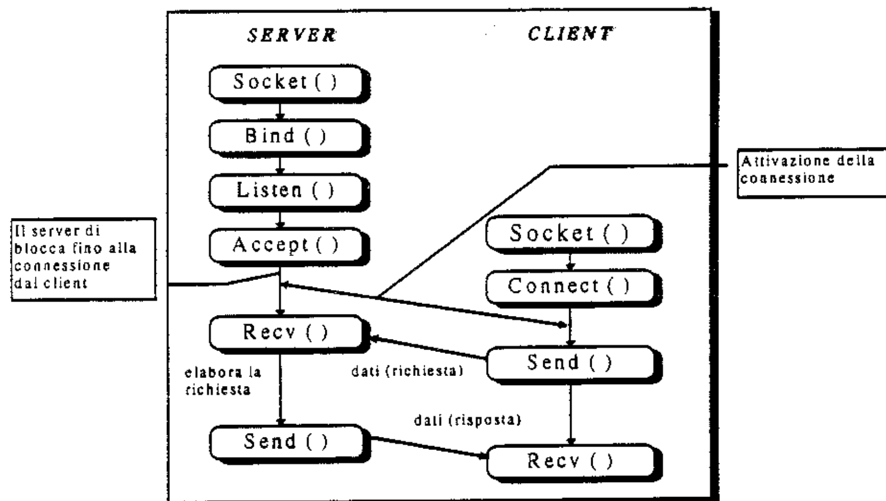


Figura 18 Schema delle chiamate sockets del client e del server per un protocollo connection-oriented

1.11.2 Routines di libreria sockets

Oltre alle chiamate di sistema, l'interfaccia sockets del sistema Unix BSD offre un insieme di routines di libreria che svolgono funzioni utili in relazione al networking.

Le più importanti routines di libreria forniscono servizi di database che consentono ad un processo di determinare nomi di macchine e di servizi di rete, numeri di porta di protocollo ed altre informazioni affini.

Ciascuna di queste routines consente all'applicazione di ottenere informazioni da un sistema di nomi di dominio (*name servers*) che gestisce i vari database della rete internet; in sostanza è come se si stabilisse una connessione col database, si ricevesse l'informazione voluta ed alla fine si chiudesse la connessione.

Ad esempio esistono le procedure di libreria `gethostbyname()` e la `gethostbyaddr()` le quali permettono di reperire informazioni su un host (database dell'host), una volta specificato il suo nome di dominio o il suo indirizzo IP.

Per ottenere informazioni sulla rete, (quali il nome ufficiale dell'host, la lista degli alias, il tipo di indirizzo e l'indirizzo stesso) sono disponibile le routines `getnetbyname()` e la `getnetbyaddr()` le quali forniscono l'accesso al database della rete.

Esistono procedure di libreria che forniscono l'accesso al database dei protocolli disponibili in una macchina (ogni protocollo ha un nome ufficiale ed un corrispondente numero ufficiale) queste sono la
`getprotobyname()` e *la* `getprotobynumber()`.

Inoltre esistono routines di libreria che consentono di ottengono informazioni (dal database di servizio) sui servizi e sulle porte di protocollo che utilizzano; queste sono la `getservbyname()` e la `getservbyport()`.

1.12 Interfacce winsockets

Nell'ambiente operativo Windows fino all'anno 1991 non era presente alcuna interfaccia (API) di sistema per l'uso dei protocolli TCP-IP, ogni produttore di applicazioni di rete doveva fornire una apposita libreria a corredo della propria versione con la conseguenza immediata che ciò impediva la scrittura di applicazioni svincolate dalla particolare API utilizzata.

Questa esigenza di standardizzazione suggerì lo sviluppo di un'interfaccia simile a quella che nel mondo Unix va sotto il nome di interfaccia socket , la quale è lo standard de facto per le API di networking.

Il porting dell'interfaccia socket BSD ha portato alla creazione di una dll , la winsock.dll ,la quale è costituita da un insieme standard di funzioni Berkeley sockets con un ulteriore insieme di funzionalità che sono specifiche di Windows, le *Windows Sockets Asynchronous.*

1.12.1 Le Windows Socket Asincrone

L'introduzione di queste funzioni specifiche di Windows è motivata dal fatto che mentre Unix è un vero sistema operativo multitasking preemptive ossia che ciascuna applicazione (task) non deve preoccuparsi del bloccaggio delle chiamate di input-output poiché la sottostante schedulazione dei processi lo rende trasparente, l'ambiente Windows non è preemptive cioè quando la CPU viene allocata a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio in stato di waiting.

Questo significa che nell'ambiente Windows chiamate di input-output di rete(che sono fondamentali nelle applicazioni di networking) bloccanti potrebbero avere come effetto lo shut down del sistema poiché il kernel non è in grado di riprendersi autonomamente il controllo quando lo desidera, ma lo può ricevere solo dall'applicazione dopo che questa ha terminato di eseguire l'operazione relativa a un determinato messaggio.

Windows utilizza il multitasking non-preemptive basandosi sulla gestione degli eventi e dei messaggi e questo modello event-driven viene usato per la progettazione stessa di applicazioni.

Quando inizia l'esecuzione di un programma (task), Windows crea una coda di messaggi per il programma, in questa coda vengono memorizzati i messaggi per tutte le finestre create dal programma.

Quando si verifica un evento, Windows converte l'evento in un messaggio e tale messaggio può essere o inviato direttamente all'applicazione oppure inserito nell'apposita coda.

Quindi i messaggi che Windows manda alle applicazioni fondamentalmente sono di due tipi: i messaggi diretti ed i messaggi indiretti detti anche rispettivamente messaggi non accodati e messaggi accodati.

I messaggi diretti sono quei messaggi contenenti informazioni che pur essendo di competenza di una applicazione non devono essere gestiti da essa, mentre i messaggi indiretti devono essere gestiti direttamente dall'applicazione, cioè la loro gestione è demandata esclusivamente alle scelte che il programmatore ha effettuato per la gestione di quell'evento.

L'organizzazione tipica di un programma Windows prevede quindi una piccola porzione di codice chiamato loop dei messaggi, in cui i messaggi indiretti verranno prelevati dalla coda e inviati ad una appropriata procedura di gestione dei messaggi (windows procedure), mentre i messaggi diretti verranno inviati direttamente alla finestra che sottende l'applicazione.

Se nella coda dei messaggi di un task non ci sono messaggi in attesa, ma ce ne sono nella coda di un altro task, Windows gestisce questa situazione passando il controllo da un task all'altro.

Questa caratteristica è conosciuta come multitasking non preemptive poiché il passaggio tra due o più applicazioni non avviene come in un sistema operativo preemptive, dove ad ogni task viene assegnato un certo periodo di tempo (time slice) sotto il controllo dello schedulatore della CPU, ma Windows avvia il multitaking nel momento in cui un task esamina la coda dei messaggi.

Windows a livello di programmazione mette a disposizione tre funzioni che permettono di processare messaggi posti in altre code di applicazione, queste sono la `GetMessage()` `PeekMessage()` e `WaitMessage()`.

Quando ognuna di queste funzioni viene chiamata, il controllo viene restituito a Windows, e quindi i messaggi che nel frattempo si sono accumulati nelle code degli altri task possono essere elaborati.

Dato che il bloccaggio delle chiamate di I/O di rete sotto Windows può causare tempi morti (cioè momenti in cui la coda dei messaggi dell'applicazione è vuota e Windows resta in attesa di eventi di rete), le Winsockets asincrone sono state progettate in maniera tale da consentire, durante una chiamata di rete bloccante, ad altri task di funzionare per poi ripassare il controllo al task originale (quello della chiamata bloccante) quando l'evento di rete si verifica.

Per realizzare ciò, delle tre funzioni precedentemente citate, è stata utilizzata la sola funzione `PeekMessage()` il cui utilizzo prevede un loop dei messaggi alternativo al loop standard precedentemente definito.

In questo loop la `PeekMessage` restituisce TRUE se esiste un messaggio nella coda, ed in tal caso il messaggio viene elaborato, se invece restituisce FALSE il task può svolgere altre operazioni prima di ripassare il controllo a Windows.

Per controllare o modificare il loop della `PeekMessage` sono state introdotte quattro funzioni specifiche delle winsockets:

WSAIsBlocking *WSACancelBlockingCall*
WSASetBlockingHook *WSAUnhookBlockingHook*

La `WSAIsBlocking()` controlla se il task corrente è bloccato su un evento pendente ossia su una chiamata che attende il verificarsi di un evento di rete.

`WSACancelBlockingCall()` cancella l'operazione che ha determinato il bloccaggio del task.

Nel caso di applicazioni che vogliono sfruttare completamente i tempi di attesa su una chiamata bloccante (perché necessitano di elaborazioni lunghe e complesse), le winsocket mettono a disposizione del programmatore le funzioni `WSASetBlockingHook()` e la `WSAUnhookBlockingHook()`

La prima funzione dà ai programmatori la possibilità di definire una propria funzione che verrà eseguita esattamente quando il task entrerà nel loop della PeekMessage a seguito di una chiamata di rete bloccante; questa funzione in sostanza permette una gestione alternativa al meccanismo di default delle chiamate bloccanti.

Mentre la funzione WSAUnhookBlockingHook() rimuove ogni precedente blocking hook che è stato installato e reinstalla il meccanismo di bloccaggio di default.

Quindi tutte le funzioni socket asincrone hanno intrinsecamente incorporato (per default) un loop PeekMessage cioè quando una applicazione invoca una delle funzioni winsockets bloccanti, la sua implementazione prevede che l'operazione abbia inizio e che poi si entri in un loop il quale è equivalente al seguente pseudocodice.

```
for(;;) {
  /* controllo i messaggi */
  while(BlockingHook())
      ;
  /* controllo per la funzione WSACancelBlockingCall() */
  if(operazione cancellata)
  break;
  /* controllo per vedere se l'operazione è completata */
  if(operazione completata)
  break; /* normale completamento */
  }
```

Occorre sottolineare che l'implementazione delle winsockets può elaborare i passi sopra descritti in differente ordine; per esempio, controllare il completamento dell'operazione può avvenire prima della chiamata alla blocking hook.

La funzione di default BlockingHook() è equivalente a:

```
BOOL DefaultBlockingHook(void) {
  MSG msg;
  BOOL ret;
  /* prende il prossimo messaggio se esiste */
  ret = (BOOL)PeekMessage(&msg,NULL,0,0,PM_REMOVE);
  /* se c'è un messaggio, questo viene processato */
  if (ret){
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    }
  /* TRUE se non c'è nessun messaggio*/
  return ret;
}
```

Le Windows Sockets Asynchronous , basandosi sul modello di gestione degli eventi e dei messaggi, evitano il bloccaggio delle chiamate di input-output su rete attraverso il meccanismo precedentemente descritto, permettendo quindi la realizzazione di applicativi di networking per l'ambiente Windows molto efficienti.

1.12.2 La WSAAsyncSelect.

Per vedere quando e come viene utilizzata questa funzione riprendiamo lo schema delle chiamate sockets per la realizzazione di un server connection-oriented con un esempio.

Possiamo osservare che, dopo aver predisposto il server ad accettare richieste di connessione sul socket passivo, la successiva cosa da fare è ascoltare se ci sono richieste in arrivo, però prima di fare questo vogliamo mettere il socket in uno stato non bloccante per non aspettare indefinitamente una richiesta in arrivo (potrebbe anche non arrivare).

Il socket opera per default in modo bloccante.

Sotto il sistema Unix esiste la possibilità attraverso la chiamata `ioctl()` o la `fcntl()` di manipolare le opzioni del socket, porlo ad esempio in uno stato non bloccante, e utilizzare la chiamata `select()`, la quale consente al processo utente di ordinare al kernel di attendere uno qualsiasi di vari eventi (descrittore di socket pronto per la lettura, pronto per la scrittura, pronto per condizioni eccezionali quali l'arrivo di dati fuori banda), per risvegliare il processo solo quando si verifica uno di tali eventi.

In Unix quanto detto è possibile poiché i sockets, come tutti i dispositivi, vengono trattati come files.

Sotto Windows questo non è più vero, ed allora si è introdotta la funzione `ioctlsocket()` e la funzione `WSAAsyncSelect()`.

La prima permette di ottenere informazioni di stato o di predisporre il socket come bloccante o non bloccante.

La `WSAAsynctSelect()` informa la winsock.dll a quale evento siamo interessati e quest'ultima, quando si verifica l'evento, converte l'evento in un messaggio (inizialmente definito dal programmatore) e lo pone nella coda dei messaggi dell'applicazione, inoltre imposta il socket come non bloccante.

```
int WSAAsyncSelect(Socket,Task,Message,Evento)
```

Il messaggio associato all'evento deve essere un nuovo messaggio di Windows che il programmatore avrà opportunamente definito, mentre i bits di maschera che descrivono l'evento al quale siamo interessati sono:

FD_READ	Indica che i dati sul socket sono disponibili per la lettura
FD_WRITE	Indica che si può scrivere sul socket
FD_OOB	Indica che dati urgenti sono disponibili sul socket per la lettura
FD_ACCEPT	Indica che una richiesta di connessione è arrivata sul socket passivo del server
FD_CONNECT	Indica che il tentativo da parte del client di stabilire una connessione sul socket si è completata.
FD_CLOSE	Indica che il socket potrà essere chiuso

Nella WSAAsyncSelect è possibile definire una combinazione di bits di maschera attraverso l'operatore OR, ad esempio per ricevere entrambe le notificazioni di lettura e scrittura sul socket l'applicazione deve chiamare la WSAAsyncSelect con FD_READ e FD_WRITE come sotto riportato

```
int WSAAsyncSelect(Socket,Task,Message,FD_READ | FD_WRITE)
```

Questa chiamata asincrona delle winsockets ha la proprietà di cancellare ogni precedente notifica fatta sullo stesso socket, cioè non è possibile specificare differenti messaggi per differenti eventi.

Il codice sotto riportato ha il solo effetto di notificare, sul socket s, l'evento FD_WRITE attraverso il messaggio wMsg2 mentre la prima chiamata non produce alcun messaggio wMsg1 legato all'evento FD_READ poiché questa è annullata dalla seconda chiamata asincrona.

```
int WSAAsyncSelect(s,hWnd,wMsg1,FD_READ);
int WSAAsyncSelect(s,hWnd,wMsg2,FD_WRITE);
```

Per cancellare tutte le notificazioni di networking, ossia per informare la winsock.dll di non inviare all'applicazione ulteriori messaggi di rete relativo al socket s bisogna impostare al valore 0 il parametro `Evento` ed il parametro `Message` nella `WSAAsyncSelect();`

```
int WSAAsyncSelect(Socket,Task,0,0);
```

Sebbene la `WSAAsyncSelect()` sopra descritta disabiliti tutti i messaggi relativi a eventi di rete sul socket specificato, l'applicazione memorizza tali messaggi di rete nella propria coda dei messaggi in attesa di una loro riabilitazione sempre attraverso la `WSAAsyncSelect()`.

Riprendendo l'esempio iniziato precedentemente: quando il client cerca di connettersi al server, la winsock.dll riceverà dal corrispondente driver il segnale di rete corrispondente a quella richiesta di connessione ed allora convertirà l'evento nel messaggio windows corrispondente a quell'evento il quale verrà poi gestito nella windows procedure.
Questo meccanismo (`event-driven`) è lo schema di base delle winsocket asincrone di windows.

1.12.3 Le funzioni di libreria socket asincrone

Altre funzioni Sockets asincrone introdotte come estensione ai sockets BDS sono quelle relative al reperimento e alla definizione attraverso i name servers di informazioni relative al nome di un host, agli indirizzi degli host, al nome o numero del protocollo e alla porta, questo sono:

- *WSAAsyncGetHostByAddr()* - *WSAAsyncGetHostByName()*
- *WSAAsyncGetProtoByNam()* - *WSAAsyncGetProtoByNumber()*
- *WSAAsyncGetServByName()* - *WSAAsyncGetServByPort()*

Tutte queste funzioni sono disponibili anche nella versione non asincrona con tutti i problemi che ne conseguono.
L'utilizzo di queste winsocket nella versione asincrona prevede la definizione da parte del programmatore di un nuovo messaggio di windows il quale sarà inviato al task e quindi alla corrispondente windows procedure nel momento in cui l'operazione con il name server si è completata.
L'implementazione di queste funzioni prevede che quando ha inizio l'operazione questa restituisca immediatamente il controllo al chiamante e solo quando l'operazione si è completata il messaggio sia inviato al relativo task per una sua elaborazione.

1.14 Schemi semplificati dei moduli (client e server) connection-oriented nel sistema Unix e Windows

In questa parte conclusiva del manuale descriviamo, attraverso il linguaggio di programmazione C, alcuni schemi di programmi per i vari tipi di server e client connection-oriented nei sistemi Unix e Windows mettendo soprattutto in evidenza i sockets coinvolti nella comunicazione e le chiamate di rete utilizzate.

1.14.1 Schema di programma di un Server Concorrente connection-oriented nel S.O Unix

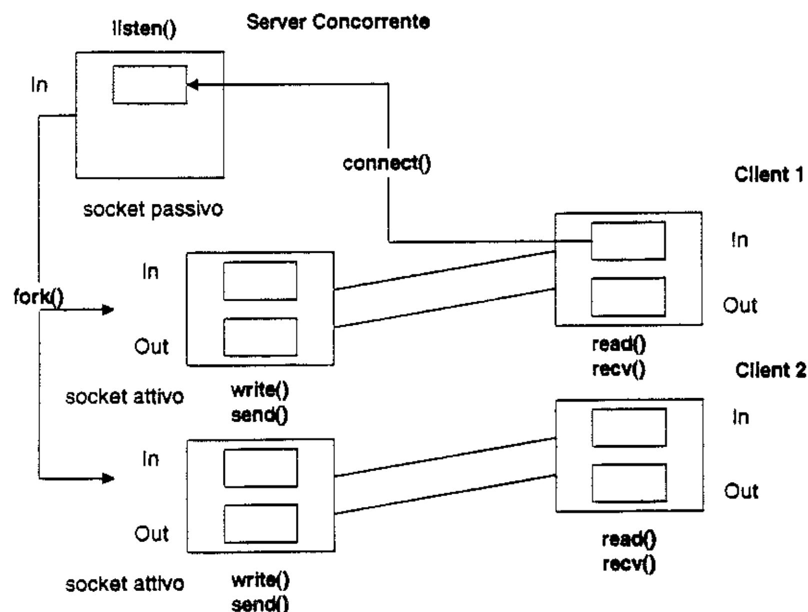
Nel listato riportato di seguito sono riportate le chiamate sockets di rete per la realizzazione di un server concorrente connection-oriented per il sistema operativo Unix.

```
#include <sys/type>
#include <sys/socket>
#include <netinet/in.h>
#include <arpa/inet.h>
#define SERVER_PORT 2000
```

```
main(argc, argv)
int sockpassivo, sockattivo;
struct sockaddr_in server_addr, cliente_addr;
server_addr.sin_family=AF_INET
server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
server_addr..sin_port=htons(SERVER_PORT);
sockpassivo=socket(AF_INET, SOCK_STREAM, 0)
bind(sockpassivo, (strauct sockaddr ]) &server_addr, sizeof(server_addr));
listen(sockpassivo,backlog);
while (1)
   {
   sockattivo=accept(sockpassive, (struct sockaddr *) &client_addr, &length);
   if ((childpd=fork())==0)
         close(sockpassivo);          /* processo figlio*/
         recv(sockattivo, , ,)
         elaborazione del servizio sul sockattivo;
         send(sockattivo, , ,)
         close(sockattivo);
         exit(0);
   }
```

Lo scopo di questo scheletro di programma (ma questo vale anche per i successivi) è quello di fornire, attraverso un linguaggio di programmazione di sistema quale il C, un esempio di implementazione (molto semplificata) di moduli di rete, con caratteristiche di gestione del servizio e di canali di comunicazione coinvolti durante l'elaborazione, visti nella parte iniziale di questo testo.

Per questo tipo di modulo di networking ad un generico istante abbiamo ad esempio i sockets rappresentati nella figura sottostante: il socket passivo in corrispondenza della chiamata listen il quale è predisposto ad aspettare richieste di connessione mentre i restanti sockets attivi sono relativi a ciascun client che il server è in grado di gestire concorrentemente (notare la rappresentazione dei sockets come canali full-duplex)



1.14.2 Schema di programma di un Server Sequenziale e di un Server Apparentemente Concorrente connection-oriented nell'ambiente operativo Windows

Windows 3.11 (per ciò che riguarda Windows 95 la descrizione e l'uso di tale S.O., per la realizzazione di moduli di rete su questo nuovo ambiente, esula da questo testo) è un sistema non preemptive e questo significa che il kernel non è in grado di riprendersi autonomamente il controllo quando lo desidera, ma lo può ricevere solo dall'applicazione, dopo che l'applicazione stessa ha finito di seguire un'unità di elaborazione (sotto l'ambiente Windows 3.11 per unità di elaborazione si intende l'insieme di istruzioni associate ad un messaggio windows in una *windows procedure*).

Compito del programmatore di applicazioni che operano in un contesto di multitasking cooperativo è di evitare di creare applicazioni che prendano il controllo del sistema per un tempo troppo elevato a scapito di tutte le altre.

La soluzione a questo problema è ad esempio quella di dividere l'applicazione in unità di elaborazione possibilmente indipendenti e che richiedono tempi di esecuzione molto brevi, al termine delle quali il controllo può essere restituito al kernel.

Oltre alle considerazioni sopra riportate, le quali valgono non solo per applicativi di networking ma per qualunque altro tipo di applicazione operante su questo ambiente operativo, dobbiamo anche tenere presente (secondo punto) che le operazioni di input-output di rete se eseguite in modalità sincrona possono causare lo shut down del. sistema, in tale contesto ricordiamo che la winsock.dll permette l'utilizzo di alcune chiamate asincrone di rete le quali, come descritto nel capitolo precedente, permettono di effettuare operazioni di input-output sui sockets basandosi su uno schema event-driven.

Tenendo presenti i due punti sopra descritti possiamo senz'altro affermare che per implementare in modo efficiente un'applicazione Server in ambiente Windows è indispensabile utilizzare le operazioni di rete basandoci su uno schema message-driven e suddividere l'intera elaborazione in unità veloci e tra di loro indipendenti.

In aggiunta, per quanto abbiamo descritto nel capitolo precedente sui vari tipi di server che utilizzano un protocollo connection-oriented, le possibili soluzioni in riferimento alla possibilità di gestire più richieste concorrentemente utilizzando il sistema Windows 3.11 sono sostanzialmente due:

- Server sequenziale
- Server apparentemente concorrente

Queste due soluzioni si differenziano nel modo in cui vengono gestite le richieste provenienti dai vari clients ossia dai sockets coinvolti in un generico istante.

Nel primo tipo di server è previsto un solo socket attivo e lo schema di funzionamento è sequenziale ossia viene gestita una richiesta alla volta e solo quando questa ha termine si torna a gestirne un'altra.

Nel secondo tipo di server sono possibili più socket attivi, in questo caso lo schema di funzionamento è quello apparentemente concorrente in quanto il server è in grado di gestire contemporaneamente più richieste provenienti dai clients intercalando ogni singola unità di elaborazione in base ad uno schema di funzionamento event-driven.

Per chiarire maggiormente i due tipi di server viene presentato un esempio di schema di implementazione sequenziale anche se la realizzazione del Server-Windows è stata fatta su uno schema di tipo apparentemente concorrente.

Un esempio di implementazione che prevede un unico socket attivo è ad esempio quella riportata di seguito nella quale sono stati definiti quattro messaggi windows ognuno dei quali è legato ad un evento di rete che la winsock.dll può gestire.

Lo schema funzionale è quello in cui le chiamate winsocket asincrone si succedono sequenzialmente nell'elaborazione.

```
#include "winsock.h"
WinMain( )
   {PostMessage(hWnd,WM_LISTEN,0,0);}
WinProc
   switch (message)
   {
   case WM_LISTEN:
      PassivSocket = socket( );
      bind( );
      listen( );
      WSAAsynctSelect(PassivSocket,hWnd,WM_CONNECT,FD_ACCEPT);
   case WM_CONNECT:
      AttivSocket = accept(PassivSocket,,,);
      WSAAsynctSelect(PassiveSocket,hWnd,0,0);
      WSAAsynctSelect(AttivSocket,hWnd,WM_GETREQUEST,FD_READ);
   case WM_GETREQUEST:
      Blocking(AttivSocket);
      recv(AttivSocket,,,,)
      Elaborazione del servizio;
      WSAAsynctSelect(AttivSocket,hWnd,WM_REPLY,FD_WRITE);
   case WM_REPLY:
      Blocking(AttivSocket);
      send(AttivSocket,,,);
      WSAAsynctSelect(PassivSocket,WM_CONNECT,FD_ACCEPT);
   }
```

Con tale implementazione il funzionamento di un Server Windows è tipicamente sequenziale (il Server è in grado di gestire una richiesta alla volta), i socket coinvolti sono sempre due: quello passivo sul quale si ricevono richieste di connessione ed uno attivo per ricevere e trasmettere i dati al client.

Per quanto riguarda i nuovi messaggi di rete sono stati definiti quattro windows-message ciascuno associato a eventi di rete che il Server-Windows deve gestire, questi sono:

- ◊ Predisposizione ad accettare richieste di connessione `WM_LISTEN`
- ◊ Attivazione di una connessione `WM_CONNECTION`
- ◊ Ricezione delle richieste `WM_GETREQUEST`
- ◊ Trasferimento dei risultati `WM_REPLY`

La definizione di messaggi Windows definiti dal programmatore che non siano di sistema può essere fatta definendo il messaggio come un nuovo messaggio utente oppure usando la funzione di sistema `RegisterWindowsMessage();` il codice sottostante chiarisce i due metodi.

```
#define WM_LISTEN  (WM_USER+1)
```

oppure

```
RegisterWindowsMessage(WM_USER);
```

Il messaggio `WM_LISTEN` viene inserito attraverso l'API di windows `PostMessage()` nella coda dell'applicazione nel momento stesso in cui questa viene lanciata.

La routine di gestione per questo messaggio prevede la creazione di un socket passivo per le richieste di connessione da parte dei clienti, e le funzioni socket impiegate sono la socket la bind e la listen.

Alla fine della routine di gestione di questo messaggio di rete viene impiegata la `WSAAsyncSelect()` la quale notifica che l'evento di una nuova connessione da parte

di un cliente verrà espresso tramite la generazione del messaggio windows WM_CONNECTION.

Il messaggio WM_CONNECTION viene generato dalla winsock.dll nel momento in cui un nuovo cliente richiede una connessione al server.

La routine di gestione di questo messaggio prevede l'impiego della funzione accept la quale restituirà il socket utilizzato per lo scambio dei dati; immediatamente dopo la creazione del socket attivo, utilizzando la WSAAsynctSelect(), vengono cancellate tutte le notificazioni sul socket passivo cioè viene segnalato all'implementazione delle winsocket di non mandare nessun ulteriore messaggio relativo a eventi di rete sul socket passivo.

Procedendo con un unico socket abilitato ad accettare messaggi di rete garantiamo un servizio di tipo sequenziale; occorre inoltre ricordare che nonostante i messaggi sul socket passivo siano disabilitati questi vengono mantenuti nella coda dei messaggi associata all'applicazione in attesa di una loro riabilitazione che avviene attraverso l'utilizzo della WSAAsycnSelect() alla fine del servizio sul socket attivo.

Alla fine la routine impiega la funzione WSAAsyncSelect() per comunicare alla winsock.dll che nel momento in cui riceverà la notifica di un dato disponibile per la lettura questa dovrà generare il messaggio windows WM_GETREQUEST.

Nella routine di gestione del messaggio WM_GETREQUEST il server setta il socket come bloccante riceve i parametri della query, elabora il servizio richiesto ed alla fine attraverso la chiamata select asincrona si notifica l'evento di scrittura sul socket.

Utilizzando le Winsockets, a differenza di quanto avviene ad esempio nel sistema operativo UNIX, per bloccare un socket non basta usare la chiamata di sistema ioctl() con argomento la costante FIONBIO ma occorre prima disabilitare i messaggi di rete relativi al socket attraverso la chiamata WSAAsynctSelect() e successivamente eseguire la funzione ioctlsocket(); la porzione di codice sotto riportata mostra come bloccare un socket in ambiente Windows.

```
u_long nonblock = FALSE;
WSAAsynctSelect(socket,hWnd,0,0);
ioctlsocket(socket,FIONBIO,&nonblock);
```

Ricordiamo che con la programmazione di rete le operazioni di invio e ricezione devono essere necessariamente bloccanti poiché una richiesta di I/O che non possa essere completata su un socket non bloccante non viene effettuata e ciò comporta come conseguenza una errata trasmissione e ricezione delle informazioni.

Nella routine di gestione del messaggio WM_REPLY si setta il socket come bloccate in modo da rendere le chiamate di scrittura bloccanti, si trasferiscono al client i risultati generati dal servizio richiesto infine terminato il trasferimento dei risultati si esegue la funzione WSAAsyncSelect con parametri il messaggio WM_CONNECTION e notifica FD_ACCEPT per esprimere il fatto che da questo punto in poi il server potrà gestire una nuova richiesta.

In questo esempio il server per quanto riguarda la ricezione della richiesta, l'elaborazione ed il trasferimento dei risultati offre un servizio di tipo sequenziale, ad un generico istante il server è in grado di gestire una richiesta alla volta e solo quando finisce con un particolare client ritorna ad accettare una nuova richiesta di connessione.

Occorre ricordare che le eventuali richieste di connessione che si potrebbero verificare durante la gestione sequenziali sono accodate dalla winsock.dll mentre il sistema è in attesa che il server esegua la chiamata accept per gestire una nuova richiesta.

Uno schema di funzionamento di questo tipo risulta poco efficiente in quanto una nuova richiesta di connessione viene semplicemente accodata e rimane passiva durante tutta la gestione sequenziale del servizio.

Sfruttando lo schema message-driven di Windows e le Winsock asincrone (in particolare la `WSAAsynctSelect`) è possibile implementare Servers Windows secondo uno schema apparentemente concorrente ossia con più canali di comunicazione attivi ed una gestione dell'intero servizio completamente guidata dagli eventi di rete.

Prima di passare a descrivere l'architettura di un generico Server Windows Apparentemente Concorrente è necessario sottolineare alcune proprietà funzionali legate ai messaggi Windows le quali sono state impiegate per la realizzazione del modulo di rete.

Nel modello di networking event-driven di Windows è molto importante disporre di informazioni associate al messaggio di rete, queste informazioni sono il numero di socket, lo stato del messaggio ed il codice di evento legato al messaggio.

Tutte queste informazioni sono inserite dalla winsock.dll nella struttura record del messaggio Windows e sono sempre passate come parametri alle windows procedures presenti nell'applicazione.

La struttura record dei messaggi di Windows contiene 6 campi; ciascuno dei essi ha un significato particolare riportato nella tabella 1.

Hwnd	handle della finestra a cui è destinato il messaggio
Message	tipo di messaggio identificato da un valore numerico
WParam	parametri che indica il significato e il valore da cui dipende un particolare messaggio
LParam	parametri che contengono informazioni supplementari sul tipo del messaggio
Time	tempo relativo all'inserimento del messaggio nella coda associata all'applicazione
Tpoint	coordinate del sistema di puntamento

Tabella 1 Struttura record dei messaggi di Windows

```
WndProc(HWND hWnd,UINT msg,UINT WParam,LONG LParam)
```

Nel caso di messaggi di rete il parametro `WParam` contiene il numero del socket mentre il parametro `LParam` è sostanzialmente suddiviso in due parole ciascuna di 16 bit dove la low word contiene il codice di evento (ricordiamo che nel caso di messaggi di rete il codice di evento può essere una delle costanti sotto riportate:

FD_READ	FD_WRITE	FD_OOB	FD_ACCEPT	FD_CONNECT	FD_CONNECT

la high word contiene lo stato dell'operazione dove 0 indica successo; la winsock.dll nel caso di `LParam` fornisce due macro sotto riportate:

WSAGETSELECTEVENT( ) WSAGETSELECTERROR( )

la prima permette di estrarre la low word di `LParam` ossia il codice di evento, mentre la seconda la high word cioè il codice di errore.

Passiamo ora a descrivere un esempio di implementazione di Server Windows che permette una gestione delle richieste di tipo apparentemente concorrente.

L'implementazione di tale modulo prevede (ad esempio) sei nuovi windows-message associati a eventi di rete più un ulteriore messaggio per l'elaborazione del servizio offerto; i messaggi di rete sono:

WM_LISTEN	WM_CONNECTION	WM_GETREQUEST	WM_REPLY	WM_CLOSE

mentre il messaggio associato al servizio che viene implementato è `WM_QUERYPROC`; a ciascuno di questi messaggi come nell'esempio precedente è associata una routine di gestione del messaggio.

Attraverso la definizione di questi messaggi si è reso possibile organizzare l'attività funzionale del server distinguendo le seguenti fasi di elaborazione:

- ◊ Predisposizione ad accettare richieste di connessione WM_LISTEN
- ◊ Attivazione di una connessione WM_CONNECTION
- ◊ Ricezione delle richieste WM_GETREQUEST
- ◊ Elaborazione delle richieste WM_QUERYPROC
- ◊ Trasferimento dei risultati WM_REPLY
- ◊ Chiusura della connessione da parte del client WM_CLOSE

Per ciascuna richiesta inviata dal Client il Server Windows esegue il relativo servizio eseguendo in successione una certa sequenza di attività funzionali; ricordiamo che ciascuna di queste attività, per come è stato progettato il server, può essere intercalata temporalmente (in base alla gestione event-driven) dall'esecuzione del servizio relativo ad un client differente.

Dopo aver inserito il messaggio WM_LISTEN nella coda dei messaggi associata all'applicazione, nella routine di gestione di questo windows message viene attivato il socket passivo sul quale verranno accettate le richieste di nuove connessioni da parte dei clients; le chiamate socket necessarie per predisporre il Server Windows ad accettare richieste di connessione sono la socket, la bind e la listen; alla fine viene eseguita la chiamata asincrona WSAAsynctSelect per notificare che su quel socket passivo potranno essere accettate richieste di connessione.

Nel programma abbiamo definito una struttura dati globale che abbiamo chiamato ActiveSocketList in cui vengono memorizzati al più MaxLevelOfConcorrency di socket attivi, questa struttura viene utilizzata per tutta la restante parte dell'elaborazione poiché in essa vengono memorizzati i socket attivi.

Di seguito è riportato uno spezzone di codice C relativo all'implementazione del Server Windows mettendo in rilievo soprattutto i messaggi di rete, le chiamate socket elementari e le chiamate Asincrone della WSAAsynctSelect().

```
#include "winsock.h"
WinMain( )
{
switch (wMsg)
{
case WM_LISTEN:return(*DoListen)(hWnd,wMsg,wParam,lParam);
case WM_CONNECTION:return(*DoConnection)(hWnd,wMsg,wParam,lParam);
case WM_GETREQUEST:return(*DoGetRequest)(hWnd,wMsg,wParam,lParam);
case WM_QUERYPROC:return(*DoQueryProc)(hWnd,wMsg,wParam,lParam);
case WM_REPLY:return(*DoReply)(hWnd,wMsg,wParam,lParam);
case WM_CLOSE:return(*DoClose)(hWnd,wMsg,wParam,lParam);
}
return(DefWindowProc(hWnd,wMsg,wParam,lParam));

PostMessage(hWnd,WM_LISTEN,0,0);
ActiveSocketList[MaxLevelOfConcurrencySocket]
while (GetMessage(msg,,,,))
{
TranslateMessage(msg);
DispatchMessage(msg);
}
}
WinProc
switch (message)
  {
case WM_LISTEN:DoListen(hWnd,wMsg,wParam,lParam)
  PassiveSocket = socket( );
  bind(PassiveSocket, , );
  listen(PassiveSocket,);
```

```
  WSAAsynctSelect(PassivSocket,hWnd,WM_CONNECT,FD_ACCEPT);
case WM_CONNECT:DoConnection(hWnd,wMsg,wParam,lParam)
Scandisco la struttura ActiveSocketList per  vedere se esiste un posto
libero per un nuovo Socket Client
  Se non esiste
    return 0;
  Se esiste
    Aggiorno la struttura dati inserendo il nuovo Socket Client
    ActiveSocket_k = accept(PassiveSocket,,);
  WSAAsynctSelect(ActiveSocket_k,hWnd,WM_CLIENT,FD_READ|FD_WRITE|FD_CL
OSE)
case WM_GETREQUEST:DoGetRequest(hWnd,wMsg,wParam,lParam)
  (WSAGETSELECTEVENT(lParam) == FD_READ)
  Scandisco la struttura ActiveSocketList per trovare il nome  del
file utilizzato per memorizzare la query
  {
  Blocking(ActiveSocket_k);
  recv(ActiveSocket,,,);
  NonBlocking(ActiveSocket_k);
  PostMessage(hWnd,WM_QUERYPROC,ActiveSocket_k);
  return 0;
}
case WM_QUERYPROC:DoQueryProc(hWnd, wMsg, wParam, lParam)
  ricavo da wParam il socket quindi scandisco la struttura
  ActiveSocketList per ricavare i files della query;
  Elaborazione della Query
  ActiveSocket_k = wParam;
  return 0;
case WM_REPLY: DoReply(hWnd, wMsg, wParam, lParam)
 if(WSAGETSELECTEVENT(lParam)==FD_WRITE)
  {
  Blocking(ActiveSocket_k=wParam)
  send(ActiveSocket_k, , );
  return 0;
  }
 else return 0;
case WM_CLOSE:DoClose(hWnd, wMsg, wParam, lParam)
 if(WSAGETSELECTEVENT(lParam) == FD_CLOSE)
  {
  KillConn( );
  return 0;
  }
}
```

1.14.3 Schema di programma di un Client Iterativo connection-oriented nel S.O. Unix

Il programma client di questo esempio segue il flusso di chiamate di sistema che è stato illustrato nei paragrafi precedenti, in particolare abbiamo il seguente codice semplificato:

```
#include <sys/type>
#include <sys/socket>
#include <netinet/in.h>
#include <arpa/inet.h>
#define SERVER_PORT 2000
#define SERVER_ADDRESS "131.114.2.42"
main(argc, argv)
int sock;
struct sockaddr_in server_addr;
server_addr.sin_family=AF_INET
server_addr.sin_addr.s_addr=inet_addr(SERVER_ADDRESS);
```

```
server_addr..sin_port=htons(SERVER_PORT);
sock=socket(AF_INET, SOCK_STREAM, 0)
connect(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
send(sock, , , );
recv(sock, , , );
close(sock);
exit(0);
```

1.15 Gestione degli errori

Come ultimo argomento, non per questo meno importante, vediamo uno degli aspetti più importanti della programmazione di applicativi di networking ossia la gestione degli errori, è infatti attraverso una loro continua rilevazione e trattamento che possiamo ottenere soluzioni particolarmente robuste.

Per quanto riguarda gli errori che possono manifestarsi su un server questi possono, in generale, essere suddivisi in due classi

- errori di comunicazione
- errori di elaborazione del servizio

Fanno parte della prima classe tutti quegli errori che possono manifestarsi sui sockets quindi sono errori che l'interfaccia di comunicazione (winsock.dll o i socket BSD) è in grado di rilevare e questi possono essere generati da malfunzionamenti sulla rete di comunicazione, dalla mancanza di risorse di sistema (di cui la winsock.dll o il S.O. necessita) ed infine da specifiche mancanti e/o errate fatte durante l'implementazione.

Per quanto riguarda gli errori della seconda classe questi sono generalmente dovuti a malfunzionamenti del sistema durante l'elaborazione del servizio e/o parametri d'interrogazione illeciti formulati dall'utente per accedere al servizio che il server offre.

Nel caso specifico di moduli Servers, i possibili errori che possono manifestarsi durante il servizio possono essere gestiti in due modi differenti:

i) il server risolve il problema dell'errore autonomamente interrompendo l'istruzione che ha causato l'errore, disallocando tutte le risorse compresa la comunicazione ed andando a gestire eventuali altre richieste provenienti dai clients.

ii) il server può passare all'applicazione client il codice di errore che è all'origine del malfunzionamento, rilasciare tutte le risorse (di rete e locali legate a quella comunicazione utente) e andare a gestire altre richieste.

La scelta di un metodo piuttosto che un altro dipende ovviamente dalla particolare situazione, e non ha molto senso adottare una strategia generale in proposito; l'unico accorgimento strettamente indispensabile è di fare in modo che un errore nel server non provochi un errore fatale che abbia come conseguenza la terminazione dell'esecuzione del server stesso (crash dell'applicazione server).

Certamente, nel caso di interrogazioni illecite, la soluzione di passare all'applicazione client il codice di errore risulta più appropriata in quanto consente all'utente di determinare se un particolare errore sia stato generato dal server oppure dall'interrogazione da lui formulata.

Per quanto riguarda la gestione degli errori sui sockets in una generica applicazione di rete sotto l'ambiente Windows, i due principali tipi di errori restituiti dalle funzioni di sistema sono:

- `INVALID_SOCKET` relativo alla fase di creazione dei sockets ed
- `SOCKET_ERROR` per le altre possibili operazioni sui sockets.

Durante l'implementazione è quindi necessario testare queste due condizioni prima di invocare la successiva funzione per evitare di porre il sistema in condizioni instabili, a tale scopo le Winsockets offrono la funzione `WSAGetLastError()` la quale

restituisce una descrizione più precisa della causa di errore sul socket (codice d'errore); tale funzione si basa sulla variabile globale errno.

Ad esempio dopo un errore corrispondente alla chiamata socket( ) di creazione di un socket, può essere utilizzata la funzione WSAGetLastError() la quale restituisce uno dei possibili codici di errore elencati nella tabella sottostante.

WSANOTINITALISED	il sistema non è stato inizializzato con la WSAStartup( )
WSAENETDOWN	la rete non è attiva o si è verificato un errore di rete
WSAEAFNOSUPPORT	la famiglia di indirizzi specificata non è supportata
WSAEPROTONOSUPPORT	il protocollo specificato non è supportato
WSAEPROTOTYPE	il tipo di socket richiesto non supporta il protocollo specificato
WSAESOCKTNOSUPPORT	il tipo specificato non è supportato dalla famiglia di indirizzi fornita

Tabella 2 Codici di errore delle winsocket relativi alla chiamata socket()

Se invece l'errore si verifica in corrispondenza di una chiamata di invio di dati attraverso un socket (send()) il questo caso la WSAGetLastError() restituisce dei codici di errore come quelli riportati nella tabella sottostante.

WSAEINPROGRES	Una operazione su un socket bloccante è in corso
WSAENETRESET	La connessione deve essere resetata poiché abbandonata dalla winsock.dll
WSAENOBUFS	Si è verificato un deadlock sul buffer associato al socket
WSAENOTCONN	Il socket non è connesso
WSAENOTSOCK	Il descrittore non è un socket
WSAEINVAL	Il socket non è stato legato con la chiamata bind( )
WSAECONNABORTED	Il circuito virtuale è abortito per il timeout o altro fallimento
WSAECONNRESET	Il circuito virtuale è stato resetato dall'host remoto

Tabella 3 Codici di errore delle Winsocket relativi alla chiamata send()

Come si può dedurre, in entrambi i casi è possibile individuare il preciso tipo di errore e decidere se è recuperabile o non recuperabile; nel primo caso, ad esempio, richiamando la funzione sulla quale si è verificato l'errore dopo un certo periodo di tempo (caso di guasto temporaneo della rete), nel secondo caso abbandonando la chiamata, rilasciando tutte le risorse corrispondenti alla gestione di quel servizio, aggiornando la struttura dati dei socket attivi e andando ad ascoltare nuove richieste riportandosi in questo modo in uno stato consistente.

Nel caso di applicazioni di rete per il S.O. Unix la gestione degli errori prevede l'utilizzo di due distinte funzioni: strerror() e herror(), la prima è basata sulla variabile globale errno e restituisce il codice di errore in corrispondenza di una chiamata di sistema sia essa di rete o locale mentre la funzione herror() restituisce il messaggio di errore basato sulla variabile globale esterna h_errno in corrispondenza di funzioni di libreria comprese le routine di libreria delle sockets BSD.

Il seguente esempio mostra l'utilizzo della funzione strerror() in corrispondenza della chiamata di sistema di rete socket:

```
s = socket(AF_INET,SOCK_STREAM,0)
if (s < 0)
fprintf(stderr,"socket:%s \n",strerror(errno));
```

Quando si verifica un errore, la variabile globale errno assume un valore minore di 0 e per risalire al codice che verifica la causa dell'errore viene utilizzata la funzione strerror().

I possibili errori su una chiamata socket() sono riportati come esempio nella tabella sottostante

Valore in errno	Causa dell'errore
EPROTONOSUPPORT	Errori sugli argomenti; il servizio o il protocollo specificati non sono validi
EMFILE	La tabella descrittori dell'applicazione è completa
ENFILE	La tabella del file di sistema è completa
EACCESS	Non c'è il permesso di creare un socket
ENOBUFS	Il sistema non ha sufficiente spazio buffer

Tabella 4 Codici di errori dei socket BSD sulla chiamata socket()

In corrispondenza di una chiamata di libreria di rete, come per esempio la chiamata `gethostbyname()`, la funzione `herror()` può essere utilizzata come riportato nel sottostante spezzone di codice:

```
s = gethostbyname("quadri.nis.garr.it");
if (s < 0)
fprintf(sterr,"gethostbyname: %s \n",herror(h_error));
```

Quando si verifica un errore in corrispondenza della funzione gethostbyname() la variabile esterna `h_error` contiene uno dei codici riportati nella tabella sottostante.

Valore in h_error	Causa dell'errore
HOST_NOT_FOUND	Il nome specificato non è conosciuto
TRY_AGAIN	Errore temporaneo: il server locale non può contattare il DNS
NO_RECOVERY	Errore non ricuperabile
NO_ADDRESS	Il nome specificato è valido ma ad esso non corrisponde un indirizzo IP

Tabella 5 Codici di errore dei socket BSD sulla funzione di libreria gethostbyname()