Lo Stack TCP/IP e i Socket

In questa lezione parleremo di architetture di protocolli di rete, ovvero di protocolli che realizzano la comunicazione su reti di calcolatori di tipo **packet-switching** (commutazione di pacchetto).
In particolare, vedremo che queste architetture condividono una struttura “a livelli” nell’intento di standardizzare separatamente le diverse funzionalità necessarie.
Infine daremo un primo accenno sull’architettura piu’ diffusa nel mondo, il cosiddetto “Stack TCP/IP”, e le primitive disponibili per utilizzarlo.

Letture consigliate per chi vuole ulteriormente approfondire:

• Appunti dalle lezioni di Sistemi3,
http:\\www.cs.unibo.it\~ghini\didattica\sistemi3\socket.html

• **Douglas E. Comer, “Internetworking with TCP/IP”, volume 1 “principles, protocols and architecture**”, second edition, ed Prentice Hall. (Ne esiste una versione in italiano del Gruppo Editoriale Jackson “Internetworking con TCP/IP, principi, protocolli, architettura.”

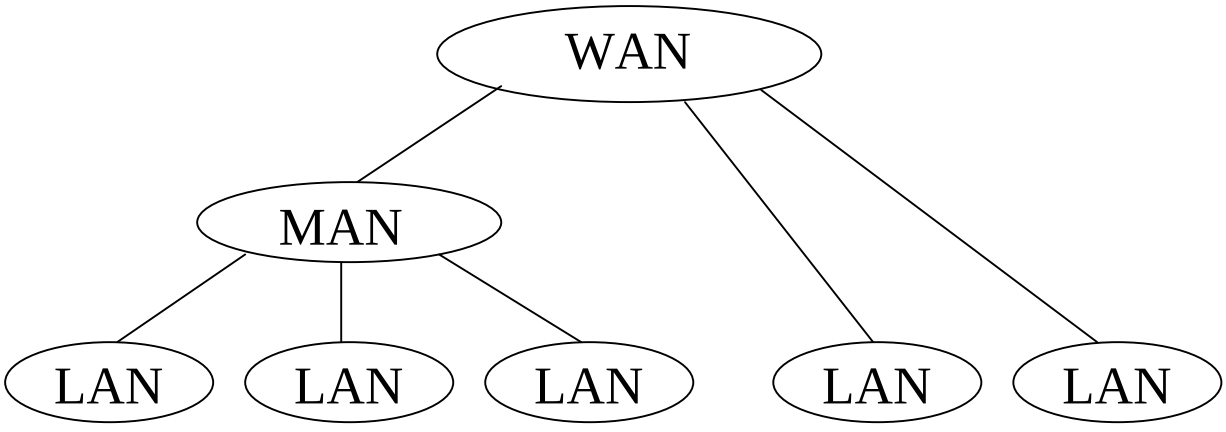
• W. Richard. Stevens, “UNIX Network Programming”, volume 1 second edition.”Networking APIs: Socket and XTI” ed. Prentice Hall

Una Tassonomia delle Reti

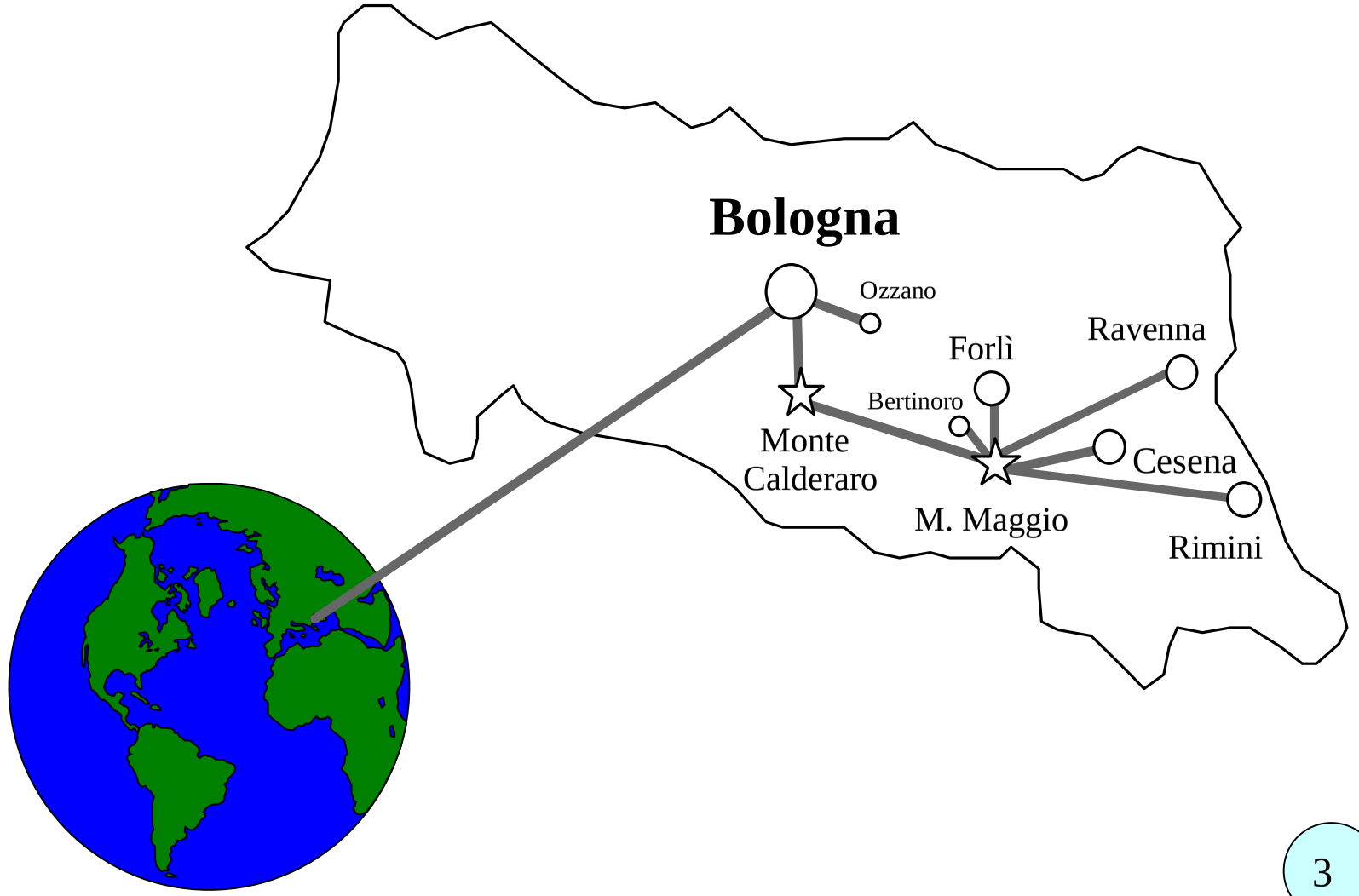
La tabella riporta una classificazione dei vari tipi di rete, in funzione dell'ambito operativo e delle distanze coperte.

AMBITO	DISTANZA COPERTA	RETE
Edificio	100 m	Reti Locali (LAN)
Comprensorio	1 km	Reti Locali (LAN)
Città	10 km	Reti Metropolitane (MAN)
Nazione	100 km	Reti Geografiche (WAN)
Continente	1000 km	Reti Geografiche (WAN)
Pianeta	10000 km	Reti Geografiche (WAN)

La figura qui sopra riporta la struttura di una rete di calcolatori di una ipotetica azienda. Essa è formata da una rete locale (LAN: Local Area Network) in ogni sede (edificio) della azienda; le LAN presenti all'interno di un'area metropolitana sono collegate tra loro tramite MAN (Metropolitan Area Network) e queste a loro volta tramite una rete geografica (WAN: Wide Area Network).



Un esempio di Wan:
la rete dell'Università di Bologna



L’OSI (Open System Interconnections)

L’ **OSI** è un progetto di ampio respiro formulato dall’ISO (International Standard Organization) alla fine degli anni ‘70 con lo scopo di fungere da **modello di riferimento** per le reti di calcolatori.
L’ISO doveva:
• **standardizzare la terminologia**;
• **definire** quali sono le **funzionalità** di una rete;
• servire come **base comune** da cui far partire lo sviluppo di standard per l’interconnessione di sistemi informatici;
• fornire un modello rispetto a cui **confrontare** le architetture di reti proprietarie.

Per gestire la complessità dei problemi, l’OSI ha adottato un **approccio a livelli** (layers): il problema della comunicazione tra due applicazioni è stato spezzato in un insieme di 7 livelli, ciascuno dei quali esegue funzioni ben specifiche.

L’OSI ha cercato di diventare più di un modello di riferimento. Infatti l’ISO ha standardizzato per OSI una serie di protocolli, da collocare nei vari livelli del modello, per formare una vera architettura di rete concorrenziale con altre, quali lo Stack TCP/IP.

I livelli 1 (Fisico) e 2 (Data Link) sono stati accettati e sono oggi degli standard, garantendo l’interoperabilità dei prodotti.

I protocolli di livello superiore incontrano più difficoltà a causa dell’alto impatto che la loro adozione avrebbe sui dispositivi di instradamento e sul software dei sistemi informativi.

Il modello ISO/OSI
Architettura a Livelli

Per ridurre la complessità del progetto, OSI introduce un'architettura a livelli (layered architecture) i cui componenti principali sono:

- i **livelli** (layers)
- le **entità** (entities)
- i **punti di accesso al servizio** (SAP: Service Access Points)
- le **connessioni** (connections)

In tale architettura, ciascun sistema è decomposto in un insieme ordinato di livelli, rappresentati per convenienza come una pila verticale. La figura seguente rappresenta i livelli che compongono il modello di riferimento ISO-OSI.

Tale approccio di progettazione a livelli è comune a tutte le moderne architetture di rete.

Il modello ISO/OSI

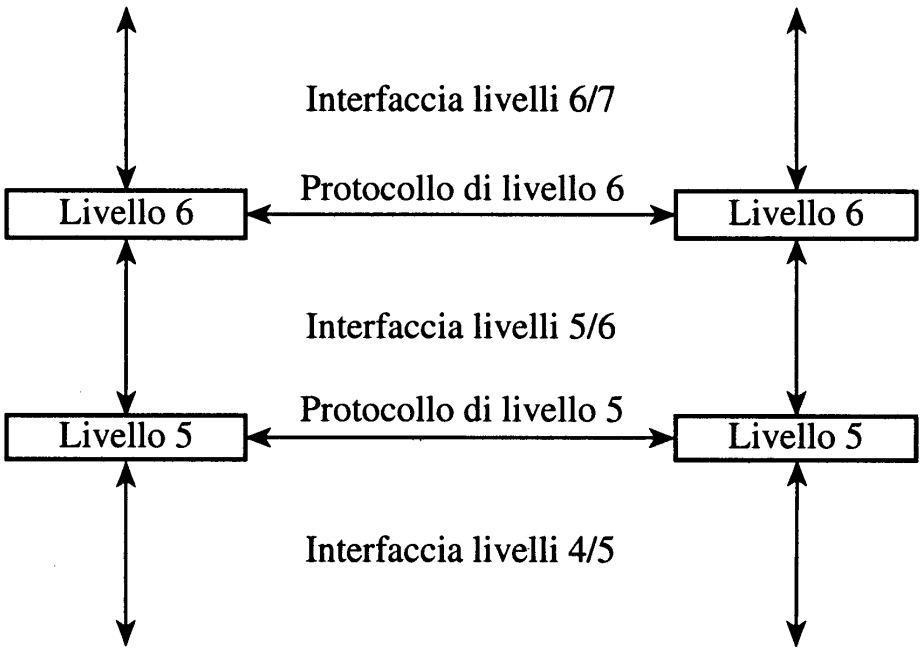
Comunicazioni tra livelli su sistemi diversi (1)

• Le operazioni specifiche di un livello, tra sistemi diversi, cioè la cooperazione tra entità appartenenti allo stesso livello ma collocate su sistemi diversi, sono realizzate da un insieme di protocolli (protocol).

• Affinchè due entità di livello N su sistemi diversi possano scambiarsi informazioni, una connessione deve essere stabilita nel livello N-1 usando un protocollo di livello N-1. Le informazioni sono contenute in delle **PDU** (Protocol Data Unit) cioè i messaggi nel formato di quel protocollo a quel livello.

•Tale connessione di livello N-1 è stabilita tra due SAP di livello N-1.

•Riassumendo, livelli N comunicano attraverso un protocollo di livello N: **ogni livello deve quindi mostrare un'interfaccia ben definita a quello immediatamente superiore e inferiore**, come in



Il modello ISO/OSI

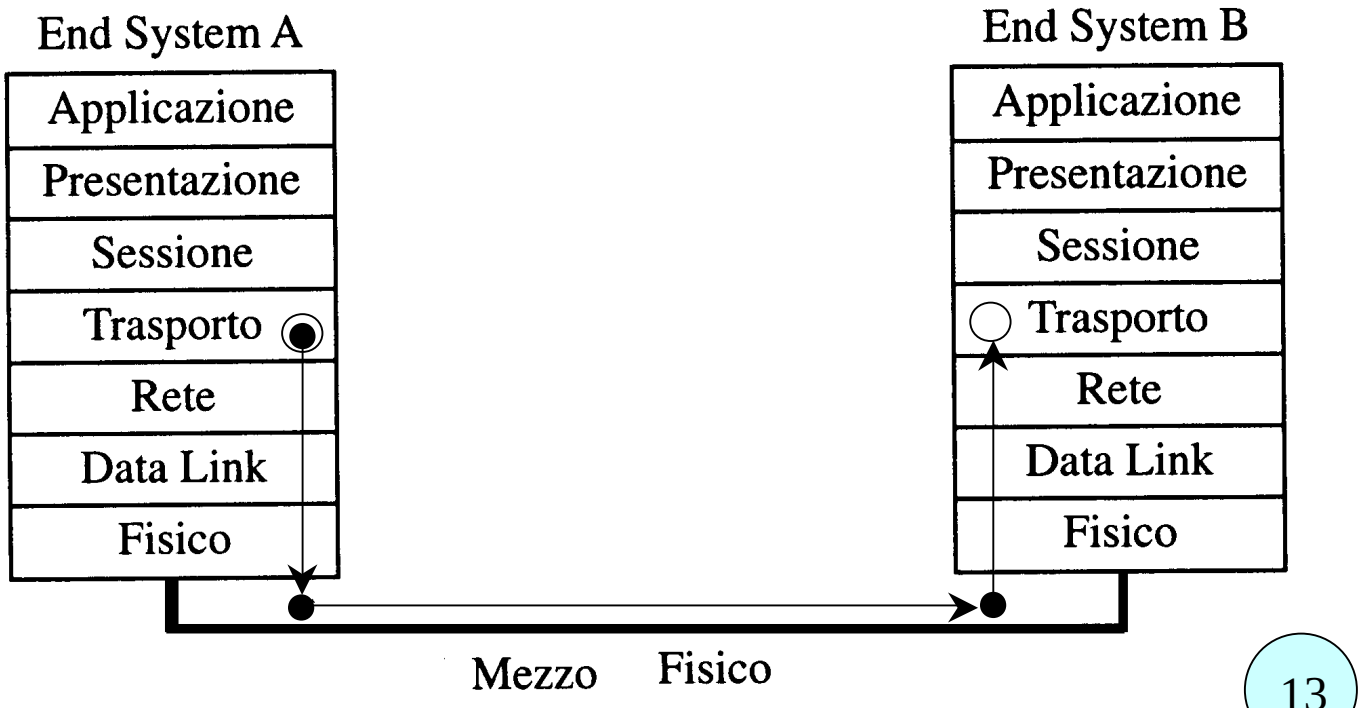
Comunicazioni tra livelli su sistemi diversi (2)

• Anche se è definito un protocollo di livello N, nessun dato è trasferito direttamente da un livello N all'altro su un diverso sistema.

• Infatti ogni livello passa dati e informazioni di controllo a quello sottostante, sino a quando si giunge al livello Fisico, che effettua la trasmissione verso l'altro sistema.

• L'interfaccia di un livello definisce quali operazioni primitive e quali servizi sono forniti da un livello ai livelli superiori.

In figura è rappresentato il meccanismo di comunicazione tra due entità di livello N= 4, con il messaggio che viene passato via via ai livelli inferiori (che aggiungono i loro header), viene trasmesso attraverso il mezzo fisico, e giunto al sistema di destinazione risale liberandosi degli header aggiunti, fino ad arrivare al livello stabilito.



Il modello ISO/OSI
I Livelli (1)

• Il livello 1: FISICO.
Il livello 1 del modello OSI si occupa di trasmettere sequenze binarie sul canale di comunicazione. A questo livello si specificano ad esempio le tensioni che rappresentano 0 e 1 e le caratteristiche dei cavi e dei connettori.

• Il livello 2: DATA LINK.
Il livello ha come scopo la trasmissione sufficientemente affidabile di pacchetti detti frame tra due sistemi contigui. Accetta come input dei pacchetti di livello 3 (tipicamente poche centinaia di bit) e li trasmette sequenzialmente. Esso verifica la presenza di errori aggiungendo delle FCS (Frame Control Sequence) e può gestire meccanismi di correzione di tali errori tramite ritrasmissione.

• Il livello 3: NETWORK.
Il livello 3 è il livello Network, che gestisce l'instradamento dei messaggi, ed è il primo livello (a partire dal basso) che gestisce informazioni sulla topologia della rete. Tale livello determina se e quali sistemi intermedi devono essere attraversati dal messaggio per giungere a destinazione. Deve quindi gestire delle tabelle di instradamento e provvedere ad instradamenti alternativi in caso di guasti (fault tollerance).

Il modello ISO/OSI
I Livelli (2)

• Il livello 4: TRASPORTO.
Il livello 4 del modello OSI fornisce un servizio di trasferimento trasparente dei dati tra entità del livello 5 (sessione). Si occupa di garantire un **servizio affidabile**. Deve quindi effettuare la frammentazione dei dati, la correzione degli errori e la prevenzione della congestione della rete. Il livello 4 è il livello più basso, a partire dall'alto, a trascurare la topologia della rete e la presenza di sistemi intermedi di instradamento, ed è quindi detto livello end-to-end.

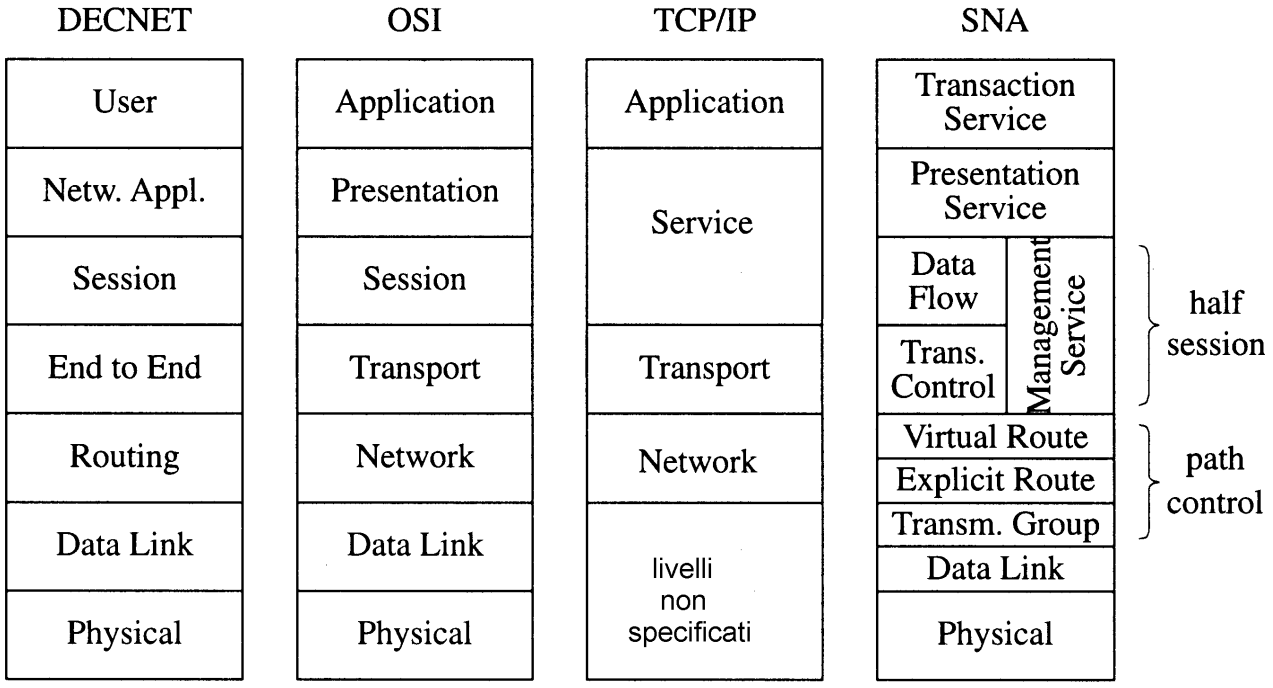
• Il livello 5: SESSIONE.
Organizza il dialogo tra due programmi applicativi, consentendo di aggiungere a connessioni end-to-end servizi più avazati.

• Il livello 6: PRESENTAZIONE.
Definisce formalmente i dati che gli applicativi si scambiano, come questi dati sono rappresentati localmente sul sistema, e come vengono codificati durante il trasferimento.

• Il livello 7: APPLICAZIONE.
Il livello 7 è il livello dei programmi applicativi, facenti parte del sistema operativo oppure scritti dall'utente, attraverso i quali l'utente utilizza la rete.

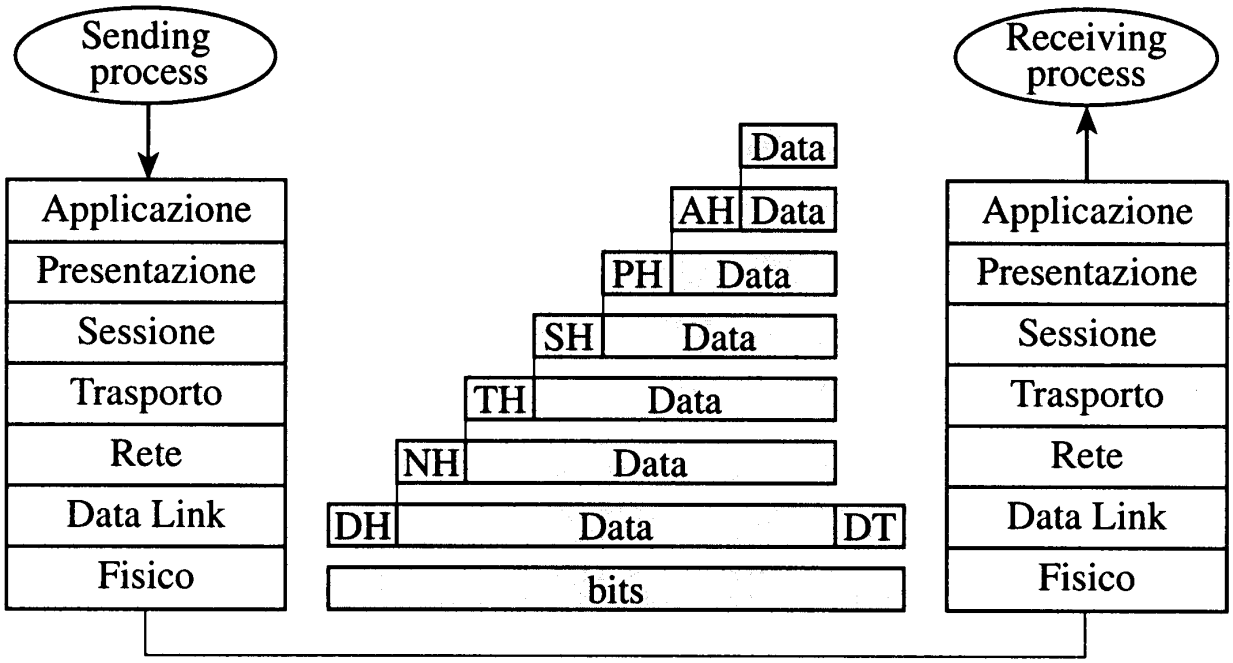
Principali Architetture di Rete

• L'insieme dei livelli, dei protocolli e delle interfacce definisce un'architettura di rete. Le architetture di rete più note sono:
• SNA (System Network Architecture) architettura della rete IBM;
• DNA (Digital Network Architecture), meglio nota come DECnet, la rete della Digital Equipment Corporation;
• Internet protocol Suite, meglio nota con il nome TCP/IP, è la rete degli elaboratori UNIX e rappresenta uno standard "de facto" attualmente impiegato per la rete Internet.
•OSI (Open System Architecture), che è lo standard "de iure" prodotto dall'ISO.



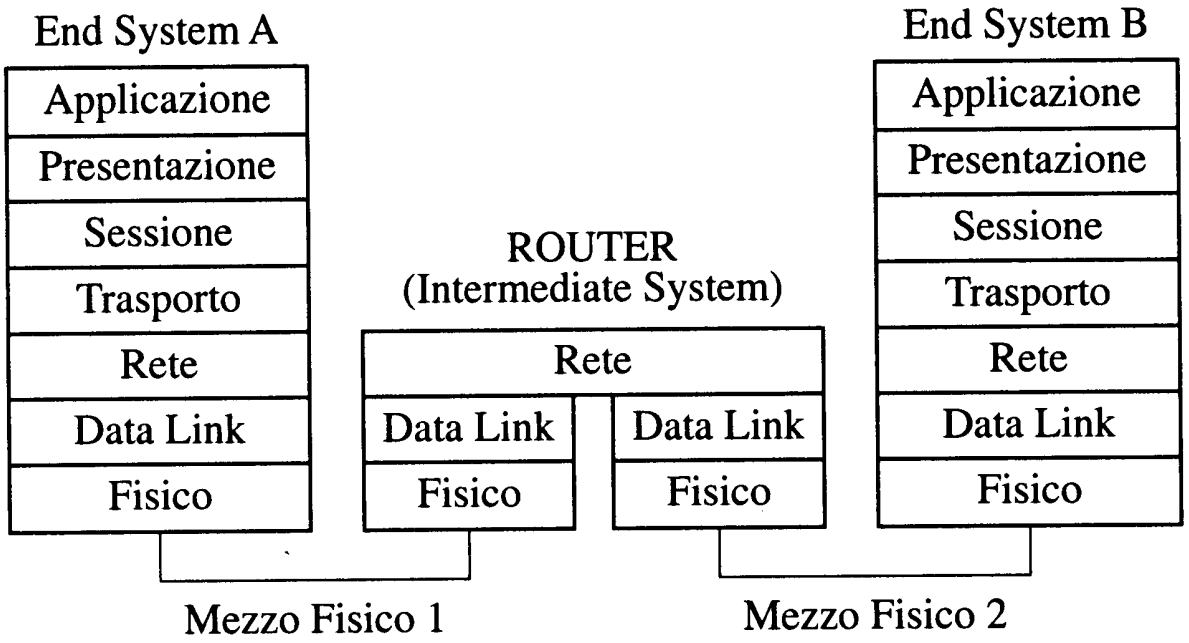
Incapsulamento dei Pacchetti

La trasmissione dei dati avviene quindi:
• attraverso una serie di passaggi da livelli superiori a livelli inferiori nel sistema che trasmette,
• poi attraverso mezzi fisici di comunicazione,
• infine attraverso un'altra serie di passaggi, questa volta da livelli inferiori a livelli superiori.
Notare come a livello 2, sia necessario aggiungere in coda un campo che identifica la fine del pacchetto prima di passare lo stesso al livello che utilizza il mezzo trasmissivo.



Sistemi Intermedi (1)

Non sempre lo scambio delle informazioni avviene direttamente tra i due sistemi finali che contengono le applicazioni (ES: End Systems). Può anche implicare l'attraversamento di alcuni sistemi intermedi (IS: Intermediate Systems).
In questi Intermediate Systems esistono delle entità che assumono la funzione di Router (in senso esteso), ovvero **entità che instradano le informazioni**.
Tali entità possono essere collocate a diversi livelli del modello OSI, ed allora gli Intermediate Systems assumono nomi diversi a seconda del livello in cui avviene l'instradamento dei dati. Si parla allora di **repeater** a livello 1, **bridge** a livello 2, **router** a livello 3 ed infine **gateway** a livello 7.
Qui di seguito è rappresentata la collocazione di un Router nel modello OSI.



Sistemi Intermedi (2)

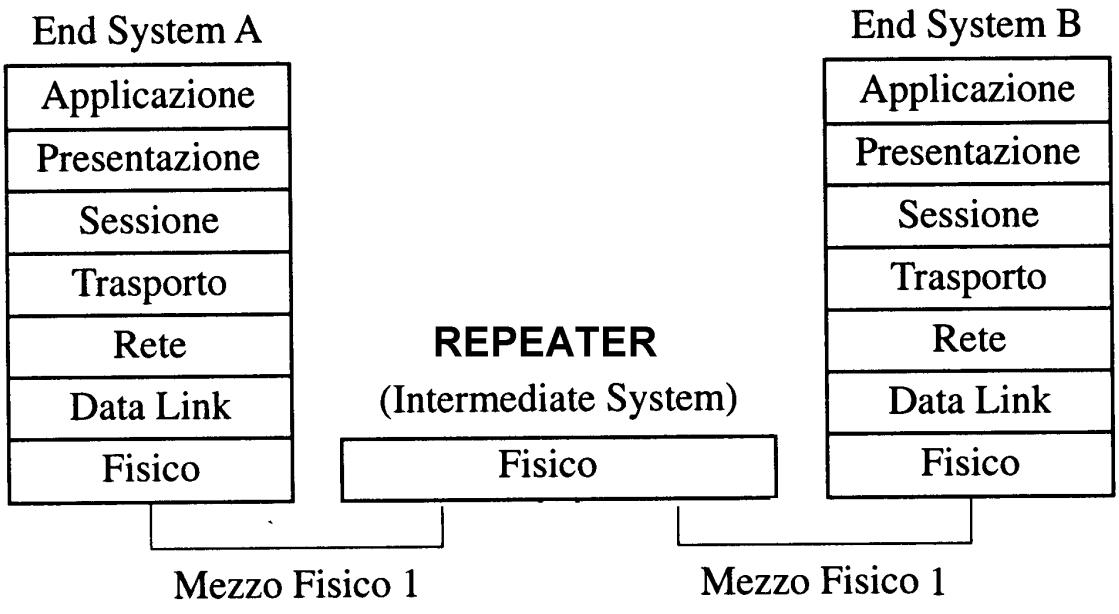
Qui di seguito è rappresentata la collocazione di un Repeater nel modello OSI. Il livello Fisico non effettua correzione dei dati ricevuti, compito che invece spetta al livello Data Link.

• Il segnale che il Repeater riceve da un lato viene amplificato e propagato all'altro lato, quindi viene propagato anche il rumore che può essersi prodotto durante la trasmissione dal sistema A al Repeater, e a tale rumore si aggiungerà il rumore prodotto nella trasmissione dal Repeater al sistema B.

• Di conseguenza il livello Fisico del sistema B riceve un segnale afflitto dalla somma dei rumori che si sono prodotti durante la trasmissione sui due tratti di percorso, e quindi aumenta la probabilità che si riscontri un errore nei dati trasmessi.

• La correzione potrà essere effettuata solo quando il segnale giungerà al sistema 2, ma l'eventuale ritrasmissione dei dati dovrà attraversare nuovamente i due tratti di rete.

• E' per questo motivo che bisogna rudurre al minimo il numero di repeater in un percorso.



Protocolli Connessi e Non Connessi

Per tutti i livelli superiori al livello fisico sono definite due modalità operative: una **modalità connessa** e una **modalità non connessa**
Un dato livello può fornire al livello superiore servizi di tipo connesso, non-connesso o entrambi. Questa è una scelta progettuale che varia per ogni livello, da architettura ad architettura.

In un servizio non connesso la spedizione di un pacchetto è simile alla spedizione di una lettera ordinaria con il sistema postale. Tutto avviene in una sola fase lasciando cadere la lettera nella buca delle lettere. La lettera deve contenere sulla busta l'indirizzo completo del destinatario. Non vi è alcun riscontro diretto che la lettera giunga a destinazione correttamente.

In un servizio connesso lo scambio di dati tramite pacchetti ricorda le frasi scambiate tra due interlocutori al telefono. Vi sono tre momenti principali:
- creazione della connessione (il comporre il numero telefonico e il "pronto" alla risposta);
- trasferimento dei dati (la conversazione telefonica);
- chiusura della connessione (posare il microtelefono).

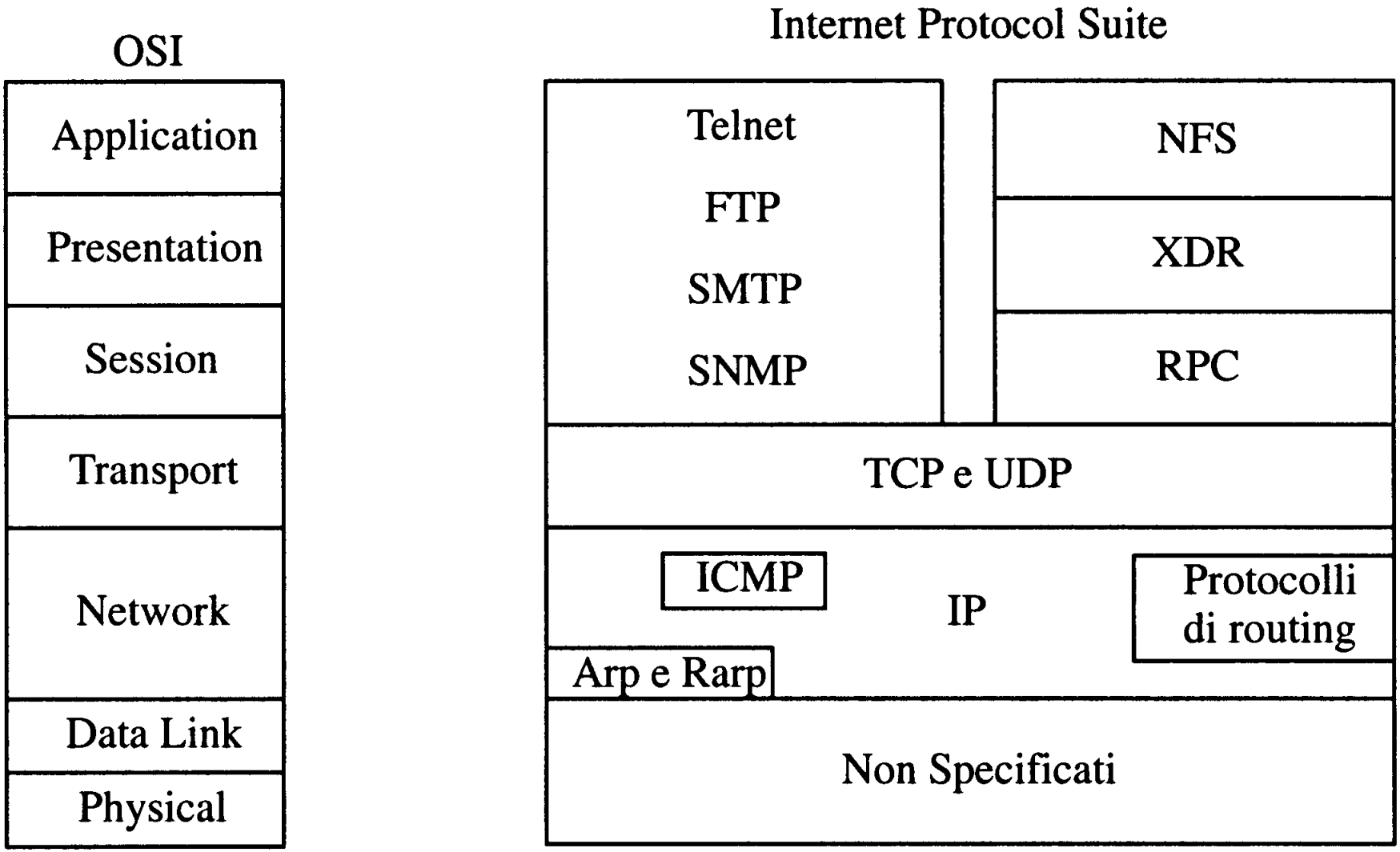
Protocolli Connessi e Non Connessi
La modalità Connessa

• Nella modalità connessa lo scambio di dati avviene tramite le tre fasi viste prima. Durante la fase di creazione della connessione (initial setup) due peer-entities concordano che trasferiranno delle PDU.
• Solo durante tale fase devono essere specificati gli indirizzi completi del mittente e del destinatario: successivamente le entità coinvolte specificheranno nei pacchetti soltanto l'identificativo della connessione stabilito durante la prima fase.
• Un servizio connesso fornisce una modalità di trasferimento delle PDU affidabile e sequenziale. Per tutta la durata della connessione le PDU inviate sono ricevute correttamente nello stesso ordine.
• Se qualcosa non funziona correttamente, la connessione può essere riavviata (reset) o terminata (released).
• Per verificare che tutte le PDU inviate giungano a destinazione correttamente un servizio connesso utilizza degli schemi di numerazione dei pacchetti e di verifica dell'avvenuta corretta ricezione (ACK: acknowledgement).
• Quindi un protocollo connesso è in generale in grado non solo di rilevare la presenza di errori, ma anche di correggerli tramite ritrasmissioni.

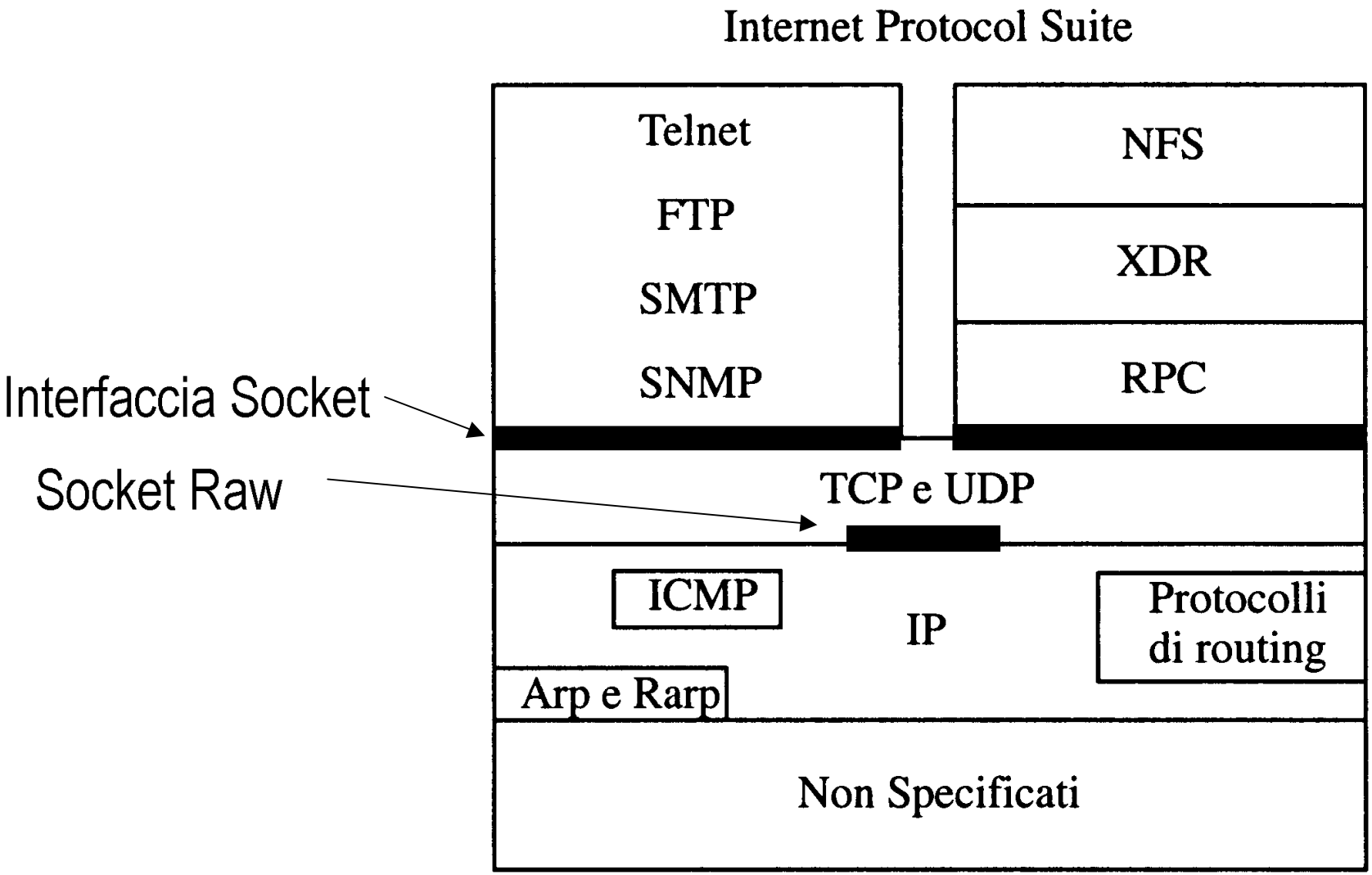
Protocolli Connessi e Non Connessi
La modalità Non Connessa

• Con una modalità non connessa la comunicazione ha luogo in una fase singola.
• Il pacchetto è inviato e deve contenere l'indirizzo completo del destinatario.
• Non essendo i pacchetti organizzati in una connessione, un pacchetto non può fare riferimento ad altri pacchetti trasmessi precedentemente o in seguito.
• Quindi un protocollo non connesso può solo rilevare la presenza di errori (scartando quindi le PDU errate), ma non correggerli in quanto non si possono realizzare meccanismi di ritrasmissione (in un pacchetto non è possibile fare riferimento ad altri pacchetti).
• Un protocollo non connesso è in generale più efficiente di un protocollo connesso, specialmente se bisogna trasferire piccole quantità di dati: in quest'ultimo caso infatti l'overhead della creazione e distruzione della connessione è rilevante.
• Un protocollo non connesso (detto anche datagram), non potendo garantire l'affidabilità del trasferimento dati, necessita che almeno un protocollo di livello superiore sia di tipo connesso.

Architettura TCP/IP e confronto con OSI



Interfaccia Socket



Socket in ambiente Windows ed NT

• In ambiente Windows ed NT sono state messe a disposizione delle librerie che definiscono un'interfaccia socket simile (ma non uguale) a quella disponibile per i sistemi UNIX. Sono note come le librerie winsock.

• Le winsock sono un aggiunta in ambiente windows 95 mentre sono native in ambiente NT.

• Le winsock possono essere utilizzare **aggiungendo** a livello di codice C l'inclusione ad un file:

```
#include <winsock.h>
```

mentre **andranno eliminate** le inclusioni tipiche dei sistemi unix quali:

```
#include <sys/socket.h>
#include <netinet/in.h>
```

• Per quanto riguarda le **librerie**, in ambiente windows dovranno essere utilizzate le librerie per le winsock, diverse da ambiente ad ambiente. In particolare in ambiente NT si dovrà utilizzare la libreria **ws2_32.lib**, mentre in ambiente Windows 95 (come nei laboratori per il 1° biennio) si dovrà utilizzare la libreria **wsock32.lib**

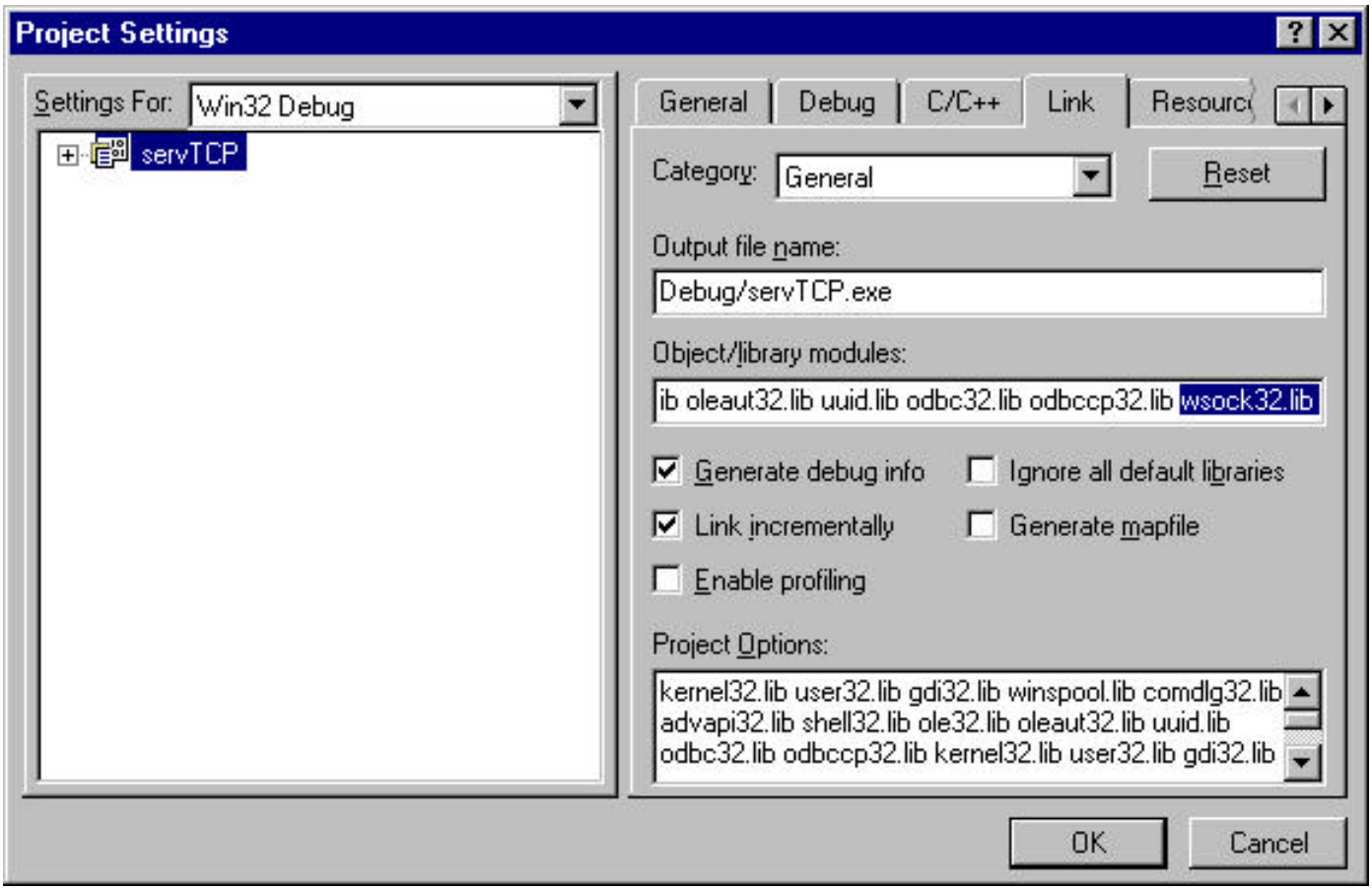
• All'interno dell'ambiente Microsoft C++ , le librerie dovranno essere specificate (aggiunte) nel menu:

Project -> Setting -> Link

alla voce:

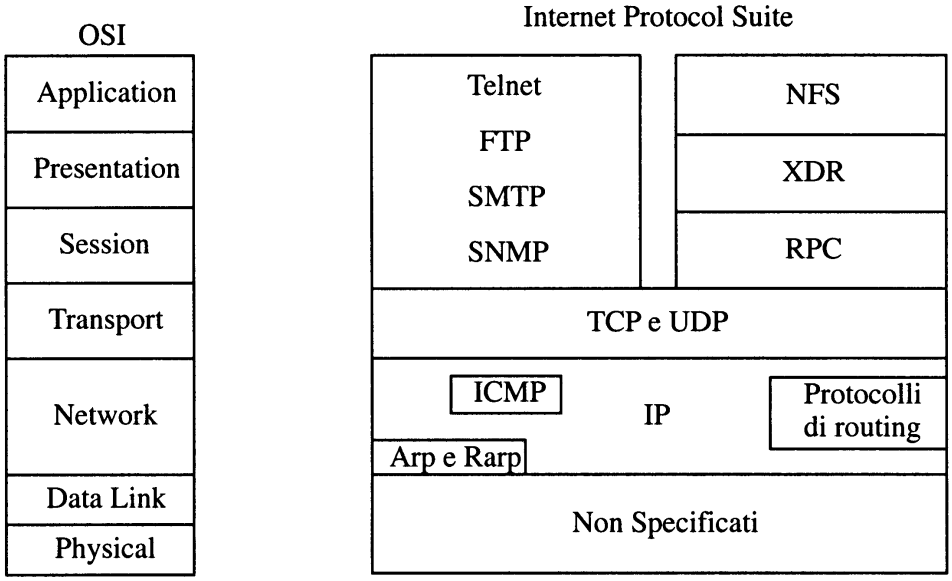
Object/Library Modules

Specifica delle librerie winsock per 95



Lo Stack TCP/IP: Le Basi

I livelli TCP/IP hanno questa relazione con i livelli di OSI.



Lo stack di protocolli TCP/IP implementa un livello network (livello 3) di tipo:

- packet-switched;
- connectionless.

Il livello più basso (corrispondente ai livelli 1 e 2 di OSI) non è specificato dall'architettura, che prevede di utilizzare quelli disponibili per le varie piattaforme HW e conformi agli standard.

Per quanto riguarda le reti in ambito locale (LAN), lo standard riconosciuto per i livelli 1 e 2 è rappresentato dal progetto IEEE 802, che è stato riconosciuto anche da OSI.

Per capire il funzionamento dei protocolli TCP/IP, nel seguito considereremo proprio la situazione di una rete locale. Risulta necessario perciò soffermarsi brevemente sulla strutturazione delle LAN.

Le LAN (Local Area Network)

DEFINIZIONE: una LAN è un sistema di comunicazione che permette ad apparecchiature indipendenti di comunicare tra loro, entro un **area delimitata**, utilizzando **un** canale fisico a **velocità elevata** e con **basso tasso d'errore**.

• Le LAN hanno quindi sempre **un solo canale trasmissivo** ad alta velocità condiviso nel tempo da tutti i sistemi collegati. Quando un sistema trasmette diventa proprietario temporaneamente (per la durata di uno o pochi pacchetti) dell'intera capacità trasmissiva della rete.

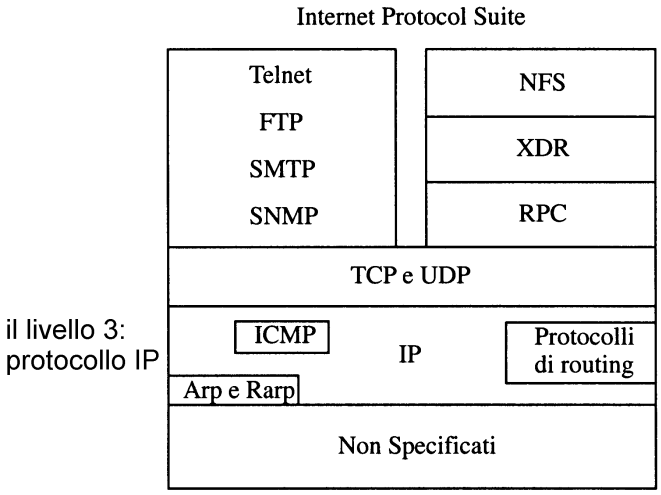
• La trasmissione è sempre di tipo **broadcast**, ovvero un sistema trasmette e tutti gli altri ricevono. Tale organizzazione ha enormi vantaggi, ma impone anche alcune complicazioni: è **necessaria la presenza di indirizzi di livello 2** per stabilire chi sono il reale destinatario e il mittente della trasmissione e occorre arbitrare l'accesso all'unico mezzo trasmissivo tra tutti i sistemi che hanno necessità di trasmettere.

• L'unico canale trasmissivo presente deve anche essere caratterizzato da un **basso tasso di errore**. Questo è ottenibile abbastanza facilmente in un'area delimitata usando mezzi trasmissivi di buona qualità. L'effetto ottenuto è quello che le LAN, essendo intrinsecamente affidabili, non hanno la necessità di correggere gli errori a livello 2 OSI e quindi normalmente utilizzano **protocolli di livello 2 connectionless** ad alte prestazioni.

Il livello Network del TCP/IP.

Il protocollo IP (versione 4)

L'architettura TCP/IP (il cui nome più preciso è **Internet Protocol Suite**) è formata da diversi componenti, che si posizionano nello stack dei protocolli a partire dal livello 3 (network).
I protocolli appartenenti a questa architettura sono specificati tramite standard denominati RFC (Request For Comments) disponibili in rete. Ad es. l'RFC 791 specifica il protocollo IP.



Il protocollo IP (**Internet Protocol**) è il protocollo principale del livello 3 dell'architettura TCP/IP. Si tratta di un protocollo semplice, di tipo datagram ovvero **senza connessione** e **non affidabile**;

• **non affidabile**: il pacchetto inviato può essere perso, duplicato ritardato o consegnato fuori sequenza, ma il protocollo IP non informerà nè il trasmettitore nè il ricevitore.

• **senza connessione**: ogni pacchetto viene trattato in maniera indipendente dagli altri, pacchetti diversi aventi stesso mittente e stesso destinatario possono seguire percorsi diversi, alcuni possono essere consegnati ed altri no. Se le risorse della rete lo consentono il pacchetto viene portato a destinazione, in caso contrario verrà scartato.

Il protocollo IP versione 6

Attualmente lo standard per il livello network dello stack TCP/IP è rappresentato dal protocollo IP versione 4, ma già dal 1995 è stata proposta una nuova versione del protocollo IP, nota col nome di IP versione 6 (RFC 1883: Internet Protocol, Version 6 (IPv6) Specification, December 1995 R. Hinden, S. Deering).
IP versione 4 versione soffre di almeno 3 problemi principali, che IPv6 vuole correggere:

• **numero degli indirizzi IP disponibili ormai insufficiente**: gli indirizzi IP sono composti da 4 byte (32 bit) e a causa del grande incremento del numero degli hosts nel mondo la disponibilità degli indirizzi è in forte calo. Il protocollo IPv6 prevede invece indirizzi formati da 16 byte (128 bit) e quindi rende disponibili un numero enorme di indirizzi.

• **traffico gestito esclusivamente in modo best-effort**: in IPv4 tutti i pacchetti sono trattati allo stesso modo dai router, anche se esiste nell'header dei pacchetti IPv4 un campo priorità, che però non viene utilizzato. Con IPv6 si vuole definire delle classi di servizio a cui assegnare priorità diverse. Si vuole anche gestire la comunicazione con un meccanismo simile ad un protocollo con connessione, cioè implementando un flusso di dati.

• **sicurezza**: in IPv6 saranno rese standard e disponibili alcune primitive per l'autenticazione e la cifratura dei dati.

Nonostante queste prospettive, il protocollo IP versione 6 è ancora poco diffuso, e rimane ancora a livello di sperimentazione, forse perchè l'adozione del nuovo protocollo costringerebbe a modificare fortemente gli apparati di rete esistenti, con un grande dispendio di denaro. Esistono a tuttoggi, in un oceano IPv4, solo delle **isole** in cui si parla IPv6.

Nel seguito parleremo di IPv4, che rappresenta ancora lo standard più diffuso.

Funzioni del protocollo IPv4

Il protocollo IP svolge le seguenti funzioni:

• **distingue** ogni hosts, o meglio **ogni scheda di rete mediante un identificatore**, detto **indirizzo IP**. Un indirizzo IP di tipo single o **unicast** identifica un unico host, ma uno stesso host può avere più indirizzi IP unicast, tanti quante sono le schede di rete che possiede. Si parla allora di MultiHomed Systems. Ad es. i router hanno più indirizzi, perchè dovendo fungere da centri di smistamento dispongono di più schede di rete. Un host può comunque disporre di più schede di rete anche senza essere un router, cioè anche se non effettua un servizio di instradamento per pacchetti destinati ad altri hosts, ma dovrà prevedere una politica che definisca quale scheda di rete utilizzare per inviare i dati.

• **riceve i dati** (una sequenza di PDU) **dal livello trasporto** (4).

• **incapsula ciascuna PDU ricevuta in pacchetti** di dimensione massima 64 Kbyte (normalmente circa 1500 byte), **aggiungendovi un proprio header** (o intestazione).

header del Datagram IP	area dati del Datagram IP (PDU del livello trasporto)

• eventualmente **frammenta I dati dei pacchetti** all'inizio o durante il trasporto, per inserirli nei frame di livello 2.

• **instrada i pacchetti** sulla rete,

• **effettua la rilevazione**, non la correzione, **degli errori**,

• **alla destinazione**, se necessario, **riassembla i frammenti** ricostruendo i pacchetti originali,

• **estrae dai pacchetti i dati** (PDU) **del livello trasporto**,

• **consegna al livello trasporto i dati nell'ordine in cui sono arrivati a destinazione**, che può essere diverso dall'ordine in cui sono partiti.

Il pacchetto IPv4: lo HEADER

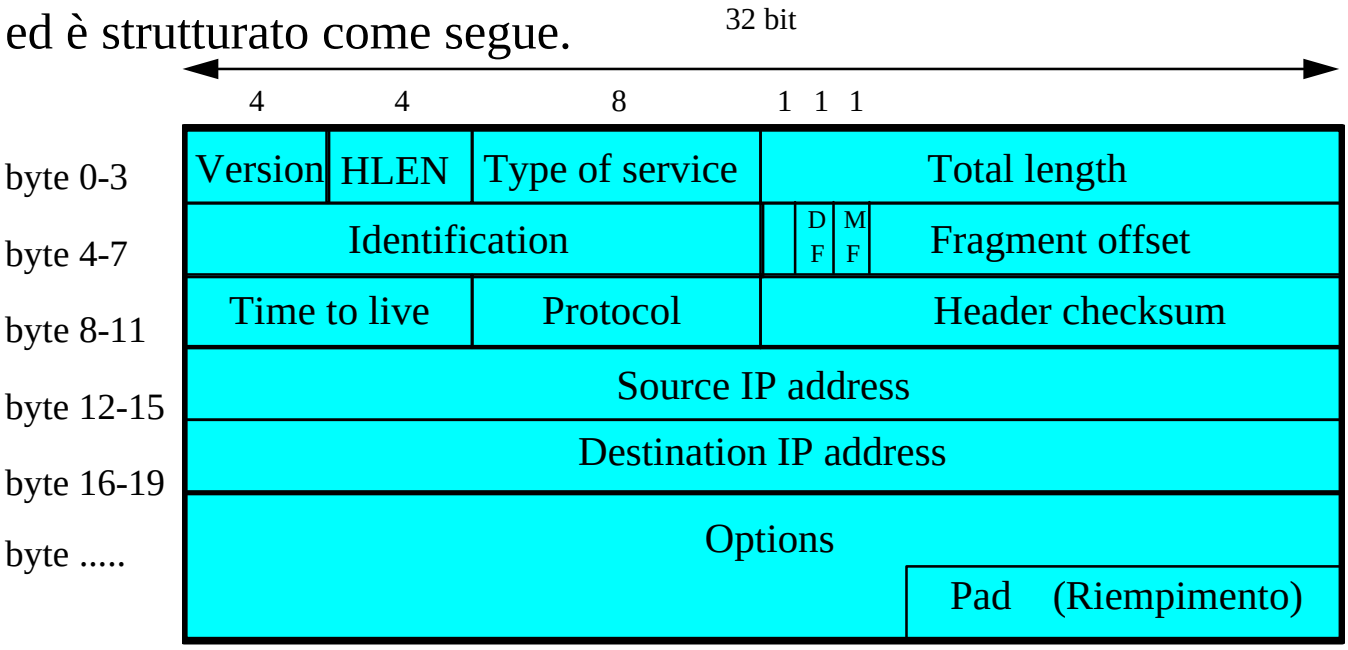
Un pacchetto IP è costituito da un header e da una parte dati, che rappresenta la porzione di dati del livello trasporto da trasferire.

header del Datagram IP	area dati del Datagram IP (PDU del livello trasporto)

L'header IP ha:

- una prima parte fissa di 20 bye,
- una seconda parte, opzionale, di lunghezza variabile, ma sempre multiplo di 4 byte

ed è strutturato come segue.



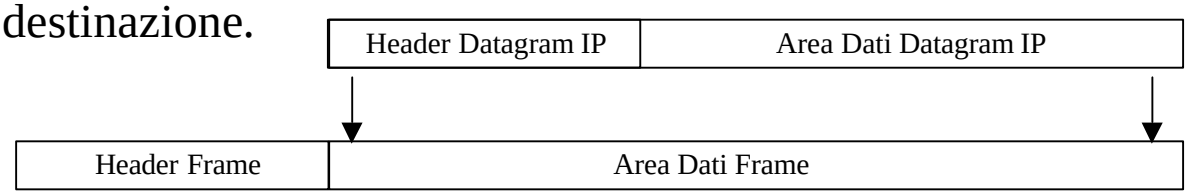
• I campi **Source IP address** e **Destination IP address** contengono gli indirizzi IPv4 a 32 bit della provenienza originale e della destinazione finale del datagramma IP. Tali indirizzi quindi non cambiano mai durante tutto il percorso, comunque venga instradato il pacchetto, e comunque venga frammentato.

• Il campo **Time to Live**, indica il numero massimo di router che il pacchetto può attraversare prima di essere scartato. Viene decrementato di uno ogni volta che viene ritrasmesso da un router. Il pacchetto viene scartato quando TTL è zero. Serve ad impedire che un pacchetto giri in rete in eterno.

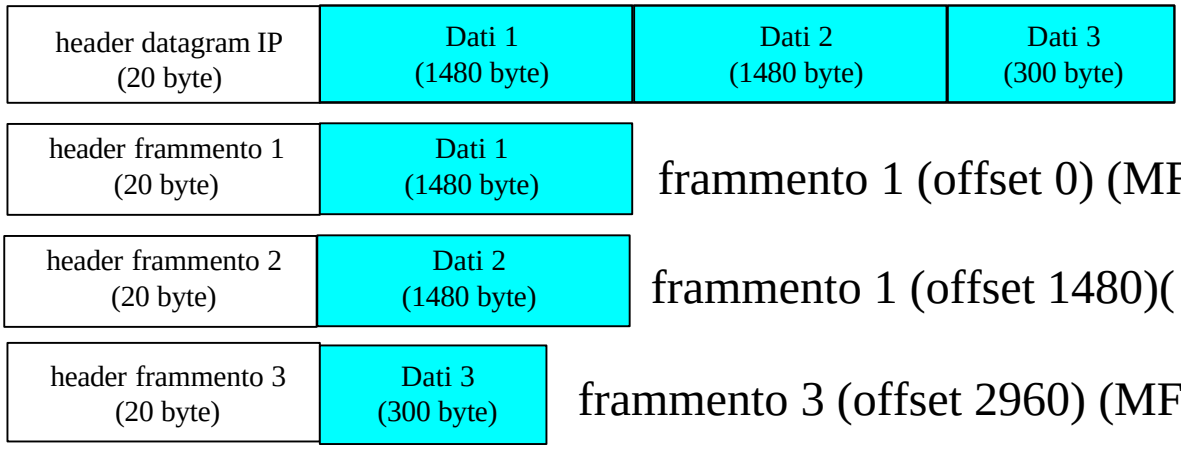
La Frammentazione dei pacchetti IP (1)

L'hardware di ogni tipo di rete impone un limite superiore alla dimensione del frame di livello 2, e quindi anche alla quantità di dati di livello 3 che possono essere trasportati in un unico frame a livello 2. La dimensione massima di dati di livello 3 che possono essere trasportati in un frame del Data Link viene chiamata **Massima Unità di Trasferimento (MTU: Maximum Transfer Unit),** ed è caratteristico di ogni tipologia di rete. Per Ethernet MTU=1500 bytes.

• Se la porzione dati di un datagram IP (più la dimensione dell'header IP) è più piccola della MTU della rete sottostante, il datagram IP potrà essere inserito completamente in un frame di livello 2 e inviato a destinazione.



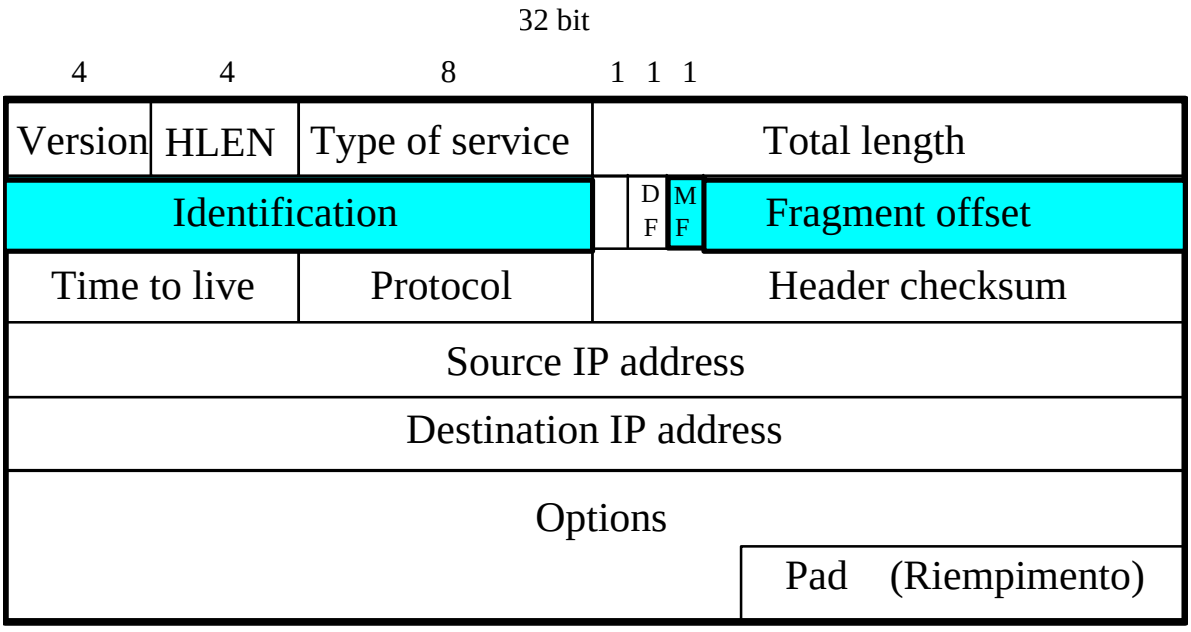
• Se invece la porzione dati del datagram IP (più la dimensione dell'header IP) è più grande della MTU della rete sottostante, **la porzione dati** dovrà essere spezzata in più pezzi che verranno incapsulati in datagram IP (detti frammenti) più piccoli della MTU, e ciascun frammento dovrà essere inserito in un frame diverso e verrà spedito separatamente dagli altri verso la destinazione finale, dove il protocollo IP provvederà a rimettere assieme i diversi frammenti e a ricostituire il datagram originale. Nell'esempio un pacchetto IP con 3260 byte di dati frammentato per una MTU di 1500 byte.



La Frammentazione dei pacchetti IP (2)

il problema della frammentazione si propone ogni volta che nel percorso seguito dai pacchetti IP, si deve attraversare una porzione di rete avente una MTU minore della porzione di rete precedentemente attraversata, sempre se la dimensione dei pacchetti IP è maggiore della MTU più piccola.
Il router preleva allora la porzione dati del datagram IP e lo spezza in più porzioni, in modo che ciascuna (aggiungendovi l'header) stia in un frame, e in modo che ogni frammento dei dati, tranne l'ultimo, abbia dimensione multipla di 8 byte, perchè così è definito il campo offset dell'header IP.
L'ultimo pezzo in genere sarà il più corto e verrà identificato come ultimo settandovi a zero il flag **MoreFragment**, ad indicare che è l'ultimo frammento. Negli altri frammenti **MF** è settato a 1.

Il protocollo IP usa tre campi dell'header per controllare il meccanismo della frammentazione e permettere il riassemblaggio dei datagram frammentati. Questi campi sono **Identificatio**n (16 bit), **Fragment Offset** (15 bit) e il flag **More Fragment** (**MF**).



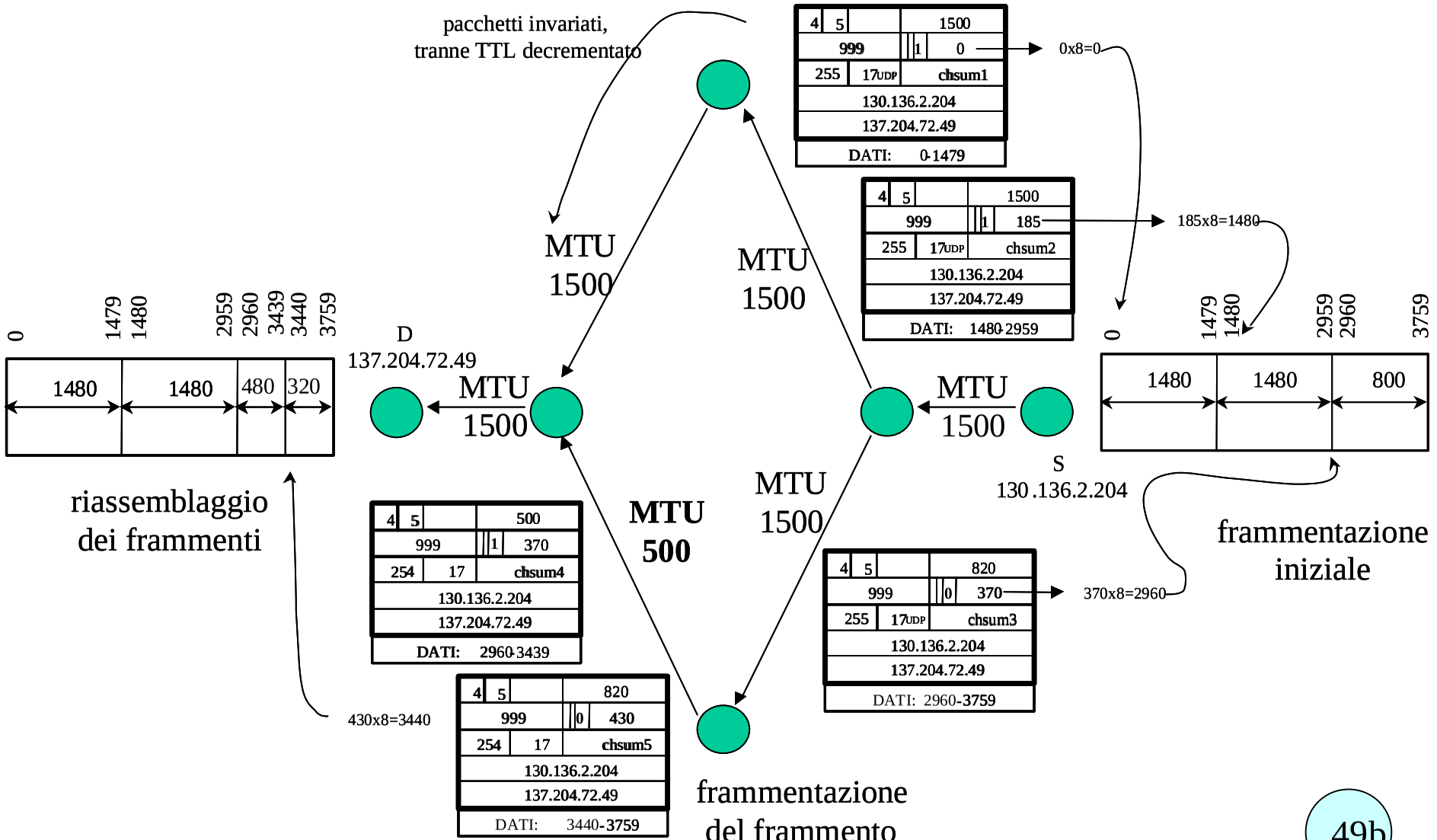
La Frammentazione dei pacchetti IP (3)

L'header del datagram originale verrà copiato integralmente nei frammenti (con qualche modifica per il campo Options) e in più verrà cambiato il campo **Fragment Offset che indica il punto dell'area dati del datagram originale in cui comincia la porzione di dati trasportata nel frammento**. Tale offset è pensato come un multiplo di 8 byte. Se l'offset indica ad es. 185, il frammento porta la porzione di dati che inizia nella posizione 185*8=1480 byte.

• **Tutti i frammenti** sono caratterizzati dall'avere lo stesso identificatore (il campo **Identification**) del datagram originale. Tale numero è assegnato univocamente dal trasmettitore (che mantiene un contatore dei datagram IP trasmessi), e la coppia (IP Provenienza, Identification) rende univocamente identificabile un certo datagram IP, e tutti i suoi frammenti.

Esempio di Frammentazione

Un datagram di 3760 byte, inviato da S=130.136.2.204 a D=137.204.72.49 , tutti i tratti di rete hanno MTU di 1500 byte, tranne uno che ha MTU pari a 500 byte



Il Riassemblaggio dei Frammenti

Dopo la frammentazione, ogni frammento viaggia separatamente dagli altri fino alla destinazione finale. Solo alla fine del viaggio avrà luogo il riassemblaggio dei frammenti, nel tentativo di ricostruire il datagram Originale.

Il ricevitore riconosce di avere ricevuto un frammento e non un datagram intero in due modi:
• il pacchetto IP ricevuto ha un **offset uguale a zero**, ma ha il flag **More Fragment settato ad uno** (è il primo frammento).
• il pacchetto IP ricevuto ha un **offset diverso da zero** (è un frammento successivo). Se il More Fragment è zero è l'ultimo frammento.

Il protocollo IP del ricevente identifica univocamente i frammenti di uno stesso datagram mediante la coppia (**IP trasmettitore, Identification del datagram**).

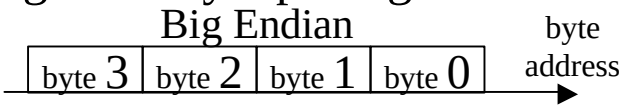
Il ricevente non conosce la dimensione del datagram originale perchè ogni frammento mantiene nel campo Total Length la lunghezza del frammento stesso, e non quella del datagram originale. Solo quando riceverà il frammento con il flag **More Fragment settato a zero** (che indica l'ultimo frammento del datagram originale), si potrà capire la dimensione totale del datagram originale sommando all'offset dell'ultimo frammento la lunghezza dei dati trasportati nell'ultimo frammento.

Se un solo frammento viene perso, è impossibile ricostruire il datagram IP originale.

Per evitare di aspettare inutilmente un frammento perso, il ricevitore nel momento in cui riceve un primo frammento inizializza un timer. Se il timer scade prima che tutti i frammenti siano giunti a destinazione il ricevitore butta via tutti i frammenti.

L'ORDINE dei BYTE in RETE

E' necessario creare un insieme di protocolli di rete che non dipendano dall'architettura del computer o della scheda di rete. Non tutte le architetture di calcolatori memorizzano i dati nello stesso modo. In particolare gli interi di 32 bit (le dimensioni dell'indirizzo IP) sono memorizzati in due modi diversi: le macchine Little Endian mantengono il byte meno significativo nell'indirizzo di memoria più basso, le macchine Big Endian mantengono il byte più significativo nell'indirizzo di memoria più basso.

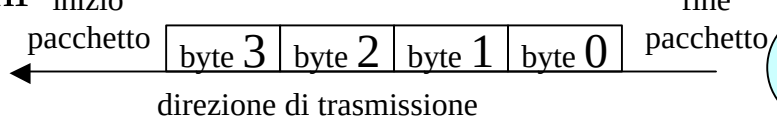


• Per evitare confusione, ovvero per evitare che un indirizzo IP possa venire scritto in due modi diversi, i protocolli TCP/IP definiscono un ordine di byte standard della rete, che tutte le macchine devono usare per i campi dei protocolli IP, ovviamente non per la parte dati.

• In trasmissione, prima di scrivere ad es. un'indirizzo IP nel campo destinazione del pacchetto IP, l'host deve convertire il numero IP dalla sua propria rappresentazione alla rappresentazione standard per la rete.

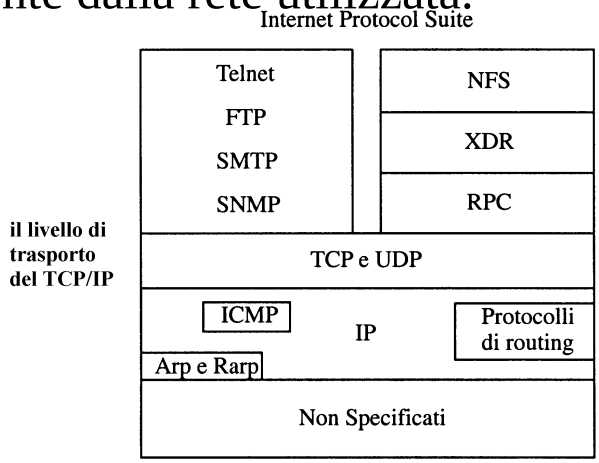
• In ricezione, prima di valutare l'indirizzo IP contenuto nel campo provenienza del pacchetto IP, deve convertire tale campo dal formato di rete standard nel formato interno della macchina.

• Internet stabilisce come **ordine standard per gli interi a 32 bit**, quello che prevede che **i byte più significativi siano trasmessi per primi** (stile Big Endian). Guardando viaggiare i dati da una macchina all'altra, un intero a 32 bit comincia ad essere trasmesso dal byte più significativo, cioè col byte più significativo più vicino all'inizio del pacchetto. Le librerie socket forniscono per le conversioni delle funzioni che sono: ntohs,ntohl, htons,htonl.



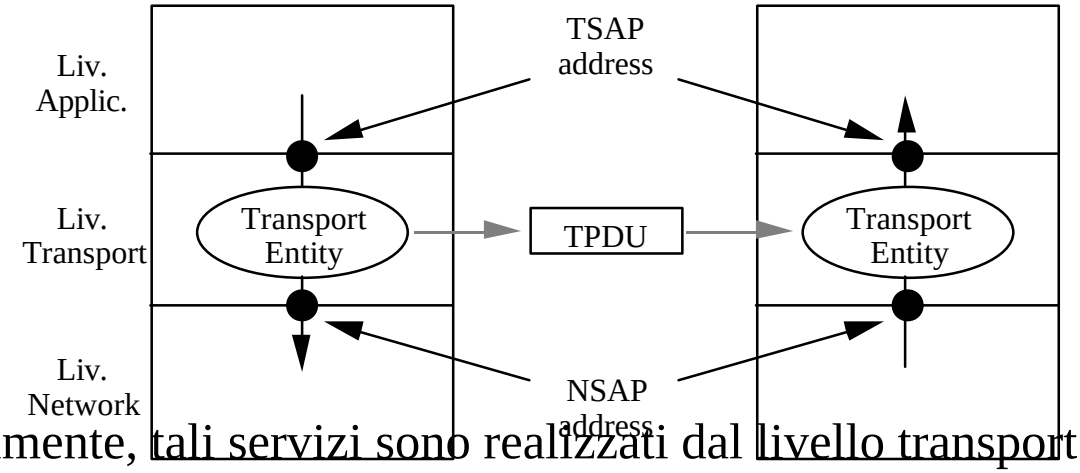
Il livello di Trasporto del TCP/IP

Il compito del livello transport (livello 4) è di fornire un trasporto efficace dall'host di origine a quello di destinazione, indipendentemente dalla rete utilizzata.



Questo è il livello in cui si gestisce per la prima volta (dal basso verso l'alto) una conversazione diretta, cioè senza intermediari, fra una transport entity su un host e la sua peer entity su un altro host.

.



Naturalmente, tali servizi sono realizzati dal livello transport per mezzo dei servizi ad esso offerti dal livello network.

Così come ci sono due tipi di servizi di livello network, ce ne sono due anche a livello transport:

• servizi affidabili orientati alla connessione, detti di tipo stream, offerti dal **TCP** (Transmission Control Protocol);

• servizi senza connessione detti di tipo datagram offerti dall' **UDP** (User Datagram Protocol).

Indirizzi a livello di Trasporto per il TCP/IP

Quando si vuole trasferire una o più **TPDU** (Transport Protocol Data Unit) da una sorgente ad una destinazione di livello 4, occorre specificare mittente e destinatario di livello 4. Il protocollo di livello 4 deve quindi decidere come deve essere fatto l'indirizzo di livello transport, detto ***TSAP address*** (***Transport Service Access Point address***).

Poichè l'indirizzo di livello 4 serve a trasferire informazioni tra applicazioni che lavorano su hosts diversi, deve poter individuare un host e una entità contenuta nell'host. Per questo motivo i protocolli di livello 4 tipicamente definiscono come indirizzo di livello 4 una coppia formata da un indirizzo di livello network che identifica l'host, e da un'altra informazione che identifica un punto di accesso in quell'host **(NSAP address, informazione supplementare)**.

Ad esempio, in TCP/IP un TSAP address (ossia un indirizzo TCP o UDP) ha la forma:

(IP address : port number)

Port number è un intero a 16 bit, che identifica un servizio o un punto di accesso e/o smistamento di livello 4.

Ad es. la coppia (137.204.72.49 : 23) indica la porta 23 dell'host poseidon.csr.unibo.it, cioè l'entry point per il demone telnet, cioè il punto di accesso all'applicazione che permette ad un utente di collegarsi mediante telnet all'host poseidon.

• Anche se l'indirizzo di livello trasporto è formato da questa coppia, **il TCP/IP**, per ridurre l'overhead causato dagli header dei vari livelli, **nella trasmissione effettua una violazione della stratificazione tra i livelli 4 e 3** (Trasporto e Network). Infatti, come vedremo TCP e UDP contengono nei loro header solo i numeri delle porte e riutilizzano gli indirizzi IP di mittente e destinaz. contenuti nel pacchetto IP.

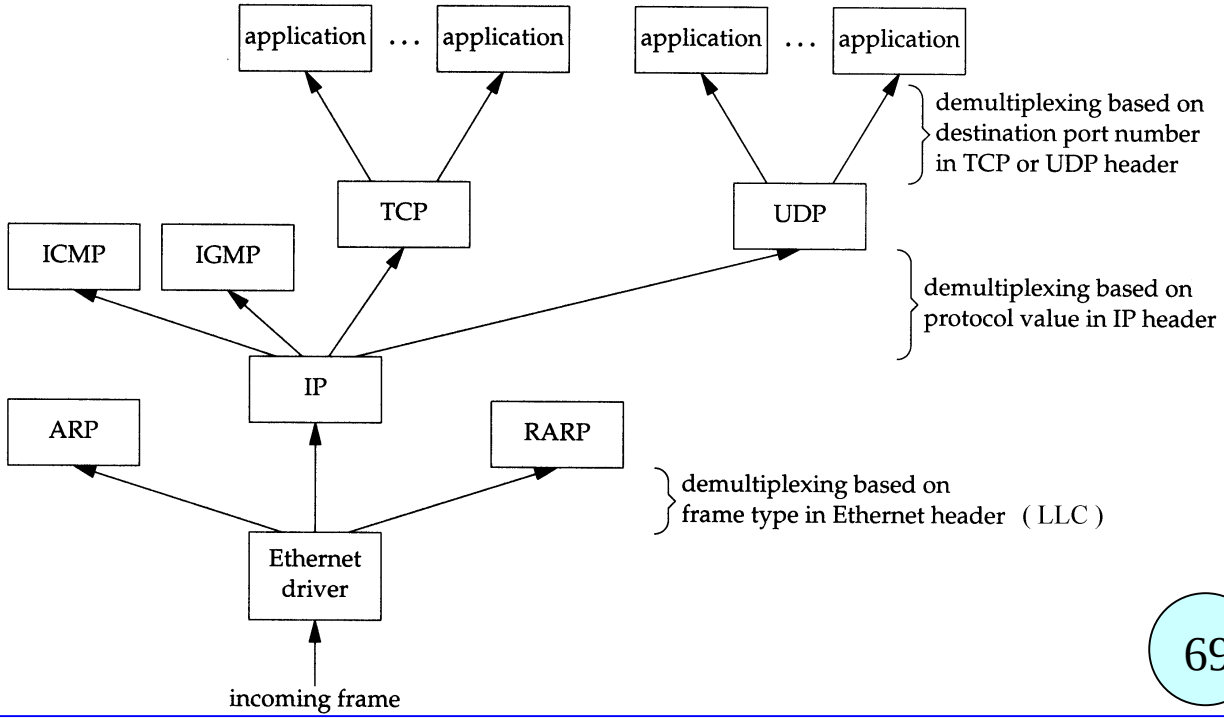
Multiplexing a livello Trasporto

• Gli **identificatori di porta** (port number) permettono di effettuare la **demultiplazione** dei pacchetti di livello 4, ovvero di discriminare l'applicazione destinazione dei pacchetti in funzione del port number contenuto nell'header del pacchetto di livello transport, sia esso di tipo TCP che UDP.

• E' ovvio che mittente e destinatario devono essere d'accordo sul valore della porta del destinatario per poter effettuare la trasmissione.

• Il mittente scrive questo numero di porta come indirizzo del destinatario, ed il destinatario si deve mettere in attesa dei pacchetti che giungono all'host del destinatario e che posseggono come identificatore proprio quel port number.

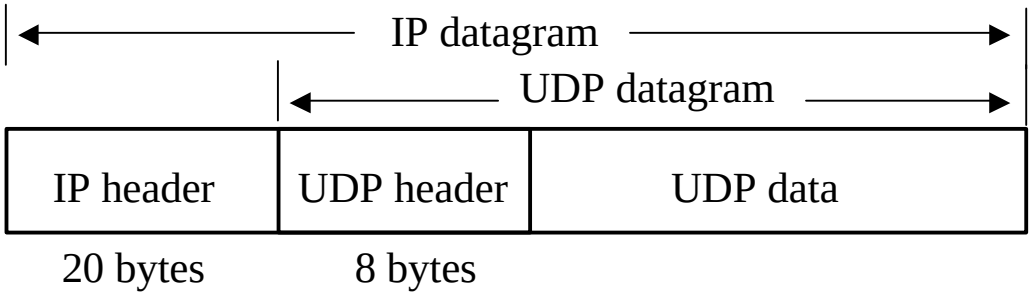
• Alcune primitive fornite dall'interfaccia socket permettono di specificare il numero di porta di cui interessa ricevere i pacchetti (stream=flussi di dati nel caso TCP). E' quindi il sistema operativo che si fa carico di effettuare le operazioni di demultiplexing dei pacchetti ricevuti dal livello network.



Il protocollo di Trasporto UDP (User Datagram Protocol)

• Il livello transport fornisce un protocollo per il trasporto di blocchi di dati non connesso e non affidabile, detto **UDP** (**User Datagram Protocol**), che utilizza l'IP per trasportare messaggi, che è molto simile all'IP in termini di risultato del trasporto, ed offre in più rispetto all'IP la capacità di distinguere tra più destinazioni all'interno di uno stesso host, mediante il meccanismo delle porte.

•Ogni Datagram UDP viene incapsulato in un datagram IP, quindi la dimensione del datagram UDP non può superare la dimensione massima della parte dati del datagram IP. Il datagram IP può essere frammentato se la MTU è piccola.



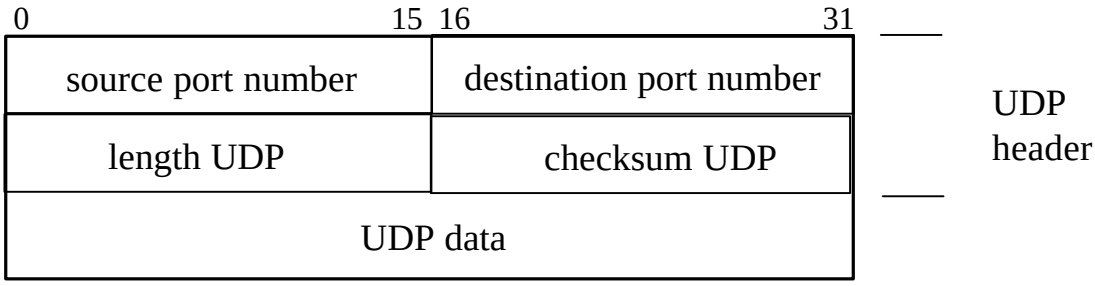
• L'UDP non usa dei riscontri per verificare se un messaggio è arrivato a destinazione, non ordina i messaggi arrivati, e non fornisce nessun tipo di controllo sulla velocità di trasmissione dei dati. Quindi i datagram UDP possono essere persi, duplicati o arrivare fuori ordine.

• Utilizzano UDP alcuni protocolli standard, a cui sono riservati dei numeri di porte predefiniti, in modo da poter essere rintracciati nello stesso punto (punto d'accesso) su tutti gli hosts. Ricordiamo tra gli altri: **nameserver** (server di nomi di dominio, porta 53), **bootps** (server del protocollo di bootstrap, 67), **tftp** (Trivial File Transfer, 69), **ntp** (Network Time protocol, 123).

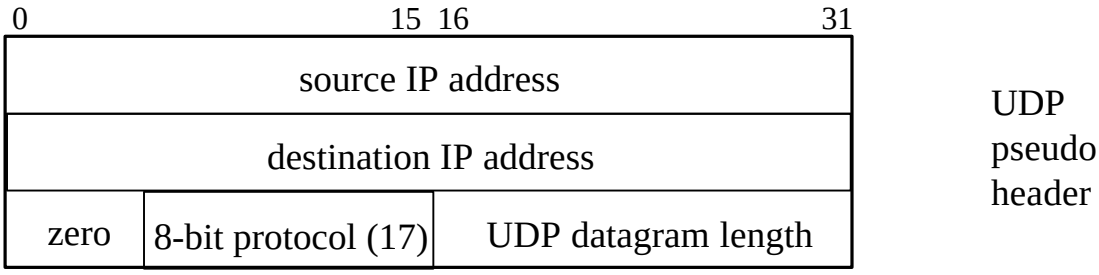
Formato del Datagram UDP

Il pacchetto UDP è costituito da un header e da una parte dati.

L'header UDP è composto da 4 campi:
• i primi due sono i numeri di porta del mittente e del destinatario del datagram, ciascuno di 16 bit.
• il terzo è la lunghezza dei dati del datagram UDP, in byte.
• l'ultimo è un checksum per il controllo d'errore, che però è opzionale. Un valore zero in questo campo indica che la chesksum non è stata calcolata



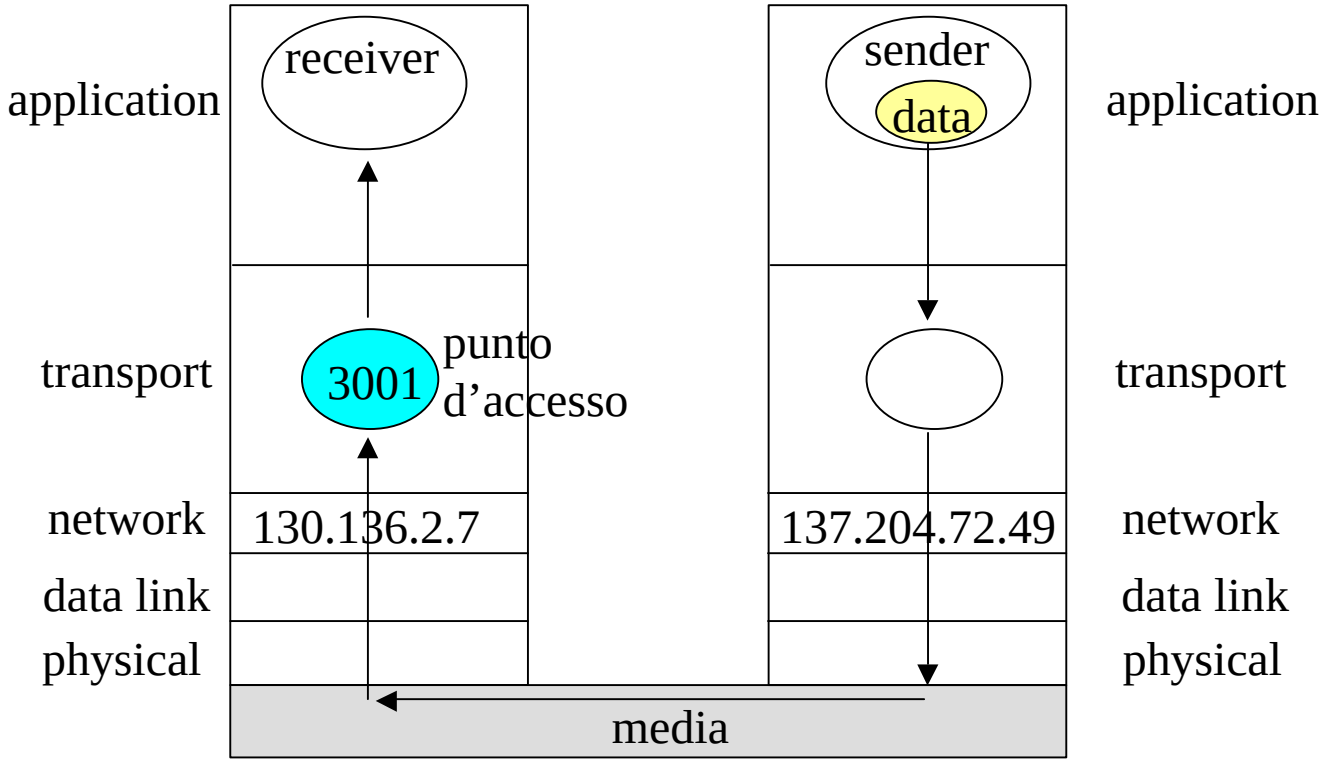
Si noti che non ci sono gli indirizzi IP di mittente e destinatario.
A differenza dell'header IP, il checksum contenuto nell'header UDP considera anche la parte dati UDP.
Inoltre il calcolo della checksum viene effettuato ponendo in testa al datagram UDP una pseudointestazione (che non viene trasmessa), con gli indirizzi IP di provenienza e destinazione ricavata dall'header IP in cui l'UDP viene trasportato, pseudointestazione fatta in questo modo:



Il motivo di questo modo di computare la checksum è verificare a livello UDP, che il datagram UDP abbia raggiunto la corretta destinazione IP, che non compare nell'header UDP.

esempio di trasmissione di datagram UDP

Senza per ora entrare nei dettagli riguardanti i socket, vediamo un semplice esempio di programma che sfrutta i socket per trasmettere un datagram UDP contenente una stringa di testo “pippo” da un host **sender 137.204.72.49** ad un host **receiver 130.136.2.7** .
Il punto di accesso stabilito dal programmatore è nel receiver, nella porta UDP caratterizzata dal numero 3001.
Il receiver si mette in attesa sulla porta **3001** fino a che il sender invia un datagram all’host receiver su quella porta, e stampa il contenuto del datagram ricevuto.
Quindi sender e receiver devono essersi messi d’accordo sulla porta da usare, ed il sender deve conoscere l’indirizzo IP del receiver.



Il codice completo (con la gestione degli errori) dei due programmi che realizzano l’esempio qui mostrato sarà disponibile allo indirizzo www.cs.unibo.it/~ghini/didattica/sistemi1/UDP1win/UDP1win.html

esempio: receiver di datagram UDP

```
#include <sys/types.h>  <errno.h> <winsock.h>
#define SIZEBUF 10000
void main(void)    {
struct sockaddr_in Local, From;   short int local_port_number=3001;
char string_remote_ip_address[100];  short int remote_port_number;
int socketfd, OptVal, msglen, Fromlen;  char msg[SIZEBUF];
WORD wVersionRequested;    WSADATA wsaData; /* per winsock */
wVersionRequested = MAKEWORD( 2, 2 );        /* per winsock */
ris = WSAStartup( wVersionRequested, &wsaData ); /* per winsock */
/* prende un socket per datagram UDP */
socketfd = socket (AF_INET, SOCK_DGRAM, 0);
/* impedisce l'errore di tipo EADDRINUSE nella bind() */
OptVal = 1;
setsockopt (socketfd, SOL_SOCKET, SO_REUSEADDR,
                (char *)&OptVal, sizeof(OptVal) );
/* assegna l'indirizzo IP locale e una porta UDP locale al socket */
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr    =  htonl(INADDR_ANY);
Local.sin_port              =  htons(local_port_number);
bind ( socketfd, (struct sockaddr*) &Local, sizeof(Local));
/* wait for datagram */
Fromlen=sizeof(struct sockaddr);
msglen = recvfrom ( socketfd, msg, (int)SIZEBUF, 0,
                              (struct sockaddr*)&From, &Fromlen);
sprintf((char*)string_remote_ip_address,"%s",inet_ntoa(From.sin_addr);
remote_port_number = ntohs(From.sin_port);
printf("ricevuto msg: \"%s\" len %d, from host %s, port %d\n",
      msg, msglen, string_remote_ip_address, remote_port_number);
WSACleanup(); /* per winsock */
}
```

esempio: sender di datagram UDP

```
int main(void)   {
struct sockaddr_in Local, To;   char msg[]="pippo";
char string_remote_ip_address[]="130.136.2.7";
short int remote_port_number = 3001;int socketfd, OptVal, addr_size;
WORD wVersionRequested;    WSADATA wsaData; /* per winsock */
wVersionRequested = MAKEWORD( 2, 2 );        /* per winsock */
ris = WSAStartup( wVersionRequested, &wsaData ); /* per winsock */
/* prende un socket per datagram UDP */
socketfd = socket(AF_INET, SOCK_DGRAM, 0);
/* impedisce l'errore di tipo EADDRINUSE nella bind() */
OptVal = 1;
setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR,
                (char *)&OptVal, sizeof(OptVal));
/* assegna l'indirizzo IP locale e una porta UDP locale al socket */
Local.sin_family         =  AF_INET;
/* il socket verra' legato all'indirizzo IP dell'interfaccia che verrà
   usata per inoltrare il datagram IP, e ad una porta a scelta del s.o.  */
Local.sin_addr.s_addr  =  htonl(INADDR_ANY);
Local.sin_port           =  htons(0); /* il s.o. decide la porta locale */
bind( socketfd, (struct sockaddr*) &Local, sizeof(Local));
/* assegna la destinazione */
To.sin_family            =  AF_INET;
To.sin_addr.s_addr  =  inet_addr(string_remote_ip_address);
To.sin_port              =  htons(remote_port_number);
addr_size = sizeof(struct sockaddr_in);
/* send to the address */
sendto(socketfd, msg, strlen(msg) , 0,
          (struct sockaddr*)&To, addr_size);
WSACleanup(); /* per winsock */
}
```

Il protocollo di Trasporto TCP (Transmission Control Protocol)

Il protocollo TCP è stato progettato per fornire **un flusso di byte** da sorgente a destinazione **full-duplex, affidabile, su una rete non affidabile**.
Dunque, offre un servizio **reliable** e **connection oriented**, e si occupa di:

- **stabilire la connessione** full duplex tra due punti di accesso a livello trasporto;
- **accettare dati dal livello application** eventualmente **bufferizzando** in input;
- a richiesta, **forzare l'invio interrompendo la bufferizzazione**;
- **spezzare o accorpare** i dati in ***segment***, il nome usato per i TPDU (Transport Protocol Data Unit) aventi dimensione massima 64 Kbyte, ma tipicamente di circa 1.500 byte;
- consegnarli al livello network, per effettuare la trasmissione all'interno di singoli datagram IP, **eventualmente ritrasmettendoli**;
- ricevere segmenti dal livello network;
- **rimetterli in ordine**, eliminando buchi e doppioni;
- **consegnare i dati, in ordine, al livello application**.
- **chiudere la connessione**.

Il servizio effettua internamente la gestione di ack, il controllo del flusso e il **controllo della congestione**.
Il servizio del TCP è di tipo **orientato allo stream**, ovvero **trasporta un flusso di byte**, il che significa che se anche la sorgente spedisce (scrive sul device) i dati a blocchi (es 1 KB poi 3 KB poi ancora 2 KB) la connessione non informa la destinazione su come sono state effettuate individualmente le scritture; la destinazione potrebbe ad es. leggere i dati a blocchi di 20 byte per volta.

Indirizzamento nel TCP (1)

• Come l'UDP, il TCP impiega **numeri di porta di protocollo** per identificare la destinazione finale all'interno di un host.

• La situazione è però molto diversa rispetto all'UDP, in cui possiamo immaginare ogni porta come una sorta di coda a cui arrivano dei datagram delimitati, provenienti ciascuno da un mittente eventualmente diverso.

• **Per il TCP si vuole invece che una connessione sia dedicata in esclusiva ad una coppia di applicazioni risiedenti su macchine diverse.** Il TCP impiega la connessione (virtuale), non la porta di protocollo, come sua astrazione fondamentale;

• le connessioni sono identificate da una coppia di punti d'accesso (endpoint) detti socket, uno su ciascuna macchina.

• Ogni socket è caratterizzato da una coppia (**IP address: Port number**) che può essere utilizzata da più processi simultaneamente. Questa è ad esempio la situazione prodotta dal processo demone del telnet (telnetd) che consente agli utenti di una macchina unix A di collegarsi ad A da un'altro host accedendo tutti alla stessa porta TCP numero 23 di quell'host A. Tutte le connessioni condividono quindi la stessa coppia (IP_A, 23).

• Però **ciascuna connessione viene univocamente individuata dalla coppia di socket dei due host A e B implicati nella connessione, ovvero dalla terna:**
(**IP address A: Port number A** , **IP address B: Port number B**)

Più utenti che effettuano tutti il telnet da una stessa macchina B verso una stessa macchina A instaurano connessioni identificate da terne del tipo (IP_A : 23 , IP_B : **tcp_port_B**) con **tcp_port_B** tutti diversi, e quindi con connessioni univocamente determinate.

Well-Know-Port nel TCP

Come per il caso UDP, anche per il TCP esistono dei port number che sono riservati ad uso di protocolli standard, e vengono detti well-know-port. Queste porte sono quelle con valore inferiore a 256.

Port Number	Service
20	Ftp (control)
21	Ftp (data)
23	Telnet
25	Smtp
80	Http

I segmenti TCP

• L'unità di trasferimento del TCP è detta **segmento**, e viene usato per stabilire connessioni, per trasferire dati, per inviare riscontri (una sorta di ricevuta di ritorno), per dimensionare le finestre scorrevoli e per chiudere le connessioni.

• TCP usa un meccanismo di sliding window (finestre scorrevoli) di tipo go-back-n con timeout. Se questo scade, il segmento si ritrasmette. Si noti che le dimensioni della finestra scorrevole e i valori degli ack sono espressi in numero di byte, non in numero di segmenti.

• ogni byte del flusso TCP è numerato con un numero d'ordine a 32 bit, usato sia per il controllo di flusso che per la gestione degli ack;

• ogni segmento TCP non può superare i 65.535 byte, e **viene incluso in un singolo datagram IP**;

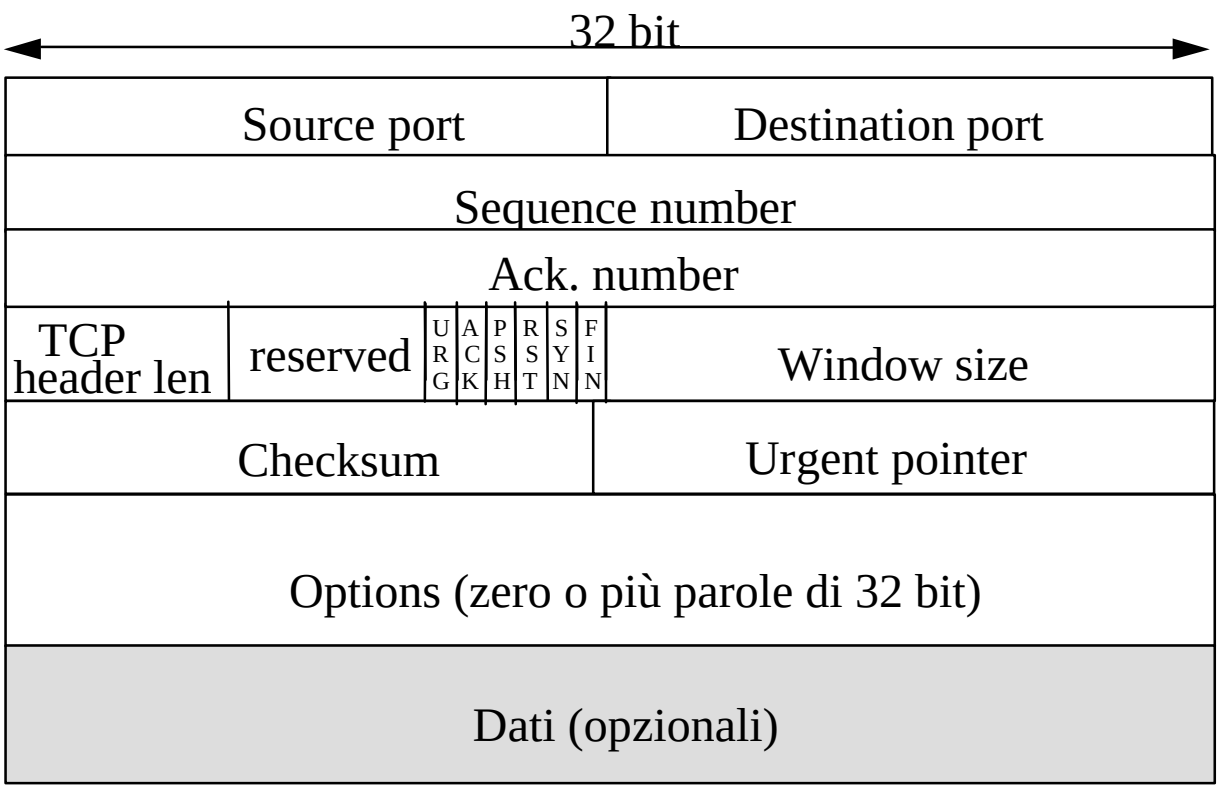
• un segmento TCP è formato da:

uno **header**, a sua volta costituito da:

- una parte fissa di 20 byte;
- una parte opzionale;

i **dati** da trasportare;

Formato del segmento TCP



Source port, destination port: identificano gli end point (locali ai due host) della connessione. Essi, assieme ai corrispondenti numeri IP, identificano la connessione a cui appartiene il segmento;

Sequence number: la posizione del primo byte contenuto nel campo dati **all'interno dello stream di byte che il trasmettitore del segmento invia** (si possono inviare quindi al max 4 miliardi di byte circa in uno stesso stream).

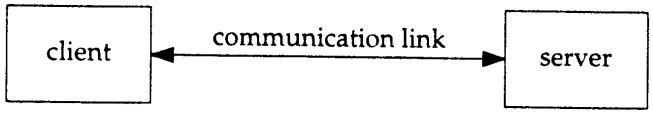
Ack. number: la posizione del prossimo byte aspettato **all'interno del segmento inviato dal ricevitore del presente segmento**.

TCP header length: lunghezza del segmento misurata in parole di 32 bit (necessario perché il campo options ha dimensione variabile).

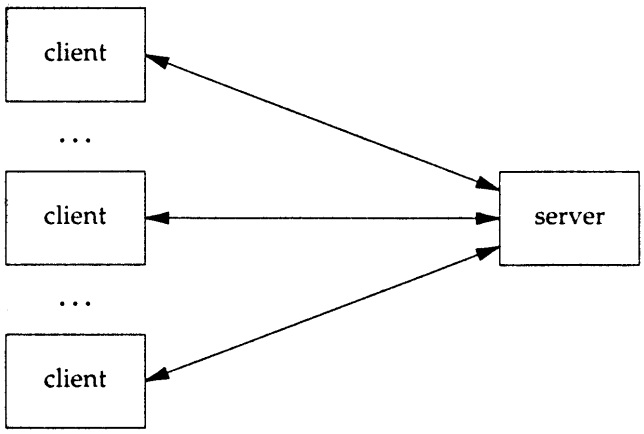
Socket per TCP: Fondamenti

Molte applicazioni di rete sono formate da due programmi distinti (che lavorano su due diversi host) uno detto server ed uno detto client.

Il server si mette in attesa di una richiesta da servire, il client effettua tale richiesta.



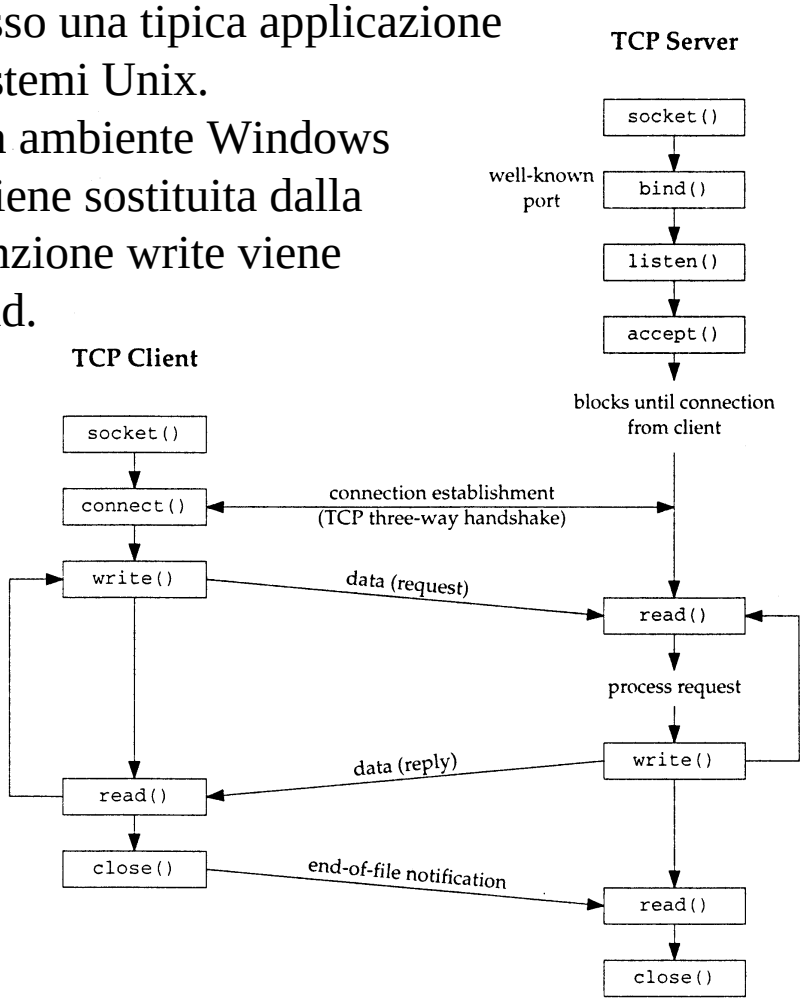
Tipicamente il client comunica con un solo server, mentre un server usualmente comunica con più client contemporaneamente (su connessioni diverse nel caso tcp).



Inoltre spesso client e server sono processi utente, mentre i protocolli della suite TCP/IP fanno solitamente parte del sistema operativo.

Nella figura in basso una tipica applicazione client-server su sistemi Unix.

Per applicazioni in ambiente Windows la funzione read viene sostituita dalla recv, mentre la funzione write viene sostituita dalla send.



Socket Address Structures (1)

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

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

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

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

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

Socket Address Structures (2)

La generica struttura dell'indirizzo è dunque cosi definita:

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

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

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

con

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

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

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

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

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

Funzioni di Ordinamento dei Byte

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

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

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

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

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

Funzioni di Manipolazione dei Byte

Vediamo solo le funzioni portabili ovunque perche sono ANSI C.

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

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

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

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

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

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

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

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

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

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

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

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

I/O su Socket TCP (1)

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

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

cerca di leggere count byte dal file descriptor fd, scrivendoli nel buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata la lettura e viene restituito il numero di byte letti. Se viene restituito zero significa end-of-file (fine stream). Se viene restituito -1 è accaduto un errore e viene settato la variabile globale errno definita in <errno.h>. Il terzo parametro specifica opzioni addizionali, ma di solito vale 0.

La funzione recv presenta una particolarità. Può accadere che la recv() restituisca meno byte di quanti richiesti, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la recv (richiedendo il numero dei byte mancanti) fino ad ottenerli tutti.

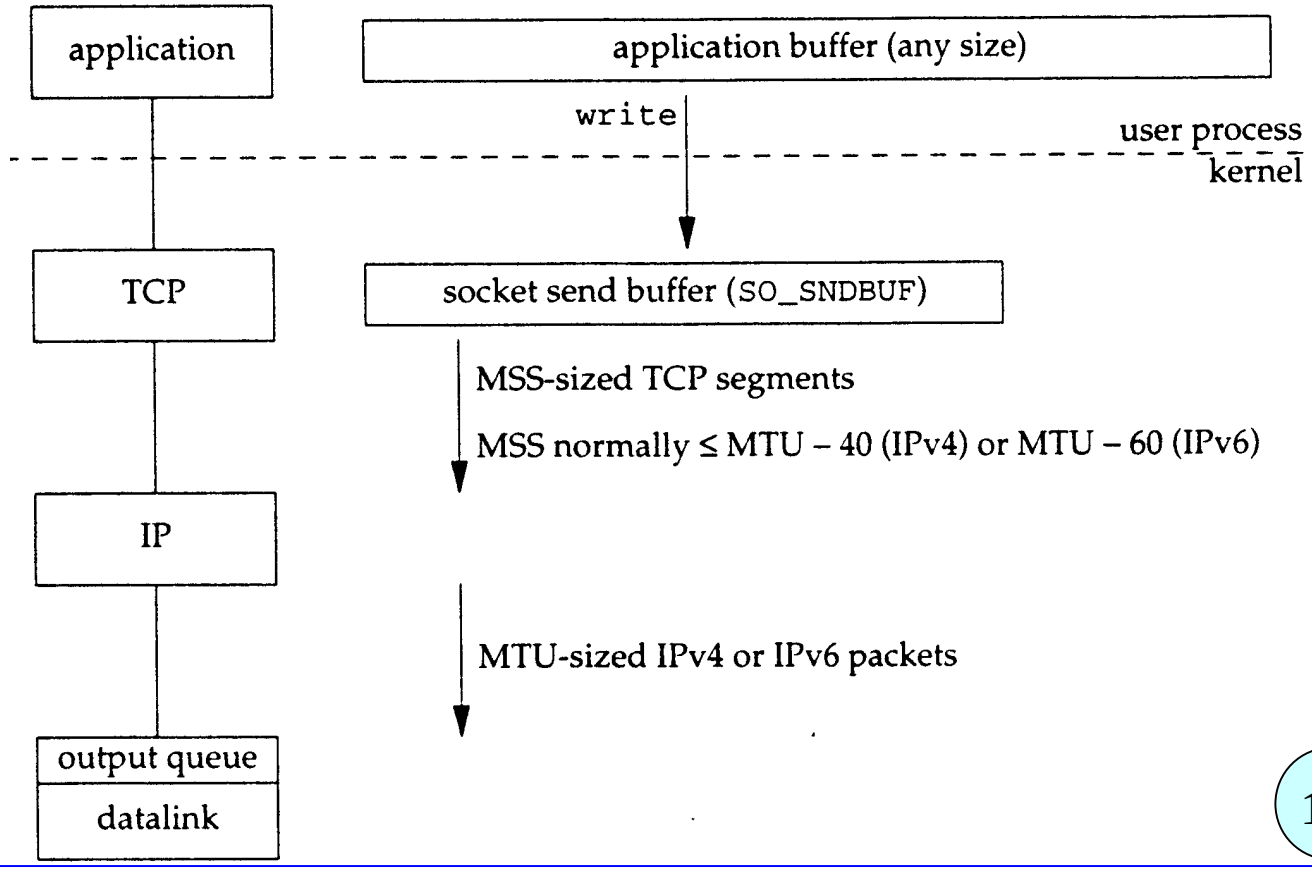
ssize_t **send** (int fd, const void *buf, size_t count , int flags);

cerca di scrivere fino a count byte sul socket descriptor fd, leggendoli dal buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata l'invio e viene restituito il numero di byte scritti nel buffer in uscita. Se viene restituito -1 è accaduto un errore e viene settato errno.

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

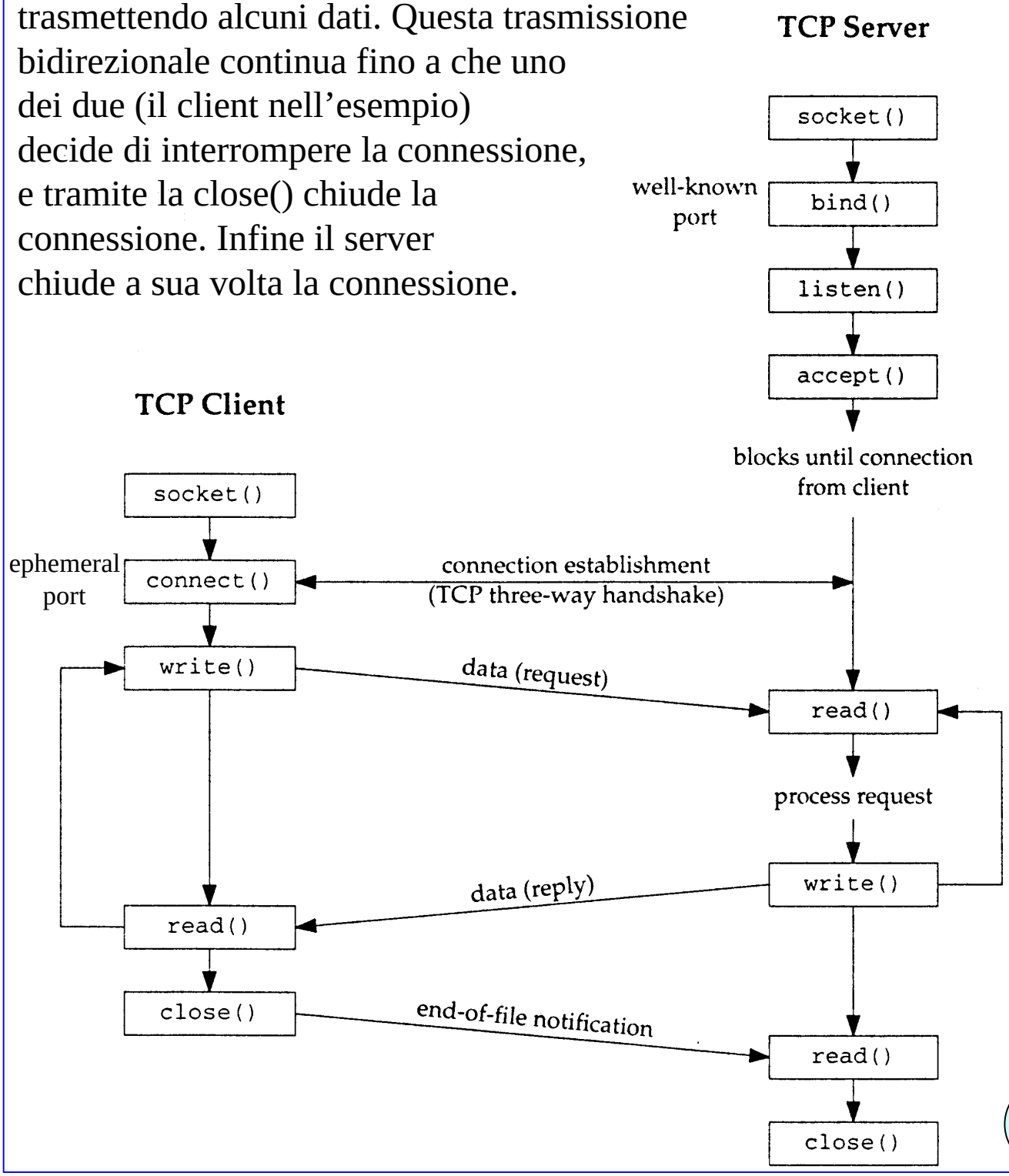
I/O su Socket TCP: TCP Output (2)

Ogni socket TCP possiede un buffer per l'output (send buffer) in cui vengono collocati temporaneamente i dati che dovranno essere trasmessi mediante la connessione instaurata. La dimensione di questo buffer può essere configurata mediante un'opzione SO_SNDBUF.
Quando un'applicazione chiama send() o write() per n bite sul socket TCP, il kernel cerca di copiare n byte dal buffer dell'appl. al buffer del socket. Se il buffer del socket e' più piccolo di n byte, oppure è già parzialmente occupato da dati non ancora trasmessi e non c'è spazio sufficiente, verranno copiati solo nc<n byte, e verrà restituito dalla write o send il numero nc di byte copiati.
Se il socket ha le impostazioni di default, cioè è di tipo bloccante, la fine della routine write o send ci dice che sono stati scritti sul buffer del socket quegli nc byte, e possiamo quindi riutilizzare le prime nc posizioni del buffer dell'applicazione. Ciò non significa affatto che già i dati siano stati trasmessi all'altro end-system.



Interazioni tra Client e Server TCP

Per primo viene fatto partire il server, poi viene fatto partire il client che chiede la connessione al server e la connessione viene instaurata. Le successive trasmissioni di dati avvengono mediante le funzioni read e write (unix) o recv e send (windows) Nell'esempio (ma non è obbligatorio) il client spedisce una richiesta al server, questo risponde trasmettendo alcuni dati. Questa trasmissione bidirezionale continua fino a che uno dei due (il client nell'esempio) decide di interrompere la connessione, e tramite la close() chiude la connessione. Infine il server chiude a sua volta la connessione.



funzione **socket**()

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

#include <sys/socket.h>

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

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

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

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

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

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

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

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

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

funzione **connect**()

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

#include <sys/socket.h>

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

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

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

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

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

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

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

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

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

funzione **bind**() (1)

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

#include <sys/socket.h>

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

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

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

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

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

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

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

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

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

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

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

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

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

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

funzione **bind()** (2)

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

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

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

L'assegnazione viene fatta con le istruzioni:

```
struct sockaddr_in localaddr;
localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
localaddr.sin_port = htons(port_number);
```

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

funzione **listen**()

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

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

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

funzione **accept**()

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

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

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

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

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

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

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

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

funzione **close**()

La funzione close è utilizzata normalmente per chiudere un descrittore di file, è utilizzata per chiudere un socket e terminare una connessione TCP.
int **close** (int socketfd);
restituisce 0 se tutto OK, -1 in caso di errore.
L'argomento **socketfd** è un descrittore di socket.

• Normalmente la chiamata alla close() fa marcare "closed" il socket, e la funzione ritorna il controllo al chiamante. Il socket allora non può più essere usato dal processo, ovvero non può più essere usato come argomento di read e write.
• **Però il TCP continua ad utilizzare il socket trasmettendo i dati** che eventualmente stanno nel suo buffer interno, fino a che non sono stati trasmessi tutti. **In caso di errore (che impedisce questa trasmissione) successivo alla close l'applicazione non se ne accorge e l'altro end system non riceverà alcuni dei dati**.
• Esiste un'opzione però (la **SO_LINGER socket option**) che modifica il comportamento della close, facendo in modo che la **close restituisca il controllo al chiamante solo dopo che tutti i dati nei buffer sono stati correttamente trasmessi** e riscontrati.

• Se un **socket connesso sockfd è condiviso da più processi** (padre e figlio ottenuto da una fork), il **socket mantiene il conto di quanti sono i processi a cui appartiene**. In tal caso la chiamata alla close(sockfd) per prima cosa **decrementa di una unità questo contatore**, e **non innesca la sequenza FIN+ACK+FIN+ACK di terminazione della connessione fino a che tale contatore è maggiore di zero**, perchè **esiste ancora un processo che tiene aperta la connessione**.
• Per innescare veramente la sequenza di terminazione, anche se ci sono ancora processi per quella connessione si usa la funzione **shutdown()**.

funzione **getsockname**()

La funzione getsockname serve a **conoscere l'indirizzo** di protocollo (IP e port number) **dell'host locale** associato ad un certo descrittore di socket connesso.
int **getsockname** (int socketfd, struct sockaddr ***Localaddr**,
socklen_t ***ptr_addrlen**);
restituisce 0 se tutto OK, -1 in caso di errore.

Il primo argomento **socketfd** è un descrittore di socket connesso.
Il secondo argomento **Localaddr** è un puntatore ad una struttura di tipo sockaddr, in cui la funzione metterà l'indirizzo **locale** della connessione.
Il terzo argomento **ptr_addrlen** è un puntatore ad intero in cui la funzione metterà la dimensione della struttura scritta.

Questa funzione viene utilizzata in varie situazioni:

In un client, **dopo una connect se non è stata effettuata la bind**, e quindi non si è specificato nessun indirizzo: in tal caso getsockname permette di conoscere l'indirizzo IP e la porta assegnati dal kernel alla connessione.

In un client dopo una bind in cui come port number è stato specificato il valore 0, con il quale si è informato il kernel di scegliere lui la porta. In tal caso la getsockname restituisce il numero di porta locale assegnato dal kernel.

In un server multihomed, dopo una accept preceduta da una bind in cui come indirizzo IP LOCALE è stata messa la wildcard INADD_ANY, ciòè una volta che si sia stabilita una connessione, la getsockname permette al server di sapere quale indirizzo IP ha la propria interfaccia di rete utilizzata per la connessione.

funzione **getpeername**()

La funzione getpeername serve a **conoscere l'indirizzo** di protocollo (IP e port number) **dell'host remoto** associato ad un certo descrittore di socket connesso.

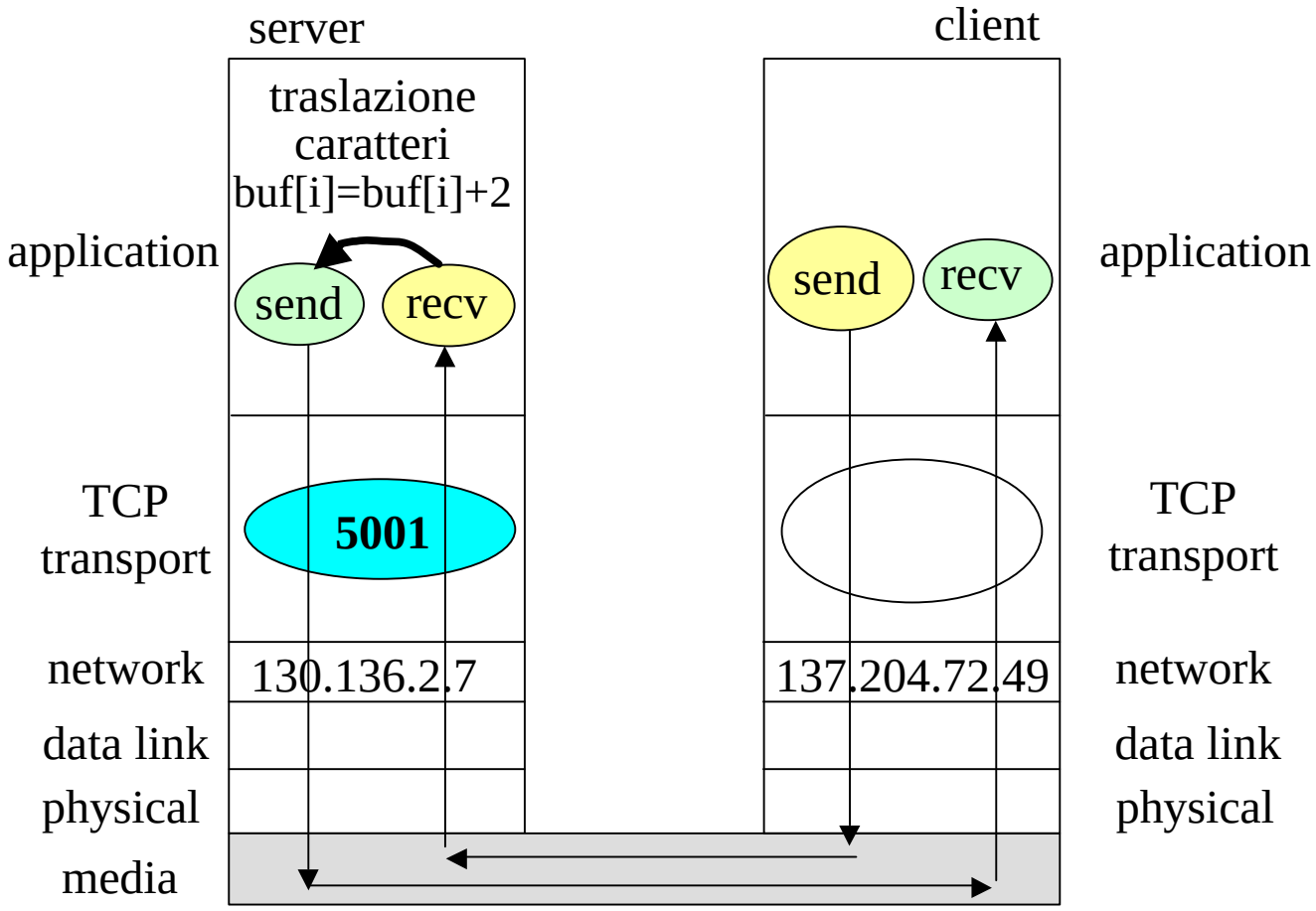
int **getpeername** (int socketfd, struct sockaddr ***Remoteaddr**,
socklen_t ***ptr_addrlen**);

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

Il primo argomento **socketfd** è un descrittore di socket connesso.
Il secondo argomento **Remoteaddr** è un puntatore ad una struttura di tipo sockaddr, in cui la funzione metterà l'indirizzo **remoto** della connessione.
Il terzo argomento **ptr_addrlen** è un puntatore ad intero in cui la funzione metterà la dimensione della struttura scritta.

esempio di trasmissione con TCP

Vediamo un semplice esempio di programma che sfrutta i socket TCP per instaurare una connessione tra un client e un server, trasmettere dal client al server una stringa di caratteri, aspettare che il server modifichi questi caratteri (tranne l'ultimo, lo'\0' che delimita la stringa) shiftandoli di due posizioni (es: 'a' diventa 'c', '2' diventa '4') e li rispedisca indietro così traslati, infine stampare il risultato.
Il server è l'host **130.136.2.7**, mentre il client è l'host **137.204.72.49**.
Il punto di accesso del servizio di traslazione è la porta TCP 5001.



Il codice completo (con la gestione degli errori) dei due programmi che realizzano l'esempio qui mostrato sarà disponibile allo indirizzo www.cs.unibo.it/~ghini/didattica/sistemi1/TCP1win/TCP1win.html

server TCP per l'esempio (1)

```
void main(void)    {
struct sockaddr_in Local, Client; short int local_port_number=5001;
char buf[SIZEBUF];  int sockfd, newsockfd, n, nrecv, nsent, len;
WORD wVersionRequested;    WSADATA wsaData; /* per winsock */

wVersionRequested = MAKEWORD( 2, 2 );
ris = WSAStartup( wVersionRequested, &wsaData ); /* per winsock */

/* prende un socket per stream TCP */
sockfd = socket (AF_INET, SOCK_STREAM, 0);
/* collega il socket ad un indirizzo IP locale e una porta TCP locale */
memset ( &Local, 0, sizeof(Local) );
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr    =  htonl(INADDR_ANY);
Local.sin_port              =  htons(local_port_number);
bind ( sockfd, (struct sockaddr*) &Local, sizeof(Local));
/* accetta max 10 richieste simultanee di inizio conness., da adesso */
listen(sockfd, 10 );
/* accetta la prima conness. creando un nuovo socket per la conness. */
newsockfd = accept(sockfd, (struct sockaddr*) &Cli, &len);
/* riceve la stringa dal client */
nrecv=0;
while( (n=recv(newsocketfd, &(buf[nrecv]), MAXSIZE ,0 )) >=0) {
    nrecv+=n;
    if(buf[nrecv-1]=='\0')
       break; /* fine stringa */
}
/* segue prossima slide */
```

server TCP per l'esempio (2)

```
/* converte i caratteri della stringa */
for( n=0; n<nrecv  -1  ; n++) buf[n] = buf[n]+2;
/* spedisce la stringa traslata al client */
nsent=0;
while (  (nwrite<nread)  &&
            ((n=send(newsocketfd, &(buf[nsent]), nrecv-nsent, 0)) >=0)
        )
   nsent+=n;
/* chiude i socket utilizzati */
close(newsocketfd); close(socketfd);
/* chiude i winsocket */
WSACleanup(); /* per winsock */
}
```

client TCP per l'esempio (1)

```
/* cliTCP.c     eseguito sull'host 137.204.72.49 */
void main(void)    {
struct sockaddr_in Local, Serv; short int remote_port_number=5001;
char msg[]="012345ABCD";  int sockfd, newsockfd, n, nrecv, nsent, len;

WORD wVersionRequested;    WSADATA wsaData; /* per winsock */

wVersionRequested = MAKEWORD( 2, 2 );
ris = WSAStartup( wVersionRequested, &wsaData ); /* per winsock */

/* prende un socket per stream TCP */
sockfd = socket (AF_INET, SOCK_STREAM, 0);

/* collega il socket senza specificare indirizzo IP e porta TCP locali */
memset ( &Local, 0, sizeof(Local) );
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr   =  htonl(INADDR_ANY);
Local.sin_port              =  htons(0);
bind ( sockfd, (struct sockaddr*) &Local, sizeof(Local));

/* specifica l'indirizzo del server, e chiede la connessione */
memset ( &Serv, 0, sizeof(Serv) );
Serv.sin_family             =  AF_INET;
Serv.sin_addr.s_addr     =  inet_addr ( string_remote_ip_address);
Serv.sin_port               =  htons(remote_port_number);
connect ( socketfd, (struct sockaddr*) &Serv, sizeof(Serv));

/* spedisce la stringa al server */
len = strlen(msg)+1;
nsent=0;
while (     (len>nsent)  &&
            ((n=send(socketfd, &(msg[nsent]), len-nsent, 0))>=0)
       )
    nsent+=n;
/* segue prossima slide */
```

client TCP per l'esempio (2)

```
nrecv=0; /* riceve la stringa traslata dal server */
while(  (n=recv (socketfd, &(buf[nrecv]), len-nrecv, 0 )) >=0) {
    nrecv+=n;
    if(buf[nrecv-1]=='\0')
        break; /* fine stringa */
  }

printf("%s\n", msg);  /* stampa la stringa traslata */
/* chiude i socket e termina*/
close(socketfd);

/* chiude i winsocket */
WSACleanup(); /* per winsock */
}
```

Interrogazione al DNS

```
query_to_dns( const char *nomehost) {
register struct hostent *hostptr;
char *ptr;
int ris,len,OptVal;

if( (hostptr = gethostbyname( nome_host ) == NULL)
        return(0); // errore

while((ptr =*(hostptr->h_aliases)) != NULL){
    hostptr->h_aliases++;
}
if(hostptr->h_addrtype==AF_INET) {
        pr_inet(hostptr->h_addr_list,hostptr->h_length );
    while((ptr =(struct in_addr *) *(hostptr->h_addr_list)++)!=NULL)
        printf ("IR address = %s\n", inet_ntoa(*ptr));
}

}


void main(void) {
        query_to_dns( "www.cs.unibo.it");
}
```