

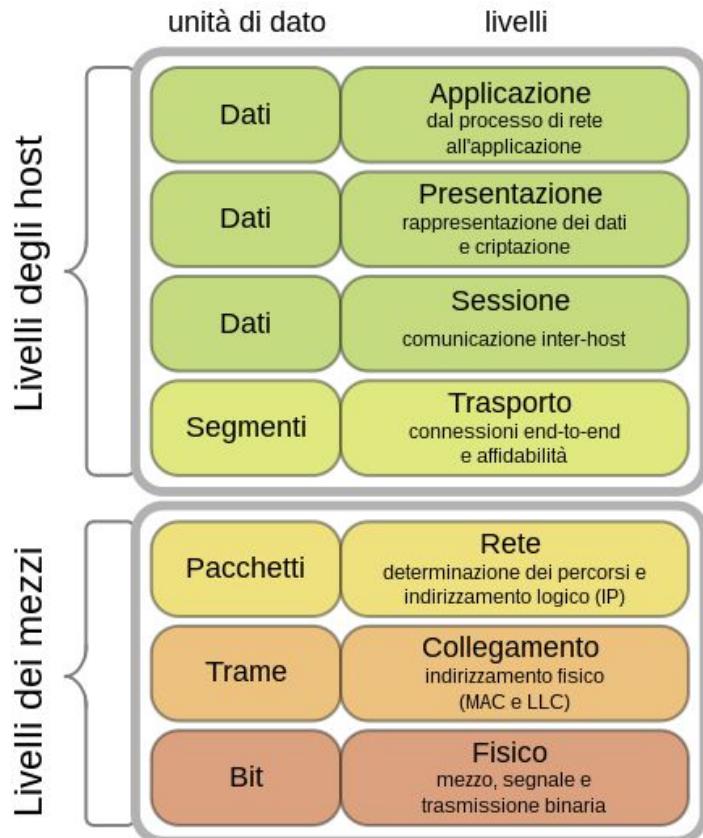
La comunicazione con i socket

5CII 2022/2023

Modello ISO/OSI

Descrizione della funzionalità dei livelli

Descrizione delle principali funzionalità dei livelli

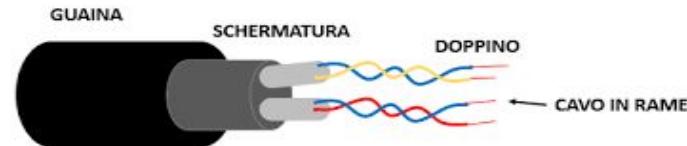


Livello fisico

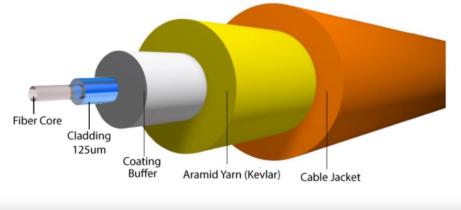
COMPITO: Inviare il segnale fisico al livello fisico del dispositivo di destinazione

CATEGORIE INVIO DATI

cavo elettrico



fibra ottica



- tipi: {
- Tight: luoghi interni
 - Loose: luoghi esterni
 - Slotted core: dorsali e luoghi esterni umidi e bagnati

reti wireless



onde radio

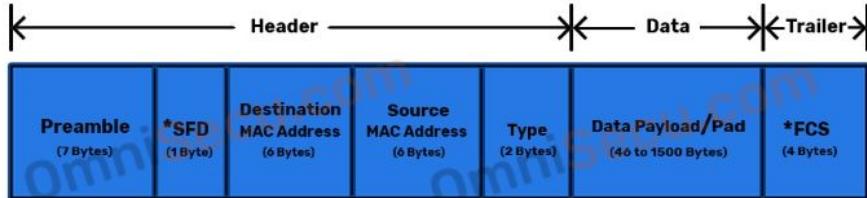


raggi infrarossi

Livello data link

COMPITI

1. riceve i dati dal livello superiore e li suddivide in frame, aggiungendone poi le informazioni di controllo all'inizio e alla fine del frame.



2. controllare gli errori nella trasmissione

MAC: sottolivello che prima che venga inviato il segnale

- controlla che sia pulito tramite il CRC
- lo corregge tramite il codice di Hamming

3. Sincronizzare mittente e destinatario

LLC: sottolivello che ha il compito di gestire la conversazione che può avvenire in diversi modi

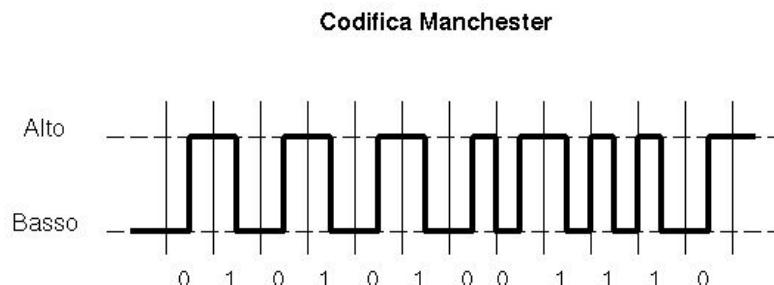
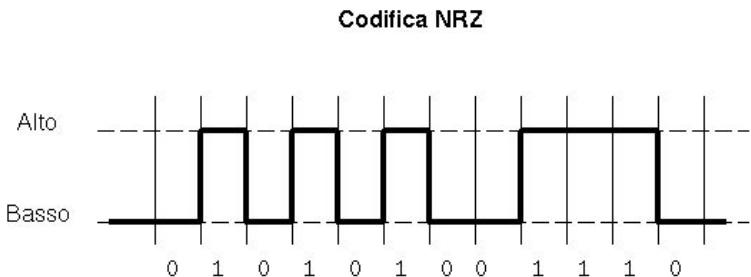
- STOP and WAIT
- PIGGYBACKING
- GO-BACK-N
- SELECTIVE REPEAT

La rete Ethernet

Il principio di funzionamento di una rete Ethernet è:

- ogni host riceve tutti i frame che passano per i cavi
- e solamente uno alla volta può trasmettere le informazioni

tipi di codifica del segnale da trasmettere



Dispositivi di rete

Livello 1

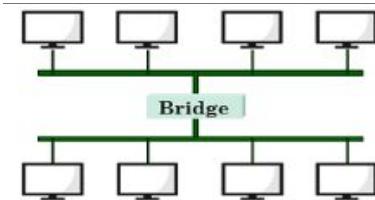
HUB



ripetitore che ritrasmette il segnale ricevuto da un canale ad un altro, anche a lunghe distanze

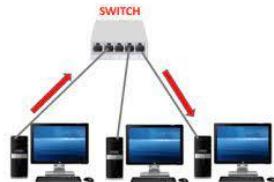
Livello 2

BRIDGE



ha la capacità di collegare più reti e decidere in che canale ripetere il messaggio

SWITCH



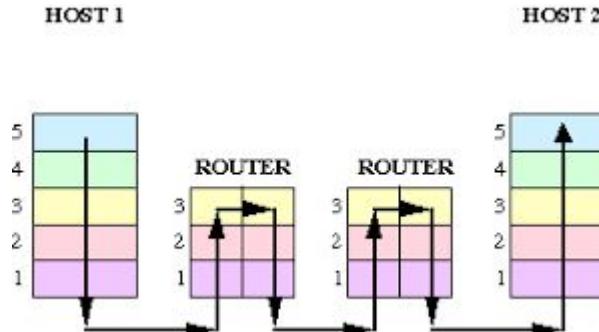
hub ad alte prestazioni

Livello di Rete

il **livello di rete** nonché il 3° livello del modello ISO/OSI riceve segmenti dal soprastante livello di trasporto, e forma pacchetti che vengono passati al sottostante livello. Il compito principale del livello di rete è quello di creare un collegamento tra 2 o più host, si occupa quindi dell'indirizzamento e l'instradamento dei pacchetti nella rete, utilizzando protocolli e **algoritmi di routing** per velocizzare l'invio di pacchetti e scegliere le strade migliori. Per identificare gli host nella rete si utilizzano gli indirizzi IP (composti da 32 bit), e permettono di effettuare i collegamenti fra i vari nodi.

Esistono 2 tipi di algoritmi di routing:

- **Statici**: fatti dall'uomo quindi presentano problemi quando si verifica un cambiamento nella rete
- **Dinamici**: viene utilizzato un LSP (link state packet) ogni qualvolta si verifica un cambiamento nella rete, si aggiorna e diffonde le informazioni in rete



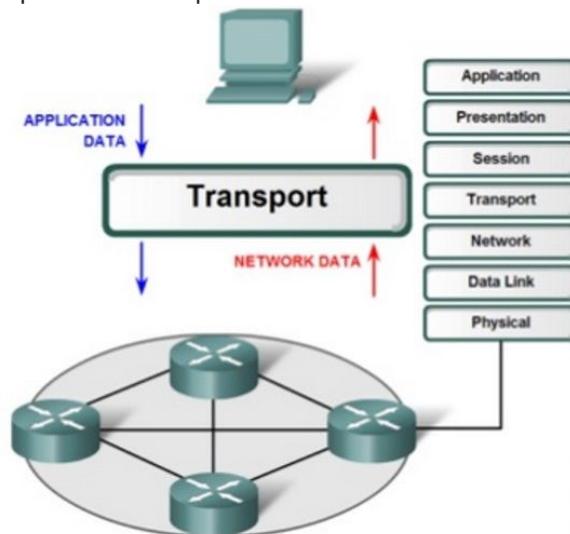
Livello di Trasporto

Il livello di trasporto nonché il 4° livello del modello ISO/OSI.

Il compito del livello di **trasporto** è quello di fornire servizi al soprastante livello, e fornisce un collegamento chiamato **canale logico** (canali indipendenti) di comunicazione end-to-end per i pacchetti in transito, offre un collegamento efficiente tra i 2 host indipendentemente dal mezzo fisico utilizzato, due esempi di protocolli di trasporto sono il protocollo TCP e UDP.

La differenza principale tra **TCP e UDP** è che l'UDP è più veloce rispetto al TCP perchè se il pacchetto non arriva a destinazione infatti l'UDP non si preoccupa di rimandare il pacchetto e continua con la trasmissione.

Dentro al computer per riconoscere il processo dove far arrivare il pacchetto, il transport layer si serve delle porte di comunicazione. Sono univoche per ogni processo e alcune sono conosciute per utilizzi specifici. La coppia indirizzo IP+porta si chiama socket.



Livello Sessione

Cos'è una sessione?

La sessione è una conversazione fra due host, dove entrambi conoscono l'identità l'uno dell'altro. Il suo ID le permette di essere univoca per ogni utente e di essere riaperta nel tempo.

La sessione è composta da tre fasi:

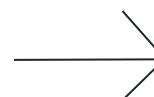
1. apertura, dove client e server si scambiano informazioni per iniziare la connessione;
2. lavoro in sessione;
3. chiusura, quando il server cancella le informazioni di sessione.



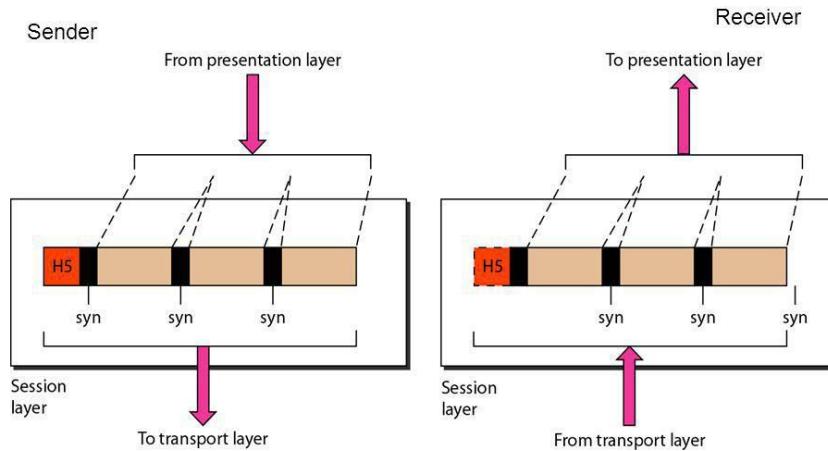
Livello Sessione

Si occupa di:

- definire la sessione;
- controllare gli errori nella connessione
- gestire l'apertura e chiusura.



Inserisce dei sync point nella conversazione, dividendola logicamente, così in caso di errore ripristina la sessione da quel punto (roll-back).

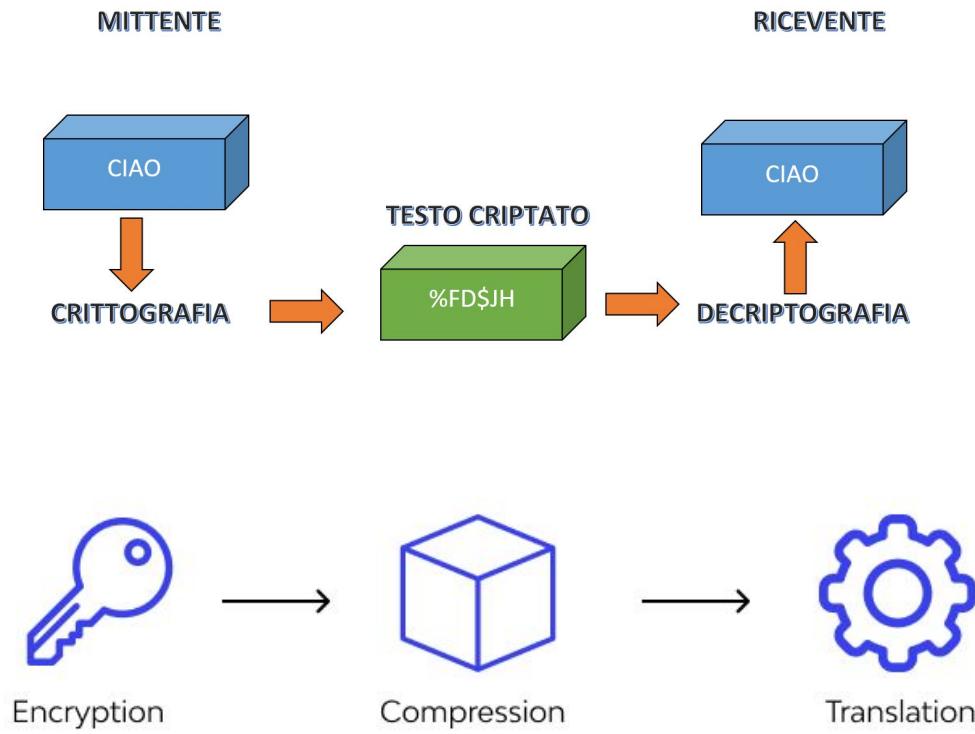


Livello presentazione

Il suo compito è quello di presentare i dati in un formato comprensibile all'application layer, o di tradurli in linguaggio di rete per il session layer.

Si occupa di:

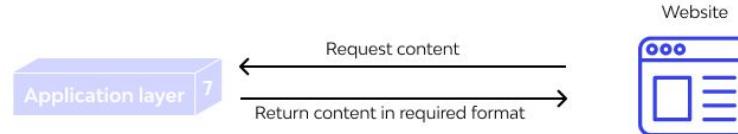
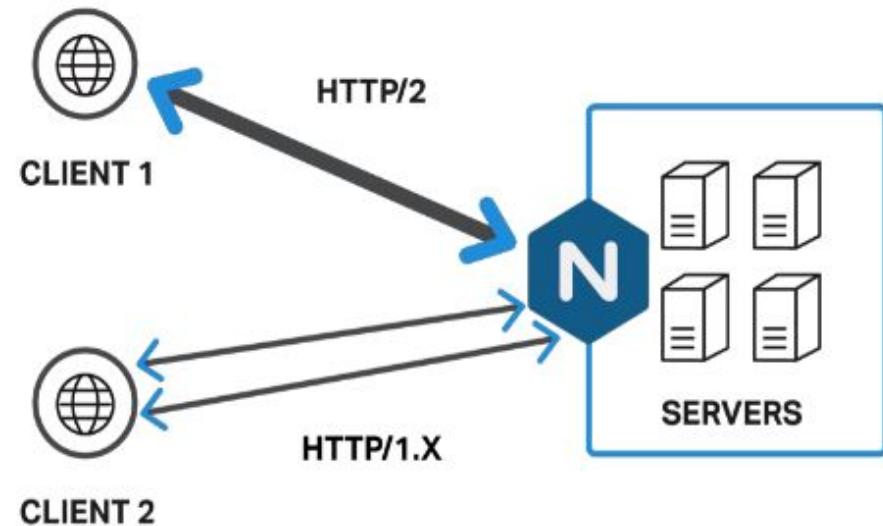
- crittografia;
- traslazione, traduce i dati ricevuti nel linguaggio locale;
- compressione, lossy, dove c'è perdita di dati come in formato MP3 e lossless, dove non si perdono informazioni come nel formato zip.



Livello applicazione

È l'ultimo livello della pila ISO/OSI, non fornisce servizi agli altri strati della pila ma fornisce un'interfaccia ai processi delle applicazioni a livello client per accedere alle reti.

Alcuni protocolli usati sono l'HTTP, che serve per trasmettere informazioni sul web, e il DNS, che serve a convertire gli indirizzi web a parole in indirizzi IP.



I Socket

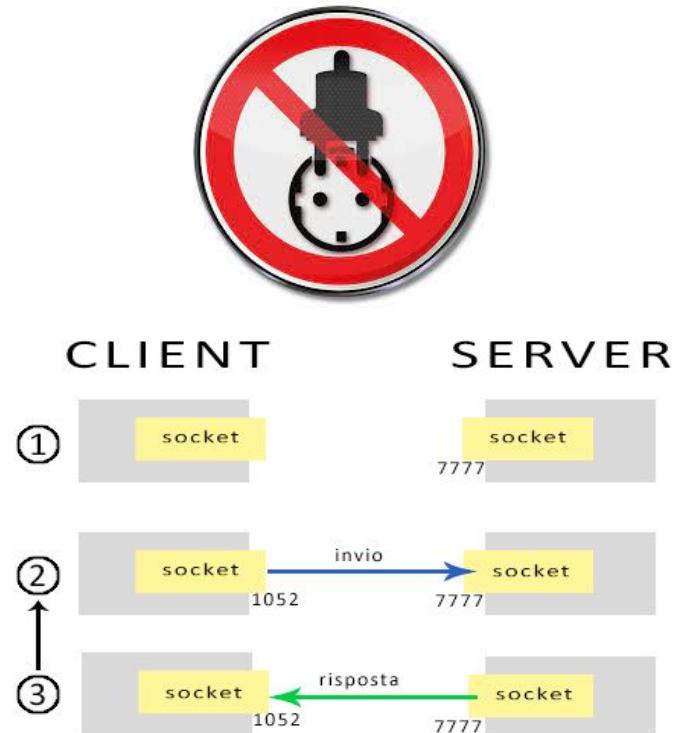
I Socket , parola che in inglese significa “presa di corrente”, sono uno strumento utilizzato per la comunicazione tra due computer in rete.

Essi sono identificati da un indirizzo IP e da una porta.

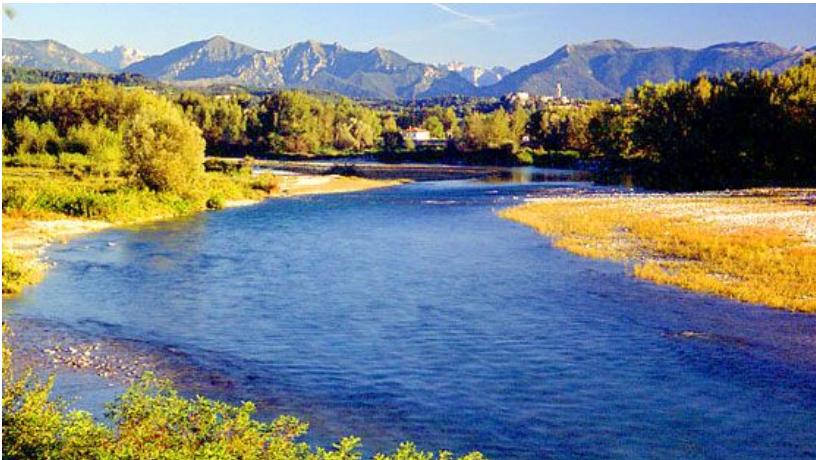
I socket usano principalmente il protocollo TCP e UDP

Esistono 3 tipi di Socket :

- STREAM SOCKET (TCP)
- DATAGRAM SOCKET (UDP)
- RAW SOCKET



Stream Socket



Uno stream socket viene utilizzato quando mi serve una connessione con queste caratteristiche:

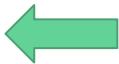
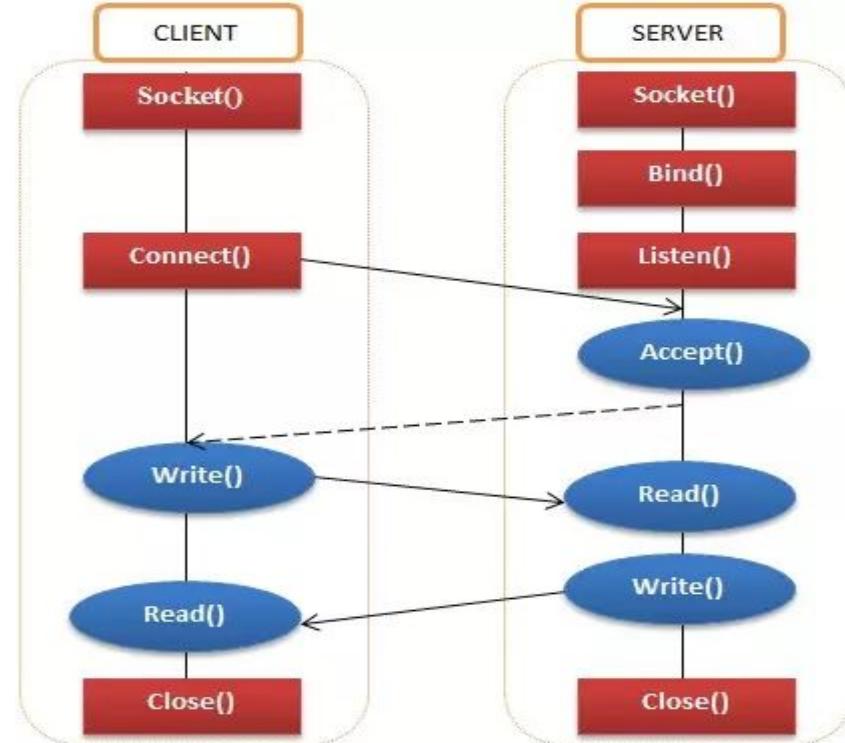
- una connessione sequenziale
- simmetrica
- affidabile
- full-duplex basata su stream di byte di lunghezza variabile
- usa principalmente il protocollo TCP

immagine fiume della patria “Piave”



Fasi Stream Socket

1. Un server crea il suo endpoint creando l'oggetto socket (**socket()**)
2. Con la funzione **bind()** collega il socket ad una porta e si mette in ascolto con **listen()**
3. Il client crea un socket locale (**socket()**) per connettersi al server e gli manda una richiesta (**connect()**)
4. Quando il server la riceve la accetta (**accept()**) e si instaura la connessione
5. Client e server comunicano tra loro tramite un “canale virtuale”
6. I due processi si scambiano dati fino alla effettiva chiusura del canale (**close()**)



I Datagram Socket

I datagram socket vengono utilizzati quando si predilige una comunicazione veloce e l'affidabilità della connessione non è un prerequisito.

Hanno le seguenti caratteristiche:

- Utilizzano una comunicazione connectionless (non richiede una connessione o accordo tra client e server)
- Non garantiscono l'ordine di arrivo dei pacchetti
- Supportano l'invio dei pacchetti verso più destinazioni (multicasting)
- Supportano la ricezione di pacchetti da più sorgenti diverse
- Utilizzano il protocollo UDP

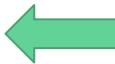
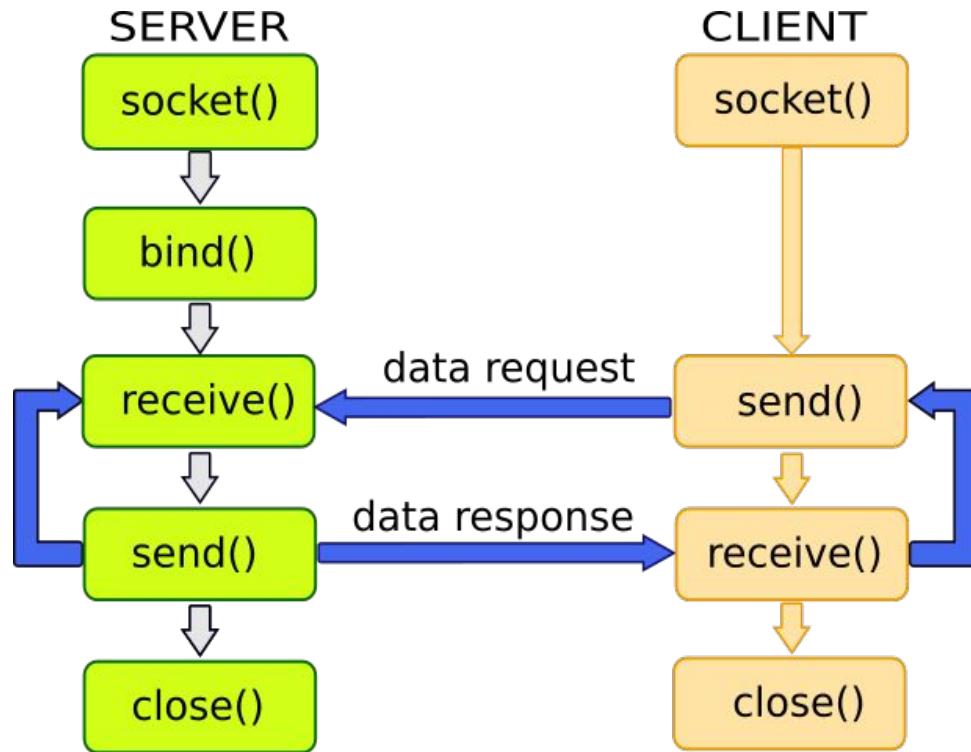
Vantaggio principale

- Trasferiscono velocemente i dati nel modo più rapido possibile



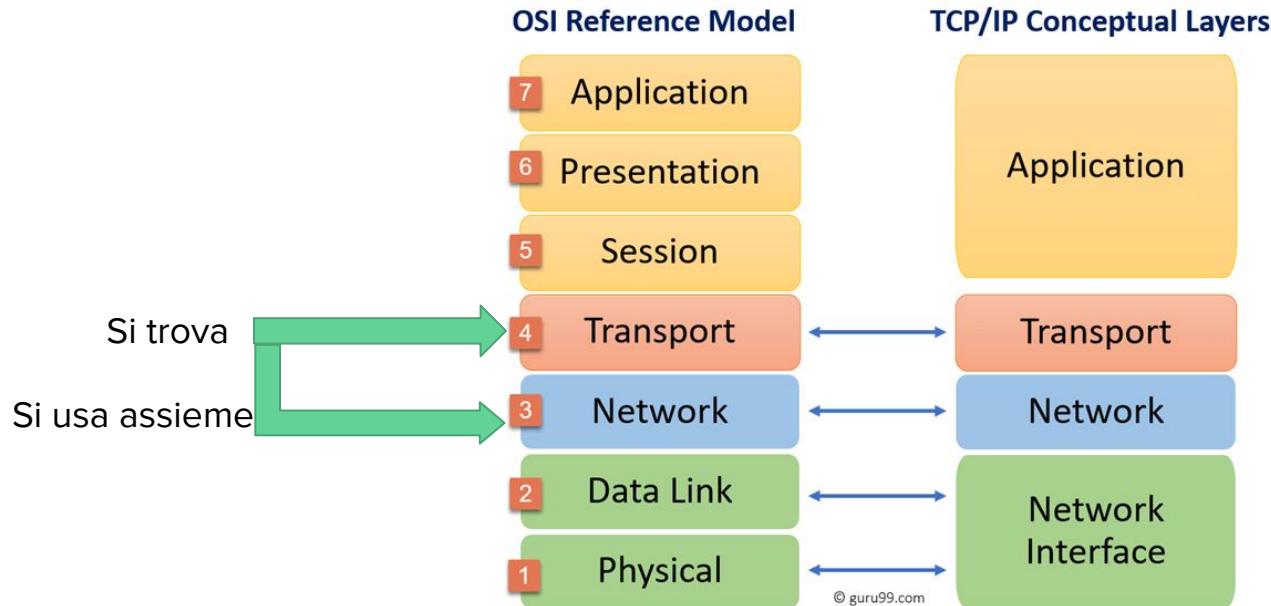
Fasi Datagram Socket

- Ogni processo crea il proprio endpoint richiamando la primitiva che crea il socket (`socket()`)
- Il server richiama la primitiva `bind()` che associa un indirizzo e una porta al socket
- Il server va in attesa tramite la primitiva `receive()` e all'arrivo dei dati risponde tramite la primitiva `send()`
- il client fa la stessa cosa con le due primitive
- Al termine della connessione il socket viene chiuso (`close()`)



Protocollo TCP

Transmission Control Protocol (TCP) è un protocollo di rete nel livello di trasporto (livello 4). Il compito principale è rendere *affidabile* la comunicazione dati in rete tra mittente e destinatario. La si usa assieme al livello 3 (IP).



Caratteristiche Protocollo TCP

- **Orientato alla connessione**
Client e server prima di comunicare si accordano su come inviare i pacchetti in modo affidabile
- **Un protocollo affidabile**
Garantisce che pacchetti arrivino tutti e se ne possa ricostruire l'ordine corretto
- **È "Full-duplex"**
Client e server prima di comunicare si accordano su come inviare i pacchetti in modo affidabile
- **Offre funzionalità di controllo di errore**
Il checksum verifica la correttezza del messaggio
- **Permette il controllo di flusso**
Permette di gestire problematiche relative alla congestione della rete tramite alcuni parametri

Intestazione TCP

Source port			Destination Port	
Sequence number				
Acknowledgment number				
DO	RSV	Flags	Window	
Checksum			Urgent pointer	
Options				

Source port: 16 bit per porta del mittente

Destination port: 16 bit per porta del destinatario

Sequence number (SEQ):

Numero intero di 32 bit che serve a tenere traccia della sequenza dei dati inviati durante la sessione TCP.

Il numero viene configurato al momento della connessione. Può essere un numero casuale o partire da zero.

Acknowledgment number (ACK):

Numero di 32 bit che viene utilizzato dal ricevitore per confermare i dati inviati dal mittente. Per confermare la ricezione dei dati si invia un ACK che corrisponde al SEQ del mittente sommato ai byte ricevuti.

Flags TCP

Flag principali nella comunicazione tra host:

- **SYN (Synchronization)**

Serve ad un host per inviare una richiesta di connessione al server. E' accompagnata a un numero di SEQ con cui inizializzare il contatore dei byte inviati.

- **ACK (Acknowledgment)**

Viene inviato per confermare una richiesta di connessione

- **FIN (Finish)**

Viene utilizzata per richiedere la terminazione di una connessione fra host

Three way handshake

Client(Alice) Server(Bob)

Procedura utilizzata per instaurare in modo affidabile una connessione TCP tra due host.

Consiste nello scambio di 3 messaggi fra host mittente e host ricevente affinché la connessione sia instaurata correttamente.

Vengono configurati i numeri di seq del server e del client

Come funziona il Three way handshake

1) Il Client invia un segmento SYN al Server

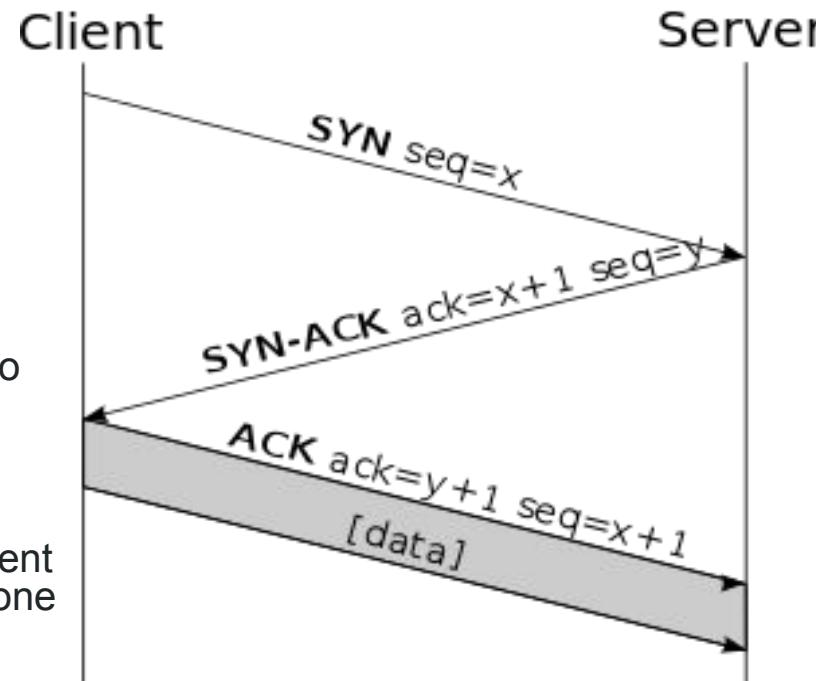
Il flag SYN è impostato a 1 e il campo Sequence number contiene il valore x che specifica l'Initial Sequence Number del Client

2) Il Server invia un segmento SYN/ACK al Client

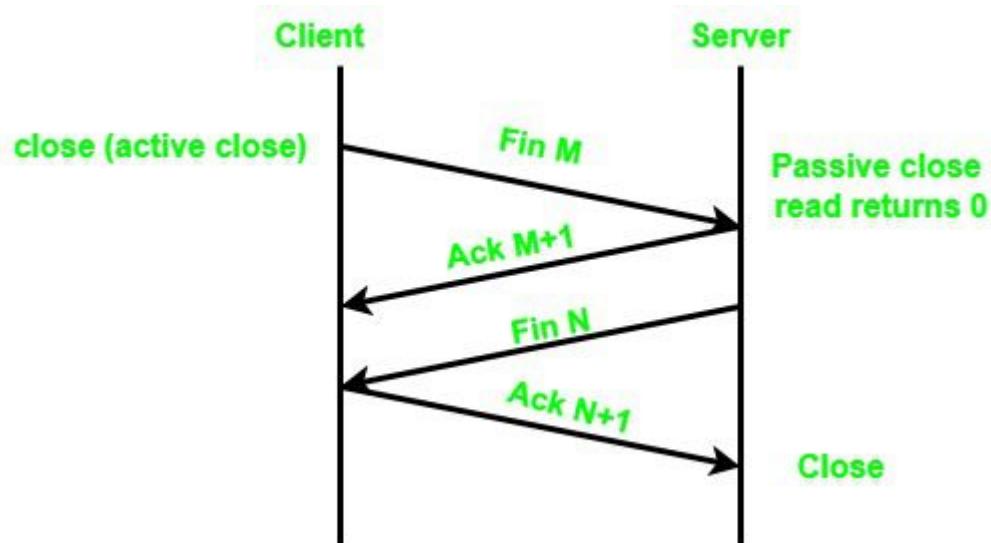
I flag SYN e ACK sono impostati a 1, il campo Sequence number contiene il valore y che specifica l'Initial Sequence Number (ISN) del Server e il campo Acknowledgment number contiene il valore x+1 confermando la ricezione del ISN del Client

3) Il Client invia un segmento ACK al Server

Il flag ACK è impostato a 1 e il campo Acknowledgment number contiene il valore y+1 confermando la ricezione del ISN del Server



4 way handshake (TCP Connection Termination)



1. Host A → Host B: FIN flag
2. Host B → Host A: ACK flag
3. Host B → Host A: FIN flag
4. Host A → Host B: ACK flag

4 way handshake (TCP Connection Termination)

Cos'è?

È il metodo utilizzato per terminare la comunicazione TCP.

Come funziona?

Quando uno tra client e server desidera chiudere la connessione, invia un messaggio con il flag FIN impostato a 1 la cui ricezione viene confermata con un messaggio di ACK.

L'altro host può finire di inviare i dati che sta inviando e anche lui può terminare la connessione invia a sua volta un messaggio di FIN.

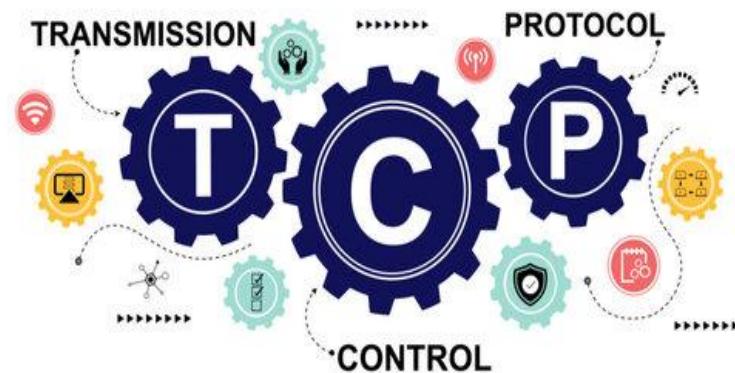
La connessione si chiude alla ricezione dell'ACK di conferma.

TCP

Quando vogliamo collegarci a Internet, con poche semplici mosse realizziamo una connessione tra router e computer o dispositivo mobile o via cavo o mediante accesso wireless.

Il Transmission Control Protocol (TCP) stabilisce le modalità di scambio dei dati tra i dispositivi collegati nella rete, permette quindi a due punti di realizzare una connessione attraverso cui può avvenire una **trasmissione bidirezionale dei dati**.

Nell'ambito di questa connessione, le eventuali perdite di dati vengono riconosciute e corrette automaticamente e per questo il TCP viene denominato anche **protocollo affidabile**.

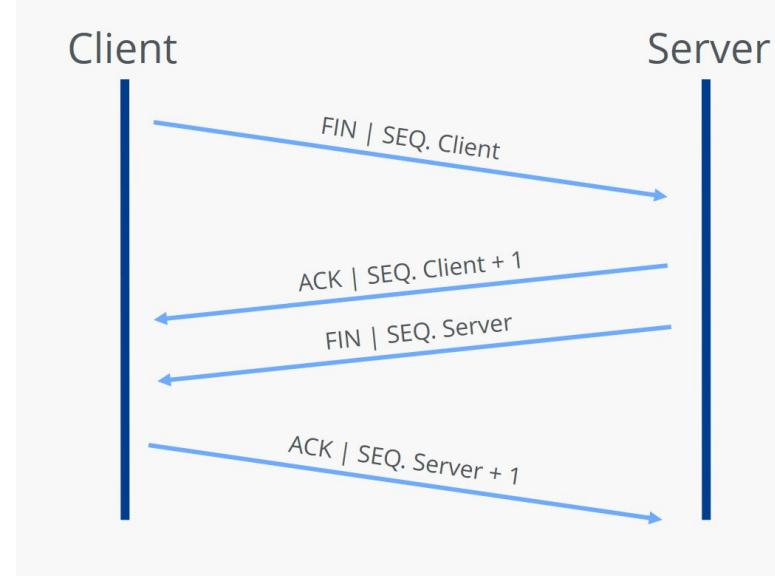


I sistemi informatici che comunicano tramite TCP possono pertanto inviare e ricevere dati contemporaneamente.

I segmenti (pacchetti), oltre al messaggio effettivo, possono contenere anche informazioni di controllo.

Per instaurare una connessione TCP entrambi i punti terminali devono disporre già di un **indirizzo IP univoco**.

1. Nel primo passaggio, il **client** che richiede la connessione invia al server un **pacchetto SYN** o segmento SYN (sincronizzare) con un numero sequenziale individuale e casuale. Questo numero assicura la trasmissione completa nella sequenza corretta (senza doppiioni).
2. Dopo che il **server** ha ricevuto il segmento, acconsente all'instaurazione della connessione restituendo un **pacchetto SYN-ACK** (conferma).
3. Infine, il **client** conferma la ricezione del segmento SYN-ACK inviando un proprio **pacchetto ACK**.



TCP in Java - Struttura iniziale

Il TCP utilizza un protocollo “connection-oriented”, quindi sicuro e, come già visto, richiede tempo per la connessione.

Nella programmazione in Java la connessione TCP si effettua con classi predisposte.

Tra di esse possiamo trovare il “OutputStream”/”InputStream” che vengono utilizzate per scrivere all'interno del canale e possono essere trovate tutte all'interno del pacchetto “java.io”

```
import java.io.InputStreamReader;  
import java.io.OutputStreamWriter;
```

TCP in Java - Server

```
try {

    System.out.println("SERVER ATTIVO");
    server = new ServerSocket(8081);

    System.out.println("SERVER IN ATTESA CHE UN CLIENT SI CONNETTA.....");
    connection = server.accept();

    System.out.println(connection);

    input = new BufferedReader(new InputStreamReader(connection.getInputStream()));
    output = new BufferedWriter(new OutputStreamWriter(connection.getOutputStream()));

    while (true) {
        String letta = input.readLine();

        if (letta.compareTo("EXIT") == 0) {
            output.write("CLOSE");
            output.flush();
            System.out.println("Chiusura del canale");
            break;
        }

        output.write(letta.toUpperCase() + "\n");
        output.flush();
    }

    input.close();
    output.close();

    connection.close();

} catch (IOException e) {
    e.printStackTrace();
}
```

Una delle metodologie più semplici per creare un server TCP in Java è la seguente:

- creiamo una funzione in cui definire le funzionalità del server:
 - creare il server socket e attendere una connessione da parte di un host
- istanziamo i buffer (reader/writer) per leggere nella stream
- eseguiamo un ciclo in cui leggiamo un valore inviato dal socket connesso e lo gestiamo in base a ciò che è richiesto.
- è buon uso chiudere sempre la connessione e i due buffer di lettura/scrittura.

TCP in Java - Client

```
public Client() {  
  
    try {  
        connection = new Socket(InetAddress.getLocalHost(), 8081);  
        System.out.println(connection);  
  
        input = new BufferedReader(new InputStreamReader(connection.getInputStream()));  
        output = new BufferedWriter(new OutputStreamWriter(connection.getOutputStream()));  
        kbReader = new Scanner(System.in);  
  
        while(true){  
  
            System.out.print("Scrivi la tua frase: ");  
            output.write(kbReader.nextLine()+"\n");  
            output.flush();  
  
            String letta = input.readLine();  
  
            if(letta.compareTo("CLOSE")==0){  
                System.out.println("chiusura del canale.");  
                break;  
            }  
  
            System.out.println("Il server mi ha risposto: "+ letta);  
        }  
  
        kbReader.close();  
        input.close();  
        output.close();  
        connection.close();  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Il Client sarà molto simile al Server, in quanto deve seguire linearmente le sue richieste (quando il server riceve un valore e quando invece lo invia).

- Istanziamo il socket e i buffered reader/writer + scanner per input data
- eseguiamo un ciclo tramite cui l'utente interagirà usando la console
- inviamo il valore dell'utente e leggiamo la risposta del server così da verificarne la correttezza
- chiudere eventuali istanze aperte prima della fine del programma.

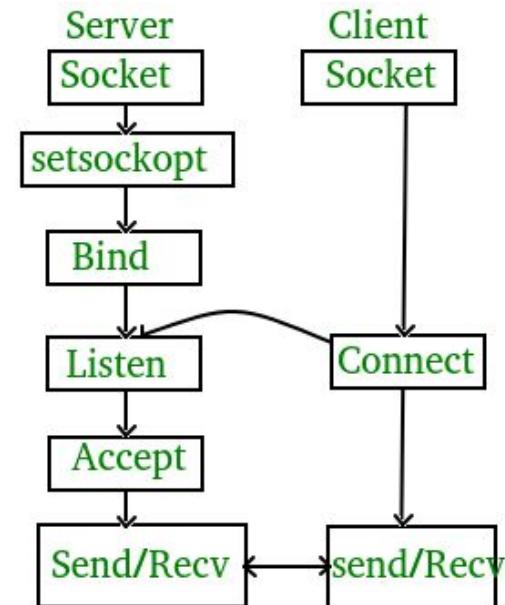
TCP in C

Come si vede nella foto, Client e Server iniziano con la creazione del proprio socket richiamando la funzione **Socket()**.

Il server attraverso la funzione **Bind()** associa al socket un indirizzo rete e si mette in ascolto(**Listen()**), aspettando che il Client si connetta.

Il Client chiamando la funzione **Connect()** si mette in collegamento con il server che a sua volta deve accettare tale collegamento tramite la funzione **Accept()**, che valida la richiesta del Client.

Quando questa pratica di Handshake è terminata e entrambi sono finalmente collegati, inizia il loop di comunicazione con una parte che ascolta e l'altra che riceve e viceversa.



Socket in C (Vlady)

```
int socket(int domain, int type, int protocol)
```

domain - Specifica il dominio di comunicazione (AF_INET per IPv4/ AF_INET6 per IPv6)

type - il tipo di socket che deve essere creato (SOCK_STREAM per TCP / SOCK_DGRAM per UDP)

protocol - il protocollo di comunicazione utilizzato dal socket

```
// creazione del socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    printf("socket creation failed...\n");
    exit(0);
} else
    printf("Socket successfully created..\n");
```

```
int bind(int sockfd, const struct sockaddr *addr,
socklen_t addrlen)
```

sockfd - descrittore del socket

addr - struttura dell'indirizzo a cui bisogna legare il socket

addrlen - dimensione della struttura dell'indirizzo

```
// associazione al socket appena creato un IP
if((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
    printf("socket bind failed...\n");
    exit(0);
} else
    printf("Socket successfully binded..\n");
```

```
int listen(int sockfd, int n)
```

sockfd - descrittore del socket

n - numero dei tentativi per la connessione

```
// controllo se il server e' pronto per l'ascolto
if ((listen(sockfd, 5)) != 0) {
    printf("Listen failed...\n");
    exit(0);
}
```

```
int accept(int sockfd, struct sockaddr *addr, socklen_t addrlen)
```

sockfd - descrittore del socket

addr - struttura dell'indirizzo a cui bisogna legare il socket

addrlen - dimensione della struttura dell'indirizzo

```
// controllo del arrivo dei dati dal client  
connfd = accept(sockfd, (SA*)&cli, &len);  
if (connfd < 0) {  
    printf("server accept failed...\n");  
    exit(0);  
}  
else  
    printf("server accept the client...\n");
```

```
int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen)
```

sockfd - descrittore del socket

addr - struttura dell'indirizzo a cui bisogna legare il socket

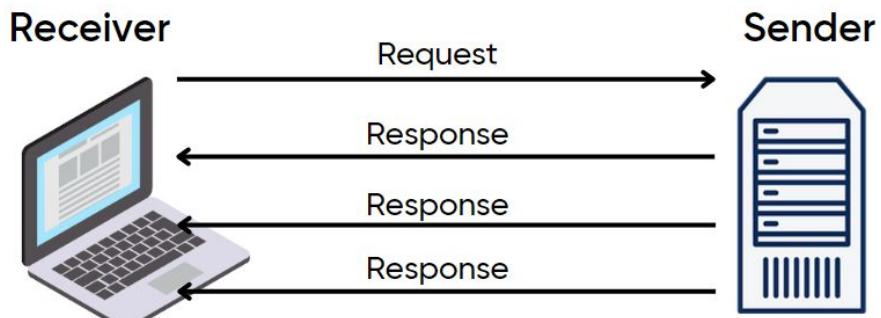
addrlen - dimensione della struttura dell'indirizzo

```
// connessione socket del client al socket del server  
if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {  
    printf("connection with the server failed...\n");  
    exit(0);  
}  
else  
    printf("connected to the server..\n");
```

UDP

E' uno dei principali protocolli di comunicazione insieme al TCP, e' definito connectionless, nel senso che lo scambio di pacchetti tra mittente e destinatario, non richiede l'operazione iniziale della creazione di un circuito su cui instradare il pacchetto.

Questo protocollo per la comunicazione spedisce i pacchetti non preoccupandosi di fare qualche controllo per vedere se i pacchetti sono arrivati tutti a destinazione, risultando la sua comunicazione veloce.



UDP

Vantaggi:

1. Connessione molto veloce
2. E' stateless
3. Ha la possibilita' di trasmettere in multicast e broadcast

Svantaggi:

1. Non controlla che i pacchetti siano arrivati a destinazione, producendo dei package lost
2. Non e' affidabile ne' stabile

Usi:

1. Giochi multiplayer online
2. Live stream
3. Protocolli di routing
4. Risoluzione dei nomi (protocollo: DNS)

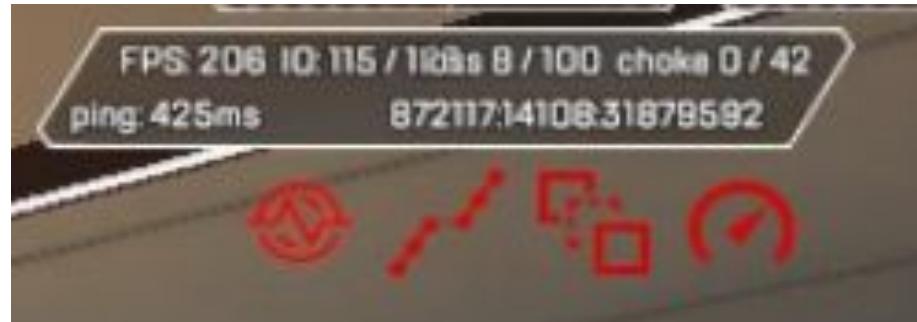


Immagine del gioco apex legends che mostra gli errori di connessione durante una partita

Struttura di un datagram UDP

E' formato da una parte chiamata header, che racchiude:

- porta di ingresso
- porta di destinazione
- la sua lunghezza
- la checksum che risulta essere il codice di controllo del pacchetto;

I dati del pacchetto :

Nei videogiochi il pacchetto può contenere per esempio il comando di movimento del giocatore.

Nelle lives il pacchetto può contenere una frase detta dallo streamer.



Socket UDP in Java - Struttura iniziale

Creare un'applicazione UDP in Java è molto simile a una TCP.

La differenza sostanziale sta nell'utilizzo dei pacchetti implementati nella libreria Java e il tipo di protocollo: “connection-less”.

Packet Utilizzati da Java:

```
import java.net.DatagramPacket;  
import java.net.DatagramSocket;
```



Ogni volta che implementeremo delle classi all'interno di file che dovranno utilizzare i socket, sarà obbligatorio importare queste classi.

Socket UDP in Java - Server

```
public class Server extends Thread {  
  
    private DatagramSocket socket;  
    private boolean running;  
    private byte[] buf = new byte[256];  
  
    public Server() {  
        try {  
            socket = new DatagramSocket(4445);  
        } catch (SocketException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Istanziamo:

- un socket che conterrà la porta di connessione;
- un boolean per verificare che siamo ancora in connessione;
- un buffer per gestire il pacchetto

Socket UDP in Java - Server

```
public void run() {
    running = true;

    while (running) {
        DatagramPacket packet = new DatagramPacket(buf, buf.length);
        try {
            socket.receive(packet);
        } catch (IOException e) {
            e.printStackTrace();
        }

        InetAddress address = packet.getAddress();
        int port = packet.getPort();
        packet = new DatagramPacket(buf, buf.length, address, port);
        String received
            = new String(packet.getData(), 0, packet.getLength());

        if (received.equals("end")) {
            running = false;
            continue;
        }
        try {
            socket.send(packet);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    socket.close();
}
```

Il metodo run resterà in esecuzione finché la connessione non terminerà.

Inizialmente riceviamo il pacchetto con socket.receive(), ci salviamo l'ip da dove è arrivato il pacchetto con packet.getAddress() e in fine inviamo il nuovo pacchetto al client con socket.send() dopo aver controllato che non era un messaggio per terminare la connessione.

Socket UDP in Java - Client

```
9
0 public class Client {
1     private DatagramSocket socket;
2     private InetAddress address;
3
4     private byte[] buf;
5
6     public Client() {
7         try {
8             socket = new DatagramSocket();
9         } catch (SocketException e) {
0             e.printStackTrace();
1         }
2         try {
3             address = InetAddress.getByName("localhost");
4         } catch (UnknownHostException e) {
5             e.printStackTrace();
6         }
7     }
8 }
```

Istanziamo:

- un socket che conterrà la porta di connessione;
- un address che avrà l'ip del server a cui connettersi

Socket UDP in Java - Client

```
 5● public String sendEcho(String msg) {
 6     buf = msg.getBytes();
 7     DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
 8     try {
 9         socket.send(packet);
10     } catch (IOException e) {
11         e.printStackTrace();
12     }
13     packet = new DatagramPacket(buf, buf.length);
14     try {
15         socket.receive(packet);
16     } catch (IOException e) {
17         e.printStackTrace();
18     }
19     String received = new String(
20         packet.getData(), 0, packet.getLength());
21     return received;
22 }
23
24● public void close() {
25     socket.close();
26 }
27 }
```

Il metodo sendEcho richiede in input il messaggio da inviare al server che viene inserito nel packet.

In seguito viene inviato con socket.send() per poi farsi ritornare con socket.receive() il messaggio di risposta dal server.

Con il metodo close si interrompe la connessione

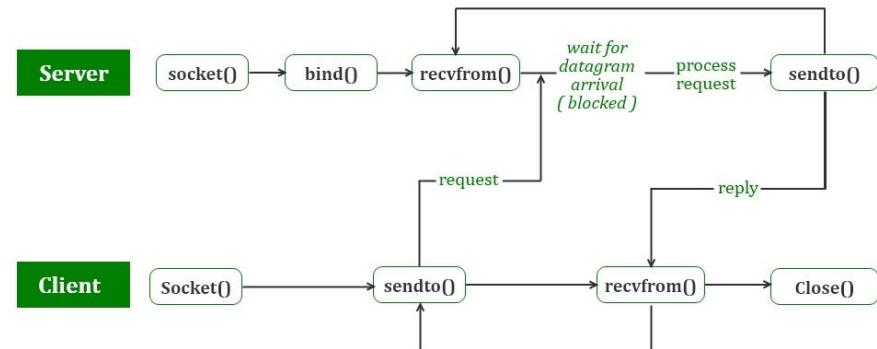
UDP in C

Sia nel server che nel client viene richiamate la funzione **socket()** che serve a creare un socket ancora non vincolato da un indirizzo.

Nel server successivamente viene chiamata la funzione **bind()** che ha proprio lo scopo di legare un socket con un indirizzo di rete e una porta.

Proprio a metà immagine inizia la comunicazione vera e propria, in questo loop possiamo notare che le principali funzioni che vengono utilizzate sono **sendto()** che serve per inviare un datagram dato un indirizzo e **recvfrom()** che blocca tutto finché non arriva un messaggio da qualcuno.

La connessione si conclude con **close()**



UDP in C

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen)
```

sockfd - descrittore del socket (ritornato dalla funzione socket())

buf - messaggio

len - dimensione del messaggio

flags - serie di flag che servono a modificare il comportamento del socket

dest_addr - struttura contenente l'indirizzo del destinatario

addrlen - dimensione della struttura dell'indirizzo del destinatario

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen)
```

sockfd - descrittore del socket

buf - variabile dove contenere il messaggio ricevuto

len - dimensione del messaggio ricevuto

flags - serie di flag che servono a modificare il comportamento del socket

src_addr - struttura contenente l'indirizzo del mittente

addrlen - dimensione della struttura dell'indirizzo del mittente

```
sendto(sockfd, (const char *)hello, strlen(hello),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
        sizeof(servaddr));
printf("Hello message sent.\n");
```

```
n = recvfrom(sockfd, (char *)buffer, MAXLINE,
              MSG_WAITALL, (struct sockaddr *) &servaddr,
              &len);
```

```

int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from server";
    struct sockaddr_in servaddr, cliaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    // Filling server information
    servaddr.sin_family      = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORT);

    // Bind the socket with the server address
    if ( bind(sockfd, (const struct sockaddr *)&servaddr,
               sizeof(servaddr)) < 0 )
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    int len, n;

    len = sizeof(cliaddr); //len is value/result

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                  MSG_WAITALL, ( struct sockaddr *) &cliaddr,
                  &len);
    buffer[len] = '\0';
    printf("Client : %s\n", buffer);
    sendto(sockfd, (const char *)hello, strlen(hello),
           MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
           len);
    printf("Hello message sent.\n");

    return 0;
}

```

UDP in C - Server

Dopo aver creato un socket utilizzando la funzione **socket()** viene ritornato un descrittore(sockfd) che deve essere conservato in una variabile intera.

Due strutture che descrivono rispettivamente l'indirizzo del socket e del client e si inizializzano entrambe richiamando la funzione **memset()** per poi riempire la struttura destinata al server con le rispettive informazioni(**tipo di indirizzo, indirizzo ip, porta**).

Il passo successivo comporta binding del **Server socket** all'indirizzo che si è definito nel passo precedente con la funzione **bind()**.

Nel finale semplicemente si resta in ascolto del messaggio inviato dal client con la funzione **recvfrom()** e si risponde con un messaggio di saluto utilizzando **sendto()**.

```
int main() {
    int sockfd;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in servaddr;

    // Creating socket file descriptor
    if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = INADDR_ANY;

    int n, len;

    sendto(sockfd, (const char *)hello, strlen(hello),
            MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
    printf("Hello message sent.\n");

    n = recvfrom(sockfd, (char *)buffer, MAXLINE,
                  MSG_WAITALL, (struct sockaddr *) &servaddr,
                  &len);
    buffer[n] = '\0';
    printf("Server : %s\n", buffer);

    close(sockfd);
    return 0;
}
```

UDP in C - Client

Nel client i passi da seguire sono gli stessi che sono stati descritti nella parte del server se non per il fatto che nel client non bisogna richiamare la funzione **bind()** e quindi fare il binding.

In [questa pagina](#) potete trovare un esempio.

Ping, DNS e tracert

Ping è una networking utility usata per misurare il round-trip time di uno o più pacchetti ICMP inviati da un host a un server.

ICMP (Internet Control Message Protocol) è un protocollo di servizio che i computer usano per inviare informazioni riguardo a malfunzionamenti della rete, informazioni di controllo e comunicazioni varie.

```
greyrat@ghost: ~
└─ ping google.it -c 10
PING google.it (142.250.180.131) 56(84) bytes of data.
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=1 ttl=111 time=45.0 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=2 ttl=111 time=269 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=3 ttl=111 time=42.9 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=4 ttl=111 time=47.5 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=5 ttl=111 time=44.5 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=6 ttl=111 time=82.6 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=7 ttl=111 time=37.3 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=8 ttl=111 time=59.9 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=9 ttl=111 time=58.8 ms
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=10 ttl=111 time=58.7 ms

— google.it ping statistics —
10 packets transmitted, 10 received, 0% packet loss, time 9010ms
rtt min/avg/max/mdev = 37.286/74.595/268.717/65.861 ms
```

Esempio di ping da un computer a google.it, da terminale di Linux

Ping

```
ping google.it -c 10
```

L'argomento **obbligatorio** è il **domain o IP** del destinatario. L'argomento opzionale **-c** indica il numero di pacchetti da ricevere prima di terminare l'operazione, in questo caso 10.

```
64 bytes from mil04s43-in-f3.1e100.net (142.250.180.131): icmp_seq=1 ttl=111 time=45.0 ms
```

Queste sono le informazioni restituite da un singolo ping in ritorno.

64 bytes - dimensione del pacchetto ICMP ricevuto

mil04s43-in-f3.1e100.net - vero domain dei server Google

142.250.180.131 - IP pubblico dei server Google

icmp_seq=1 - numero di sequenza del pacchetto

ttl=111 - Time to Live, ovvero il limite di hop rimanente

time=45.0 ms - Round-trip time del pacchetto

```
— google.it ping statistics —
10 packets transmitted, 10 received, 0% packet loss, time 9010ms
rtt min/avg/max/mdev = 37.286/74.595/268.717/65.861 ms
```

Risultati finali: pacchetti trasmessi e ricevuti, percentuale di pacchetti persi, tempo totale di esecuzione, minimo, media, massimo e deviazione standard del round-trip time

Ping

In realtà ping non fa altro che inviare un pacchetto ICMP di tipo “echo request” e misurare il tempo fino all’arrivo del corrispettivo pacchetto “echo reply”.

Per tradurre un dominio in un IP viene utilizzato DNS, spiegato più avanti nella presentazione dai compagni.

Esistono diverse implementazioni di ping (è coperto da licenza BSD), ad esempio quella di Linux e quella di Windows, che hanno alcune differenze tecniche.

Inoltre esistono implementazioni con il supporto per IPv6 tramite ICMPv6.

Il nome deriva dal suono tipico dei sonar, ma è anche un acronimo di Packet InterNet Groper.



Mike Muuss, il ricercatore inventore di ping

Ping in C

Per poter ricevere il valore di ping da un host destinatario, in questo programma, abbiamo bisogno di:

- un file descriptor di un socket opportuno(3);
- un pacchetto da mandare(64 bytes);
- l'hostname del destinatario(bom07...100.net);
- l'indirizzo ip del destinatario(172.217.27.206);
- il dominio del destinatario(google.com).

Fonte:

<https://www.geeksforgeeks.org/ping-in-c/>

un possibile output del programma →

```
Resolving DNS..  
  
Trying to connect to 'google.com' IP: 172.217.27.206  
  
Reverse Lookup domain: bom07s15-in-f14.1e100.net  
Socket file descriptor 3 received  
  
Socket set to TTL..  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=1 ttl=64 rtt = 57.320584 ms.  
  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=2 ttl=64 rtt = 58.666775 ms.  
  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=3 ttl=64 rtt = 58.081148 ms.  
  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=4 ttl=64 rtt = 58.700630 ms.  
  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=5 ttl=64 rtt = 58.281802 ms.  
  
64 bytes from bom07s15-in-f14.1e100.net (h: google.com) (172.217.27.206)  
msg_seq=6 ttl=64 rtt = 58.360916 ms.  
  
==172.217.27.206 ping statistics==  
6 packets sent, 6 packets received, 0.000000 percent packet loss.  
Total time: 6295.187804 ms.
```

Logica del programma

controlla che sia dato correttamente l'input da riga di comando;

ricava l'indirizzo ip del destinatario dal dominio utilizzando il DNS e poi controlla che esista;

ricava l'hostname del destinatario dall'indirizzo ip tramite il DNS;

crea un file descriptor di un socket da poter utilizzare e controlla che sia valido;

associa una funzione che interrompe il loop per il ping che sia attiva quando viene premuto ctrl+c;

crea un loop che manda un pacchetto verso la destinazione e mostra il valore di ping ogni tot;

```
int main(int argc, char *argv[])
{
    int sockfd;
    char *ip_addr, *reverse_hostname;
    struct sockaddr_in addr_con;
    int addrlen = sizeof(addr_con);
    char net_buf[NI_MAXHOST];

    if(argc!=2)
    {
        printf("\nFormat %s <address>\n", argv[0]);
        return 0;
    }

    ip_addr = dns_lookup(argv[1], &addr_con);
    if(ip_addr==NULL)
    {
        printf("\nDNS lookup failed! Could not resolve hostname!\n");
        return 0;
    }

    reverse_hostname = reverse_dns_lookup(ip_addr);
    printf("\nTrying to connect to '%s' IP: %s\n",
          argv[1], ip_addr);
    printf("\nReverse Lookup domain: %s",
          reverse_hostname);

    sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if(sockfd<0)
    {
        printf("\nSocket file descriptor not received!!\n");
        return 0;
    }
    else
        printf("\nSocket file descriptor %d received\n", sockfd);

    signal(SIGINT, intHandler);

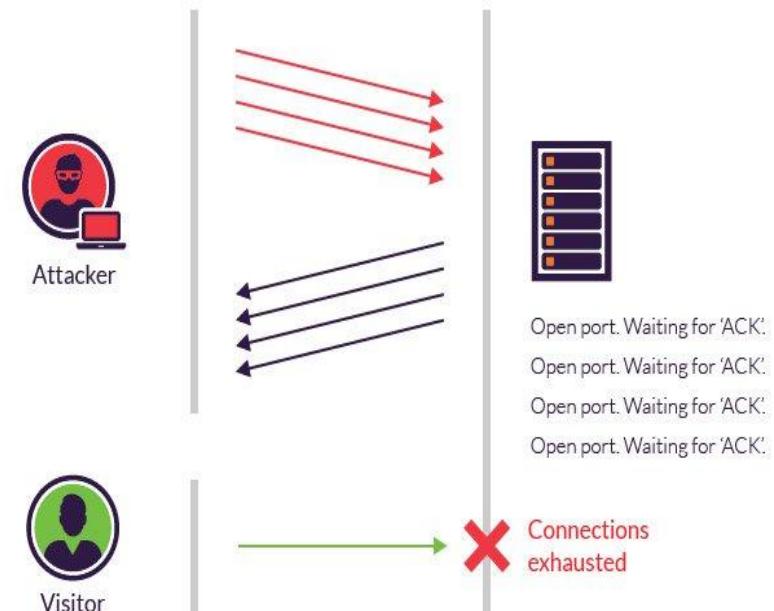
    send_ping(sockfd, &addr_con, reverse_hostname,
              ip_addr, argv[1]);

    return 0;
}
```

Raw Socket

Il raw socket è un tipo di socket che permette, ad un software, di scambiare informazioni in rete direttamente con il destinatario, evitando interventi da parte del sistema operativo o da altri programmi.

Alcuni software limitano o negano l'utilizzo di questo socket, perché spesso utilizzati da parte di cracker per effettuare attacchi TCP su una rete.



come avviene un attacco TCP

Il protocollo DNS

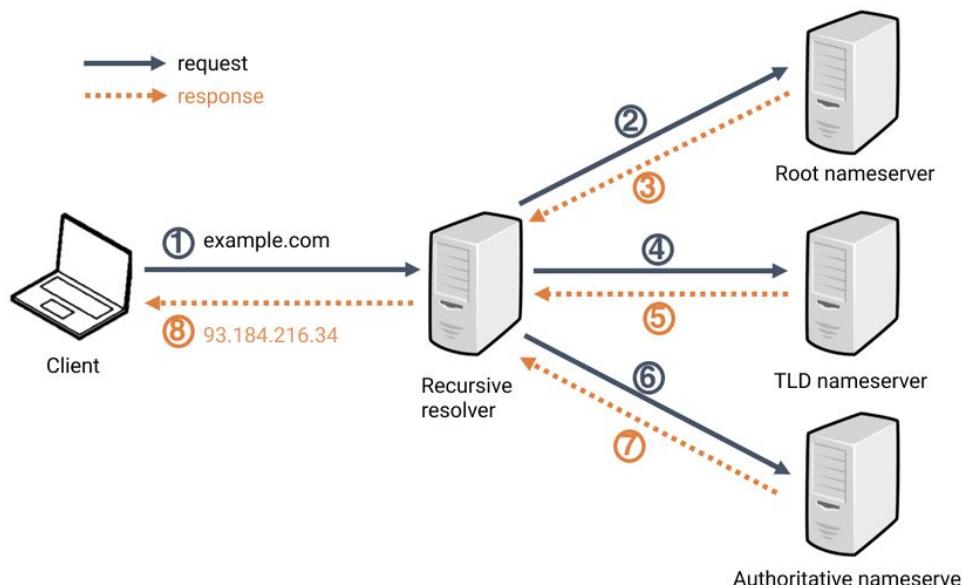
Il DNS (Domain Name System) è il protocollo che permette di ottenere l'indirizzo IP di una macchina a partire da un dominio.



È un componente fondamentale per il funzionamento di Internet a partire dal 1985.

Come funziona il DNS

La risoluzione di un dominio col DNS è fatta in modo ricorsivo attraversando una gerarchia di server fino ad arrivare al dominio richiesto.



Una volta raggiunto il dominio richiesto, viene restituito l'indirizzo IP associato ed il computer è in grado di connettersi alla macchina.

Query: come è strutturata?

La query è una richiesta effettuata da un computer per ottenere un indirizzo IP a partire da un dominio. È solitamente gestita da un resolver.

	45 6.730748849	192.168.1.96	192.168.1.1	DNS	81 Standard query 0x5d69 A google.com OPT
	46 6.752417935	192.168.1.1	192.168.1.96	DNS	97 Standard query response 0x5d69 A googl
▶ Frame 45: 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface wlp2s0, id 0					
0000	a4 91 b1 7d 51 63 3c 95 09 ad 65 6f 08 00 45 00				... }Qc< .eo .E
0010	00 43 90 b6 00 00 40 11 66 42 c0 a8 01 60 c0 a8				.C .@. fB
0020	01 01 d4 9b 00 35 00 2f 83 f2 5d 69 01 00 00 01			 5 / .]i ..
0030	00 00 00 00 00 01 06 67 6f 6f 67 6c 65 03 63 6f			 g oogle.co
0040	6d 00 00 01 00 01 00 00 29 05 c0 00 00 00 00 00 00				m ..)
0050	00				

Intestazione (Header)

Transaction ID: (2 byte)

Flags: 0100 è standard query

Questions: L'indirizzo che si richiede

Answer RRs: il numero delle risposte

Authority RRs: il numero delle risposte authority

Query

Name

Type: 0001, A per host

Class: 0001 per INternet

Risposta: come è strutturata?

La risposta è identica alla query fatta eccezione per ulteriori record che aggiungono informazioni

L	46 6.752417935	192.168.1.1	192.168.1.96	DNS	97 Standard query response 0x5d69 A google.com A 142.250.184.78 OPT
▼ google.com: type A, class IN, addr 142.250.184.78					
0000	3c 95 09 ad 65 6f a4 91	b1 7d 51 63 08 00 45 00	<... eo... }Qc E		
0010	00 53 42 d8 00 00 40 11	b4 10 c0 a8 01 01 c0 a8	.SB @.....		
0020	01 60 00 35 d4 9b 00 3f	c0 9a 5d 69 81 80 00 01	.^5 .? .]i		
0030	00 01 00 00 00 01 06 67	6f 6f 67 6c 65 03 63 6f g oogle.co		
0040	6d 00 00 01 00 01 c0 0c	00 01 00 01 00 00 00 4d	m.....(.....M		
0050	00 04 8e fa b8 4e 00 00	29 04 c4 00 00 00 00 00	(.....N ..)		
0060	00				

Answer

Name (prima)

Type di record (A, ...)

Class in genere IN
(INternet)

Time To Live (TTL): Il tempo che il risultato si può memorizzare in cache

Data Length: 4 byte

Address: L'indirizzo convertito in esadecimale

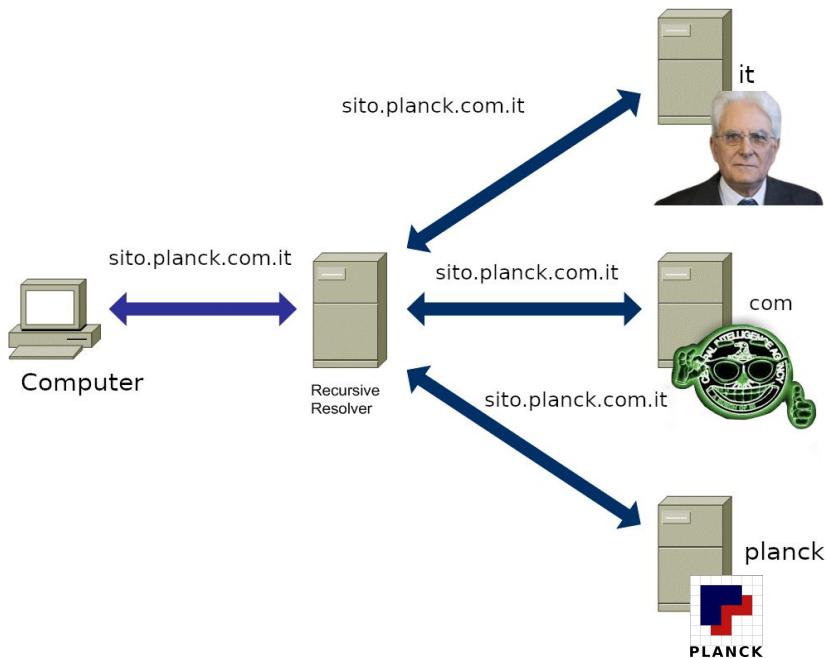
Tipi di DNS

- DNS-over-UDP/53 ("Do53")
- DNS-over-TCP/53 ("Do53/TCP")
- DNS-over-TLS ("DoT")
- DNS-over-HTTPS ("DoH")
- DNS-over-TOR

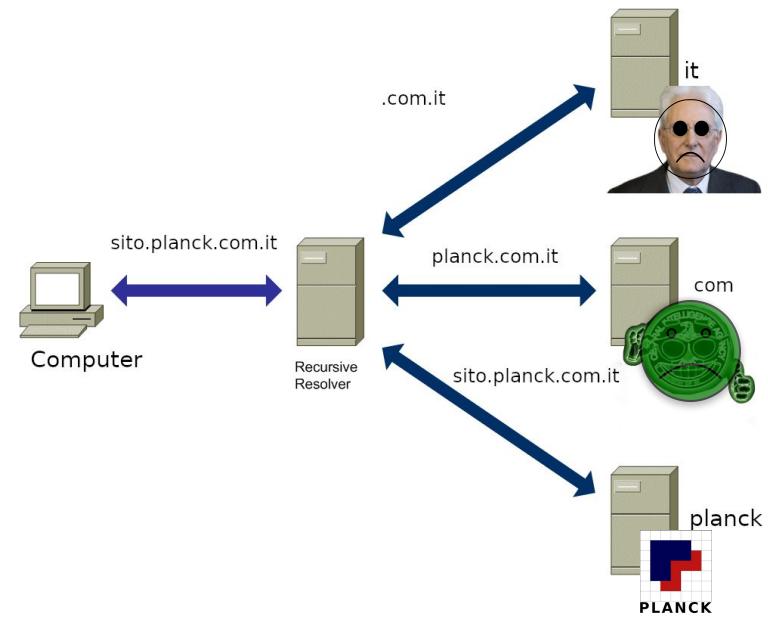


QName Minimization

Senza QName Minimization:



Con QName Minimization:



Esempio di codice in C

```
31// Struttura header DNS.
32struct DNS_HEADER
33{
34    unsigned short id; // Transaction ID.
35
36    unsigned char rd :1; // Recursion desired.
37    unsigned char tc :1; // Truncated message.
38    unsigned char aa :1; // Authoritive answer.
39    unsigned char opcode :4; // Purpose of message.
40    unsigned char qr :1; // query/response flag.
41
42    unsigned char rcode :4; // Response code.
43    unsigned char cd :1; // Checking disabled.
44    unsigned char ad :1; // Authenticated data.
45    unsigned char z :1; // Riservato.
46    unsigned char ra :1; // Recursion available.
47
48    unsigned short q_count; // Numero di record question.
49    unsigned short ans_count; // Numero di record risposta.
50    unsigned short auth_count; // Numero di record authority.
51    unsigned short add_count; // Numero di record addizionali.
52};
```

```
53
54// Campi di lunghezza costante della domanda.
55struct QUESTION
56{
57    unsigned short qtype;
58    unsigned short qclass;
59};
60
61// Campi di lunghezza costante del record di risorse.
62#pragma pack(push, 1)
63struct R_DATA
64{
65    unsigned short type;
66    unsigned short class;
67    unsigned int ttl;
68    unsigned short data_len;
69};
70#pragma pack(pop)
71
72// Puntatori di un record di risorse.
73struct RES_RECORD
74{
75    unsigned char *name;
76    struct R_DATA *resource;
77    unsigned char *rdata;
78};
79
80// Struttura di una query.
81typedef struct
82{
83    unsigned char *name;
84    struct QUESTION *ques;
85} QUERY;
```

```

120 s = socket(AF_INET , SOCK_DGRAM , IPPROTO_UDP); // Socket UDP per le query DNS
121 dest.sin_family = AF_INET;
122 dest.sin_port = htons(53);
123 dest.sin_addr.s_addr = inet_addr(dns_servers[0]); // Server DNS.
124
125 // Impostare la struttura DNS ad standard queries.
126 dns = (struct DNS_HEADER *)&buf;
127
128 dns->id = (unsigned short) htons(getpid());
129 dns->qr = 0; // Questa è una query.
130 dns->opcode = 0; // Questa è una query standard.
131 dns->aa = 0; // Non authoritative.
132 dns->tc = 0; // Il messaggio non è troncato.
133 dns->rd = 1; // Ricorsione desiderata.
134 dns->ra = 0; // Ricorsione non disponibile! Non la abbiamo (lol).
135 dns->z = 0;
136 dns->ad = 0;
137 dns->cd = 0;
138 dns->rcode = 0;
139 dns->q_count = htons(1); // Abbiamo solo una domanda.
140 dns->ans_count = 0;
141 dns->auth_count = 0;
142 dns->add_count = 0;
143

```

```

388 // Questo converte www.google.com to \www\6g\oo\gle\3\com.
389 void ChangetoDnsNameFormat(unsigned char* dns,unsigned char* host)
390{
391     int lock = 0 , i;
392     strcat((char*)host,".");
393
394     for(i = 0 ; i < strlen((char*)host) ; i++)
395    {
396        if(host[i]=='.')
397        {
398            *dns++ = i-lock;
399            for(;lock<i;lock++)
400            {
401                *dns++=host[lock];
402            }
403            lock++; // o lock=i+1;
404        }
405    }
406    *dns++='\0';
407}

```

```

// Puntare alla posizione della query.
qname =(unsigned char*)&buf[sizeof(struct DNS_HEADER)];

ChangetoDnsNameFormat(qname , host);
qinfo =(struct QUESTION*)&buf[sizeof(struct DNS_HEADER) + (strlen((const char*)qname) + 1)]; //fill it

qinfo->qtype = htons( query_type ); // Tipo della query: A , MX , CNAME , NS etc
qinfo->qclass = htons(1); // E internet (lol)

printf("\nSending Packet...");
if( sendto(s,(char*)buf,sizeof(struct DNS_HEADER) + (strlen((const char*)qname)+1) + sizeof(struct QUESTION),0,(struct sockaddr*)&dest,sizeof(dest)) < 0 )
{
    perror("sendto failed");
}
printf("Done");

```

Tracert

Tracert è un software applicato per mostrare i diversi percorsi ed eventuali ritardi/perdite di un pacchetto spedito con l'utilizzo del protocollo IP individuandone la posizione.

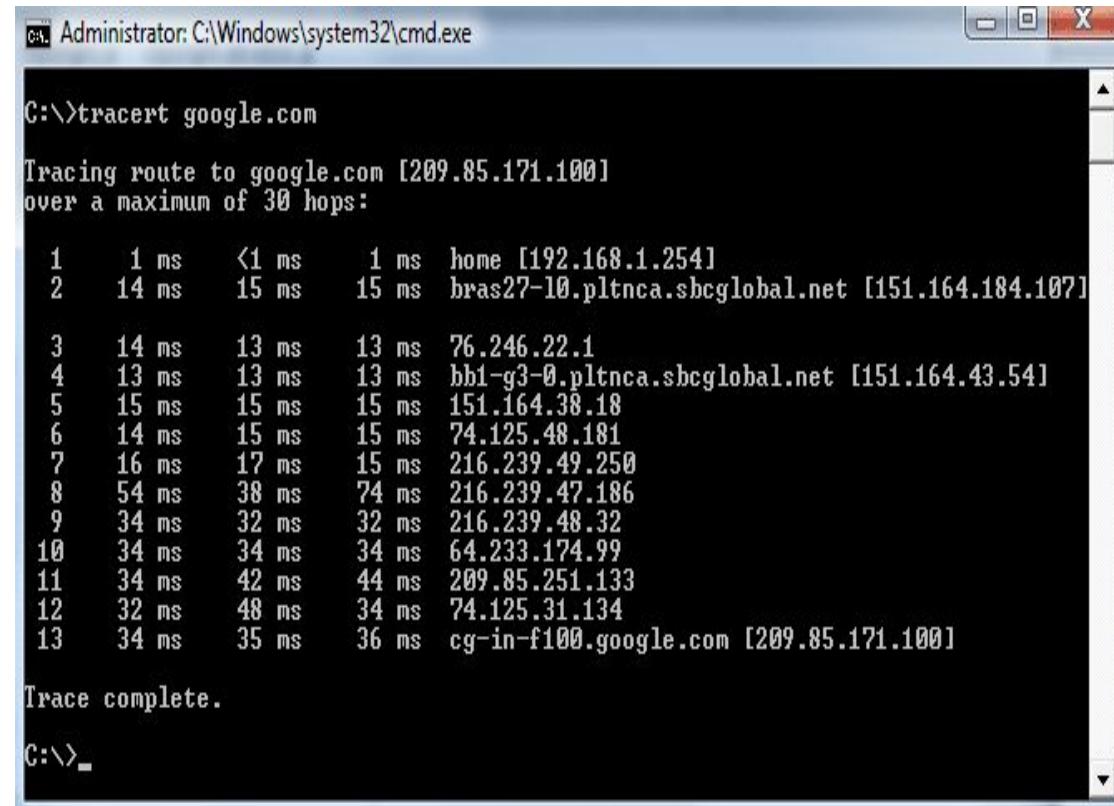


Funzionamento Tracert

Si inizia con il digitare il comando tracert seguito poi dall'indirizzo ip o del domain name completo dell'host remoto.

Una volta fatto ciò il pacchetto di dati viene spedito alla destinazione e ad ogni hop vengono inviate le informazioni quali l'ip e il tempo degli hop di quel router al mittente.

Traceroute ci aiuta quindi ad identificare DOVE si verifica un problema sulla rete tracciando il percorso da una posizione all'altra registrando ogni singolo hop eseguito nella route.



```
C:\>Administrator: C:\Windows\system32\cmd.exe
C:\>tracert google.com
Tracing route to google.com [209.85.171.100]
over a maximum of 30 hops:
1  1 ms    <1 ms    1 ms  home [192.168.1.254]
2  14 ms   15 ms   15 ms  bras27-10.pltnca.sbcglobal.net [151.164.184.107]
3  14 ms   13 ms   13 ms  76.246.22.1
4  13 ms   13 ms   13 ms  bb1-g3-0.pltnca.sbcglobal.net [151.164.43.54]
5  15 ms   15 ms   15 ms  151.164.38.18
6  14 ms   15 ms   15 ms  74.125.48.181
7  16 ms   17 ms   15 ms  216.239.49.250
8  54 ms   38 ms   74 ms  216.239.47.186
9  34 ms   32 ms   32 ms  216.239.48.32
10 34 ms   34 ms   34 ms  64.233.174.99
11 34 ms   42 ms   44 ms  209.85.251.133
12 32 ms   48 ms   34 ms  74.125.31.134
13 34 ms   35 ms   36 ms  cg-in-f100.google.com [209.85.171.100]

Trace complete.
C:\>
```

L'utilizzo del TTL in Tracert

All'invio di un pacchetto ICMP il suo TTL è impostato ad 1 così che il primo router incontrato rinvia un TTL scaduto nei messaggi di transito;

Questo messaggio contiene le informazioni necessarie per l'identificazione del router che lo ha prodotto, l'identificazione del router viene documentata e il pacchetto ICMP viene rinviaato ma con TTL a 2.

Così facendo il pacchetto ICMP raggiunge il secondo router prima della scadenza del valore di TTL, il processo viene poi ripetuto fino a che l'host viene raggiunto.

Con questo procedimento si ha un report di tutti i router e con le informazioni ricavate è possibile individuare i problemi lungo il percorso.



Possibili Problematiche

Se un pacchetto non viene tracciato entro un determinato periodo di tempo, l'informazione non disponibile sul tempo di risposta viene segnata con un "**",

e il router che attendeva la notifica ICMP non sarà identificabile.



CODICE TRACEROUTE

https://opensource.apple.com/source/network_cmds/network_cmds-77/traceroute.tproj/traceroute.c.auto.html

Decodifica comando inserito

```
while ((ch = getopt(argc, argv, "dm:np:q:rs:t:w:v")) != EOF)
    switch(ch) { //switch per decifrare la stringa inserita
    case 'd':
        options |= SO_DEBUG;
        break;
    case 'm': //ttl massimo, ovvero numero massimo di volte a cui si può accedere al pacchetto
        max_ttl = atoi(optarg);
        if (max_ttl <= 1) {
            Fprintf(stderr, "traceroute: max ttl must be >1.\n");
            exit(1);
        }
        break;
    case 'n': //stampa indirizzo numericamente
        nflag++;
        break;
    case 'p': //numero porta di uscita
        port = atoi(optarg);
        if (port < 1) {
            Fprintf(stderr, "traceroute: port must be >0.\n");
            exit(1);
        }
        break;
    case 'q': //numero di probes (pacchetti ip mandati)
        nprobes = atoi(optarg);
        if (nprobes < 1) {
            Fprintf(stderr, "traceroute: nprobes must be >0.\n");
            exit(1);
        }
        break;
```

Gestione invio, risposte e stampa

```
for (ttl = 1; ttl <= max_ttl; ++ttl) { //ciclo for per il numero di ttl massimi
    u_long lastaddr = 0;
    int got_there = 0;
    int unreachable = 0;

    Printf("%2d ", ttl);
    for (probe = 0; probe < nprobes; ++probe) {
        int cc;
        struct timeval t1, t2;
        struct timezone tz;
        struct ip *ip;

        (void) gettimeofday(&t1, &tz);
        send_probe(++seq, ttl); //invio probe
        while (cc = wait_for_reply(s, &from)) { //attesa della risposta
            (void) gettimeofday(&t2, &tz);
            if ((i = packet_ok(packet, cc, &from, seq))) { //controllo correttezza pacchetto
                if (from.sin_addr.s_addr != lastaddr) {
                    print(packet, cc, &from); //stampa del pacchetto
                    lastaddr = from.sin_addr.s_addr;
                }
            }
            Printf(" %g ms", deltaT(&t1, &t2)); //stampa tempo di risposta
        }
    }
}
```

Funzione invio pacchetto

```
void send_probe(seq, ttl)
    int seq, ttl;
{
    struct opacket *op = outpacket;
    struct ip *ip = &op->ip;
    struct udphdr *up = &op->udp;
    int i;

    //imposta le informazioni all'interno del pacchetto da mandare

    ip->ip_off = 0;
    ip->ip_hl = sizeof(*ip) >> 2;
    ip->ip_p = IPPROTO_UDP;
    ip->ip_len = datalen;
    ip->ip_ttl = ttl;
    ip->ip_v = IPVERSION;
    ip->ip_id = htons(ident+seq);

    up->uh_sport = htons(ident);
    up->uh_dport = htons(port+seq);
    up->uh_ulen = htons((u_short)(datalen - sizeof(struct ip)));
    up->uh_sum = 0;

    op->seq = seq;
    op->ttl = ttl;
    (void) gettimeofday(&op->tv, &tz);

    i = sendto(sndsock, (char *)outpacket, datalen, 0, &whereto, sizeof(struct sockaddr));
    if (i < 0 || i != datalen) {
        if (i<0)
            perror("sendto");
        printf("traceroute: wrote %s %d chars, ret=%d\n",
               hostname,
               datalen, i);
        (void) fflush(stdout);
    }
}
```

Funzione che attende la risposta

```
int
wait_for_reply(sock, from)
    int sock;
    struct sockaddr_in *from;
{
    fd_set fds;
    struct timeval wait;
    int cc = 0;
    int fromlen = sizeof (*from);

    FD_ZERO(&fds);
    FD_SET(sock, &fds);
    wait.tv_sec = waittime; wait.tv_usec = 0; //attendo 5 secondi per le risposte

    if (select(sock+1, &fds, (fd_set *)0, (fd_set *)0, &wait) > 0)
        cc=recvfrom(s, (char *)packet, sizeof(packet), 0, (struct sockaddr *)from, &fromlen); // ricezione pacchetto

    return(cc);
}
```

Risultato esecuzione

```
[ettoregreco@MacBook-Pro ~ % traceroute -m 20 8.8.8.8 52
traceroute to 8.8.8.8 (8.8.8.8), 20 hops max, 52 byte packets
 1  h388x (192.168.1.1)  12.568 ms  3.112 ms  3.035 ms
 2  * * *
 3  172.17.217.104 (172.17.217.104)  16.627 ms  6.037 ms  9.209 ms
 4  172.17.216.80 (172.17.216.80)  6.348 ms  7.719 ms
     172.17.217.44 (172.17.217.44)  6.838 ms
 5  172.19.184.8 (172.19.184.8)  11.048 ms
     172.19.184.14 (172.19.184.14)  8.347 ms
     172.19.184.8 (172.19.184.8)  11.948 ms
 6  172.19.177.28 (172.19.177.28)  16.569 ms
     172.19.177.16 (172.19.177.16)  15.797 ms
     172.19.177.26 (172.19.177.26)  9.212 ms
 7  195.22.205.116 (195.22.205.116)  13.775 ms
     195.22.205.98 (195.22.205.98)  12.676 ms
     ae48.milano11.mil.seabone.net (195.22.192.144)  11.404 ms
 8  72.14.219.236 (72.14.219.236)  14.881 ms
     142.250.165.98 (142.250.165.98)  15.703 ms
     142.250.168.148 (142.250.168.148)  13.619 ms
 9  * * *
10  108.170.232.169 (108.170.232.169)  18.548 ms
     142.251.50.137 (142.251.50.137)  12.827 ms
     142.251.235.179 (142.251.235.179)  17.614 ms
11  dns.google (8.8.8.8)  11.088 ms  12.669 ms  13.090 ms
```