RSO: Reti

spitfire

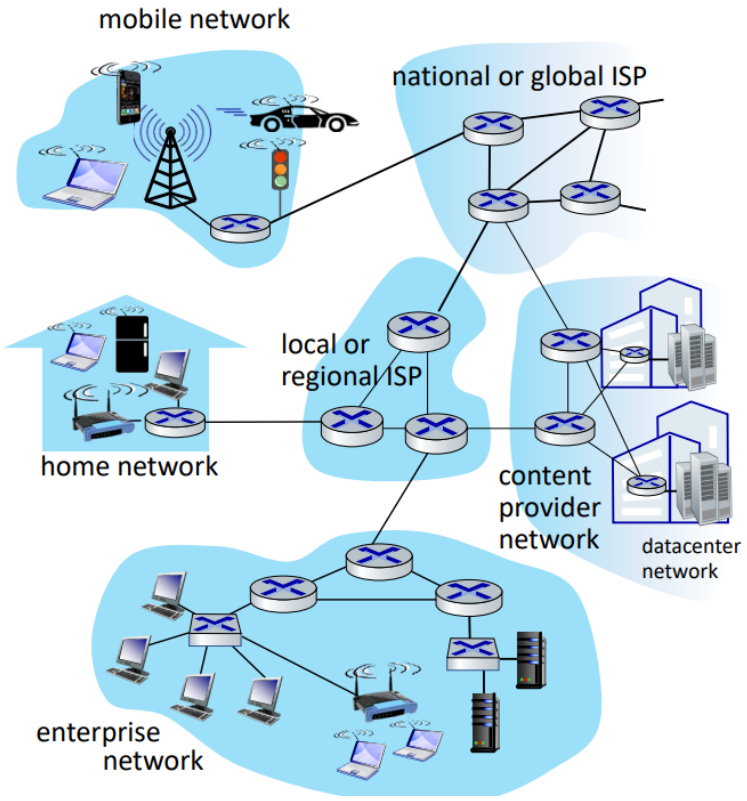
A.A. 2024-2025

Contents

1 Introduzione

Se vogliamo dare una visione "d'insieme" di internet possiamo pensarlo come formato dalle seguenti componenti:

- Miliardi di **calcolatori** connessi:
  - **Hosts**: dispositivi di computazione e sistemi periferici
  - Sono sistemi che eseguono **applicazioni di rete** al "confine" della rete
- **Packet switches**: inoltrano i pacchetti ("pezzi" di dati) tra diversi nodi di rete
  - Router, switches, ...
  - Internet è una **rete a commutazione di pacchetto**
- **Communication links**: I collegamenti fra i veri nodi della rede
  - Fibra, rame, radio, satellite...
  - **Transmission rate**: capacità, in termini di bit/s, che il canale può supportare ("larghezza di banda").
- **Networks**: Collezioni di dispositivi, router, switches e links gestiti **tutti da una stessa organizzazione**
  - Reti residenziali, enterprise ecc... vengono dette solitamente **reti di accesso**, perché sono quelle reti che raccolgono il traffico dagli utenti per mandarlo in rete o viceversa.



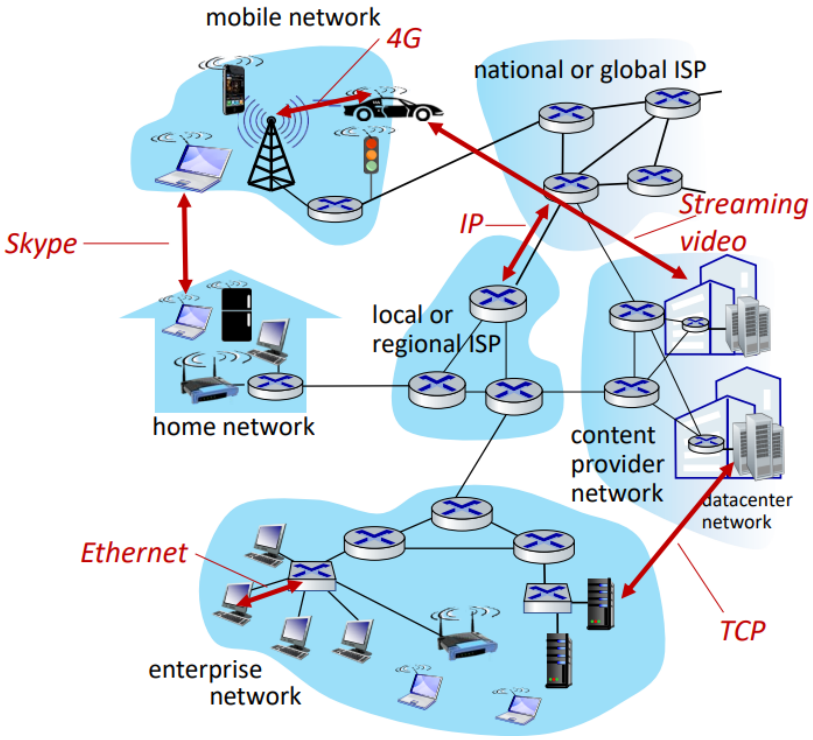
Internet è fisicamente una **rete connessa**, tuttavia vi sono dei sistemi che permettono di filtrare il traffico (firewall ecc...) per impedire che ogni nodo della rete sia accessibile da un qualsiasi altro nodo. Internet è quindi una **rete di reti** che interconnette le reti degli **Internet Service Providers** (ISP), cioè quelle entità che forniscono servizi di connettività. Il funzionamento della rete internet è governato dai **protocolli di comunicazione**:

- Controllano il modo in cui avviene l'invio e il ricevimento dei messaggi
- Esempi sono i protocolli HTTP (web), TCP, IP, WiFi, 4G, Ethernet ecc..

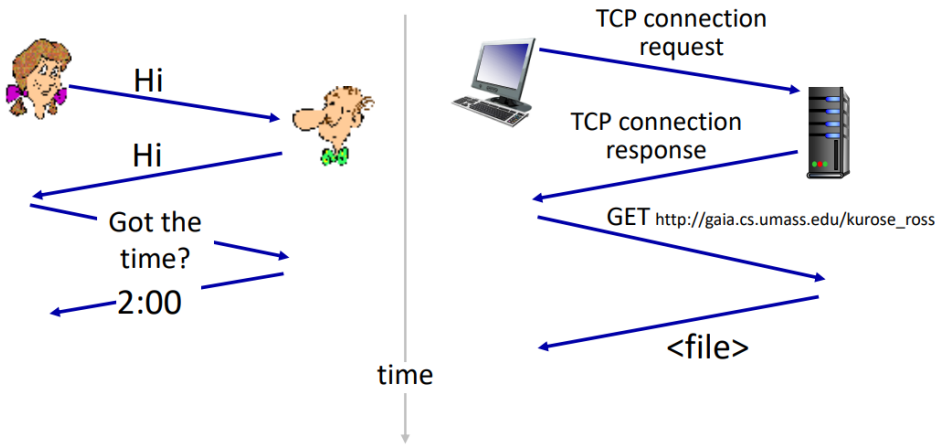
Protocolli di tipo diverso servono per **far comunicare dispositivi di tipo diverso**. Poiché il contesto delle reti è quindi molto eterogeneo, il tutto riesce a funzionare grazie agli **standard**. Esistono diversi enti di standardizzazione, tra cui citiamo:

- **RFC**: Request for Comments, documenti
- **IETF**: Internet Engineering Task Force; rilascia le RFC

Il compito degli enti di standardizzazione è quello di rilasciare documenti che vanno a definire le caratteristiche dei protocolli e delle architetture. Possiamo tuttavia vedere internet anche dal punto di vista dei **servizi**: internet può essere quindi vista come una **infrastruttura che offre dei servizi di connettività alle applicazioni distribuite**. Quindi, internet viene vista come una infrastruttura che **offre dei servizi di connettività tra nodi diversi**.



Abbiamo detto che la rete è governata da protocolli, ma **qual'è la definizione formale di protocollo?** Una definizione formale può essere la seguente: i **protocolli** definiscono il **formato, l'ordine** dei **messaggi inviati e ricevuti** tra le entità di rete e le **azioni intraprese** alla trasmissione e alla ricezione di un messaggio.



1.1 Introduzione alla struttura di internet

La struttura di alto livello di internet si può formalizzare nel seguente modo:

- **Network edge**: il confine della rete, comprende
  - **Hosts**: client e servers
  - I servers sono spesso in **data centers**

  È oggetto di dibattito se includere il confine della rete nella struttura di internet o meno; ciò nonostante, rimane comunque una componente fondamentale.
- **Access Networks**: Sono tutte quelle reti che servono a raccogliere il traffico generato e destinato per gli utenti. Sono quindi i **punti di accesso alla rete per gli utenti**. Esse possono essere **cablate oppure wireless**.
- **Network Core**: È l'insieme di tutte quelle reti che sono composte da router interconnessi che permettono di realizzare il concetto di **rete di reti**. Il suo compito è quello di **connettere le reti di accesso fra di loro** e comprendono tutte quelle reti che, su larga scala, **permettono il funzionamento di internet**.

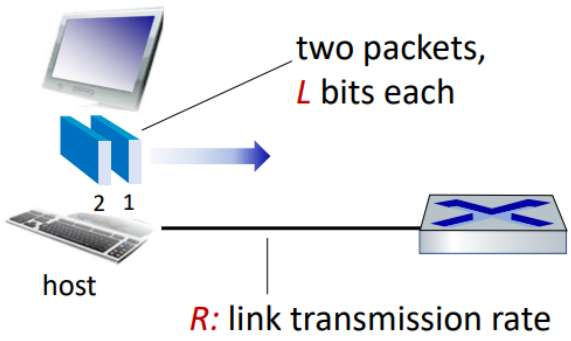
Ogni componente della struttura di internet viene detto **segmento**.

1.1.1 Network Edge

Il confine della rete è **popolato dagli host**. Il suo compito principale è quello di inviare i **pacchetti** generati dagli host (e quindi dagli utenti). Una visione ad alto livello di questo procedimento è:

- Il **messaggio applicativo** viene generato dall'host
- Esso viene **spezzettato in pezzetti**, chiamati **pacchetti**, di lunghezza L bits (questa cosa non è sempre vera: ci sono casi in cui i pacchetti hanno lunghezza variabile). Ognuno di questi pacchetti presenta una **intestazione**, cioè un determinato numero di bit che servono per permettere il funzionamento dei protocolli in rete

- Una volta generati i pacchetti, essi vengono trasmessi alla rete d'accesso ad un **tasso di trasmissione** (transmission rate) R, il quale è condizionato dalla **capacità di trasmissione del collegamento**



Ogni volta che invio un pacchetto in rete, ho un **ritardo di trasmissione**, il quale è il tempo necessario per trasmettere un pacchetto di L bit su un collegamento che ha capacità di trasporto di R bit/sec. Quindi:

$$\text{packet transmission delay} = \frac{L\ (bits)}{R\ (bits/sec)}$$

1.1.2 Access Networks

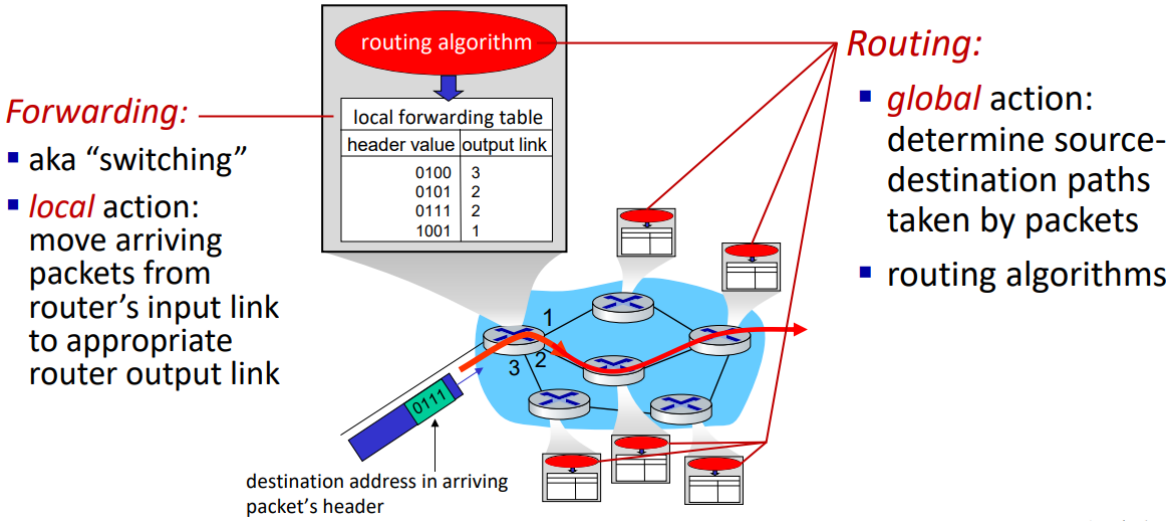
Come facciamo a **connettere gli host a "internet"**? Il compito di effettuare questa connessione è delle reti di accesso. Esse possono essere:

- Reti di accesso **residenziali**
- Reti di accesso **istituzionali**
- Reti di accesso **mobili**

1.1.3 Network Core

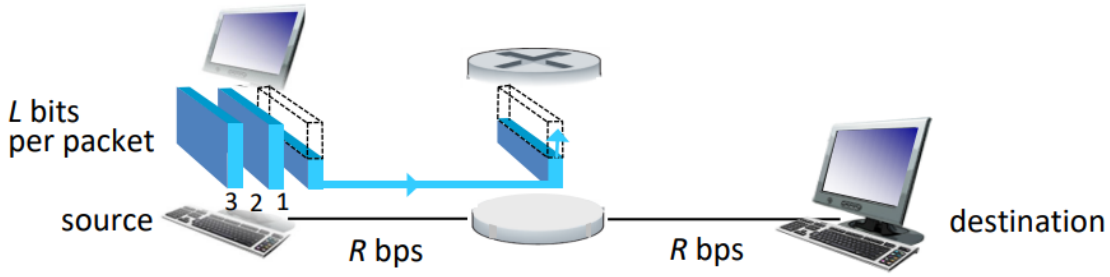
La Network Core è un insieme di **maglie di rete interconnesse**, le quali sono fondamentali per effettuare le operazioni di commutazione dei pacchetti. Esse garantiscono che i pacchetti possano essere trasmessi da un router verso l'altro attraverso dei collegamenti da una sorgente a una destinazione. Le reti di core presentano due **funzionalità fondamentali**: per spiegarle, dobbiamo prima capire **come funziona un router**: esso possiede al suo interno una tabella chiamata **tabella di inoltro** (forwarding table), che indica verso quale collegamento un pacchetto deve essere instradato in base al valore della sua intestazione (che quindi contiene, in termini generici, a chi deve essere recapitato questo pacchetto). L'operazione di **inoltro** (o **commutazione di pacchetto**) quindi consiste nell'invio sulla connessione corretta del pacchetto in arrivo (questa operazione viene anche detta **switching**). L'inoltro ha **valenza locale**: ogni router prende in considerazione solo la propria tabella di inoltro locale per decidere dove inoltrare un pacchetto. Tuttavia, questa operazione non basta per garantire che io possa raggiungere la destinazione corretta; per garantirlo dobbiamo effettuare un'altra operazione che prende il nome di **instradamento** (routing). Il routing è un'operazione **globale** che è

utilizzata per determinare **quale percorso, fra sorgente e destinazione, devono attraversare i pacchetti**. Ogni singolo router esegui quindi dei **protocolli** e degli **algoritmi di routing distribuiti** che hanno l'obbietto di **popolare le tabelle di inoltro** (viene effettuato tramite algoritmi su grafo). Poiché gli algoritmi di routing sono **distribuiti**, essi richiedono lo **scambio di messaggi e la collaborazione tra i router**.



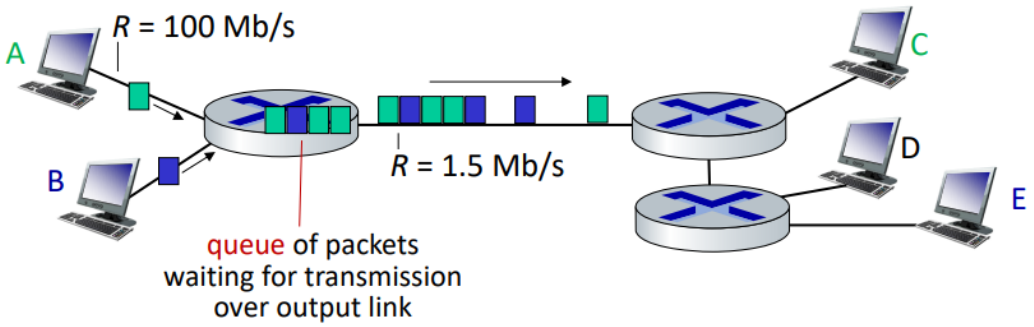
1.2 Packet Switching: Store-and-Forward

Nelle reti a commutazione di pacchetto, **ogni pacchetto ha vita propria**: una volta che il messaggio viene spezzettato in pacchetti, ognuno di esso ha un ciclo di vita indipendente dagli altri. La modalità di trasmissione usata da tutte le reti a commutazione di pacchetto è quella del **store-and-forward**: quando un pacchetto arriva ad un commutatore per essere commutato, esso deve **arrivate interamente prima di essere trasmesso al nodo successivo**. Tuttavia, si può pensare di saltare questo passaggio e trasmettere direttamente ogni bit che arriva su un certo nodo al successivo. Perché non viene fatto nelle reti a commutazione di pacchetto? Il motivo risiede nelle **intestazioni dei pacchetti**: infatti, è li che trovo **tutte le informazioni necessarie per gestire un pacchetto**, quindi devo aspettare che essa venga trasmessa per intero prima di procedere all'invio al nodo successivo.



1.3 Packet-Switching: queuing

Seppur Store-and-Forward semplifichi di molto la gestione delle reti, esso introduce una problematica che prende il nome di **accodamento** (queueing). È una problematica che esiste in tutte le reti e si manifesta quando il **tasso di arrivo dei pacchetti è superiore al tasso di trasmissione in uscita**. Nei router e negli switch, i pacchetti si accodano in aree di memoria apposite dette **buffer**; un pacchetto rimane nel buffer fino a quando non verrà trasmesso.

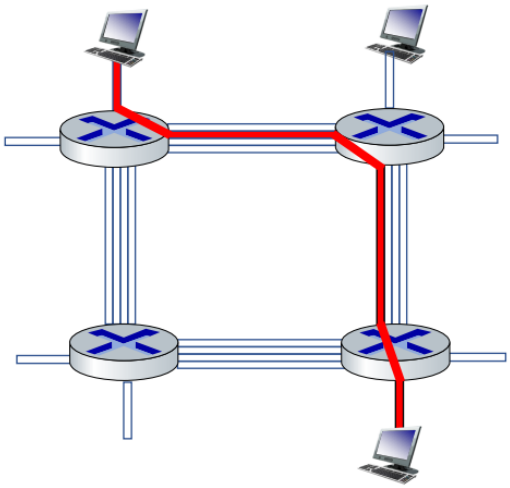


Questa è una condizione che può accadere nelle reti a commutazione di pacchetto, tuttavia **non può essere una situazione sistematica**: se si verificasse costantemente, la dimensione del buffer **crescerebbe costantemente** fino a raggiungere la massima dimensione possibile; da quel momento in poi ogni pacchetto in arrivo porta a una condizione di **buffer overflow** e andrebbe perso. Quindi, quali sono i problemi che provoca l'accodamento?

- **Buffer overflow** dei buffer dove vengono salvati i pacchetti ancora da trasmettere e susseguente perdita dei pacchetti in arrivo
- **Ritardo di accodamento** dato dall'attesa che il router trasmetta il pacchetto

1.4 Circuit Switching

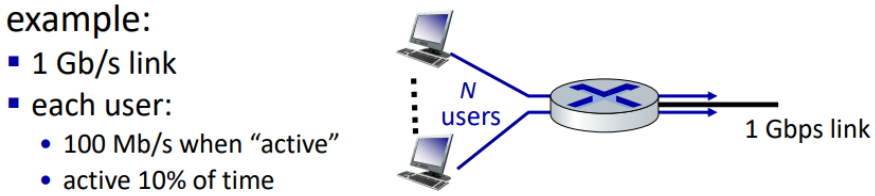
La commutazione di pacchetto non è l'unico tipo di commutazione esistente; anzi essa storicamente è posteriore ad un'altro tipo di commutazione che prende il nome di **commutazione a circuito**



Questo tipo di commutazione segue un principio diverso da quella a commutazione di pacchetto: l'elemento fondamentale di questo tipo di rete è il **circuito** (non esistono i pacchetti in questo tipo di rete). Tra i vari **commutatori di circuito** si possono stabilire un certo numero di circuiti. L'obbiettivo di questo tipo di commutazione è **destinare i circuiti alla comunicazione tra gli utenti della rete**. Le risorse di un circuito vengono **assegnate in maniera esclusiva ad un host** e non possono essere condivise con altri host. Il vantaggio di questo tipo di commutazione è **l'assenza di accodamento** data dal completo assegnamento delle risorse di un circuito ad una sola comunicazione tra sorgente e destinazione. Viene chiamato **circuito end-to-end** l'insieme di tutti i circuiti che vengono stabiliti tra sorgente e destinazione. Uno svantaggio di questo tipo di commutazione è la **necessità di una fase di setup** in cui vengono allocate le risorse dei circuiti per andare a creare il circuito end-to-end.

1.4.1 Packet Switching vs Circuit Switching

In una commutazione di circuito, le risorse vengono allocate solamente alla comunicazione tra sorgente e destinazione e non esistono problemi di accodamento. Tuttavia, l'utilizzo delle risorse di rete in questo tipo di comunicazione risulta **subottimale** rispetto ad una rete a commutazione di pacchetto. Facciamo un esempio:



Domanda: quanti utenti, in queste condizioni, possono rispettivamente accomodare una rete a commutazione di circuito e una rete a commutazione di pacchetto?

- **Circuit-switching**: 10 utenti
- **Packet-switching**: con 35 utenti, la probabilità che più di 10 utenti siano attivi allo stesso tempo è minore di 0.0004

Quindi una rete a commutazione di pacchetto è sempre la scelta migliore?

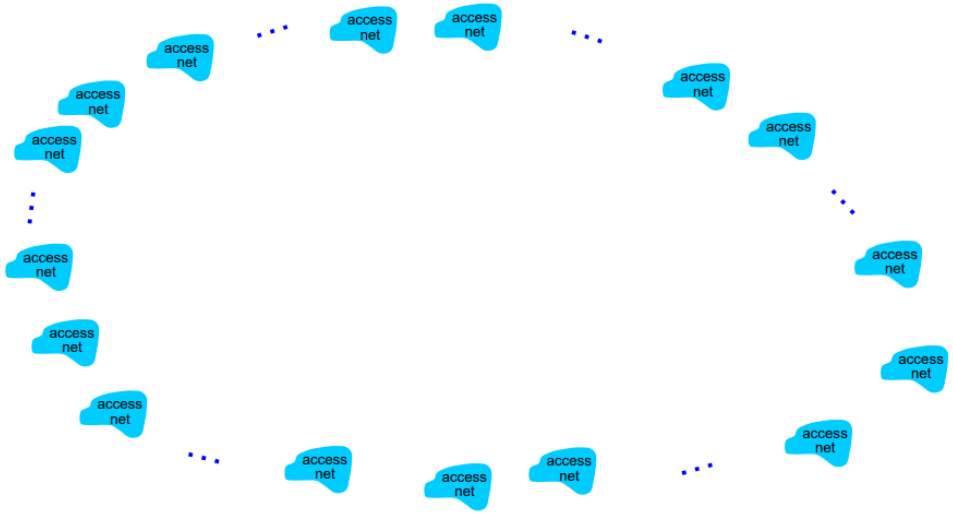
- Essa è ottima per un traffico di tipo **"bursty"** (a raffica), cioè in situazioni dove a volte ci sono dati da inviare, a volte no
- Permette la condivisione delle risorse di rete
- Non richiede setup

Tuttavia, se si creano condizioni sfavorevoli, si vanno a creare delle **situazione di congestione eccessiva della rete**, andando a creare situazioni di ritardo nella trasmissione dei pacchetti e di buffer overflow. Si rendono quindi necessari **protocolli di trasmissione affidabili** e meccanismi di **controllo della congestione**. È però possibile, visti i vantaggi delle reti a commutazione di circuito, fornire lo stesso tipo di garanzia del servizio anche nelle reti a commutazione di pacchetto? La risposta è si;

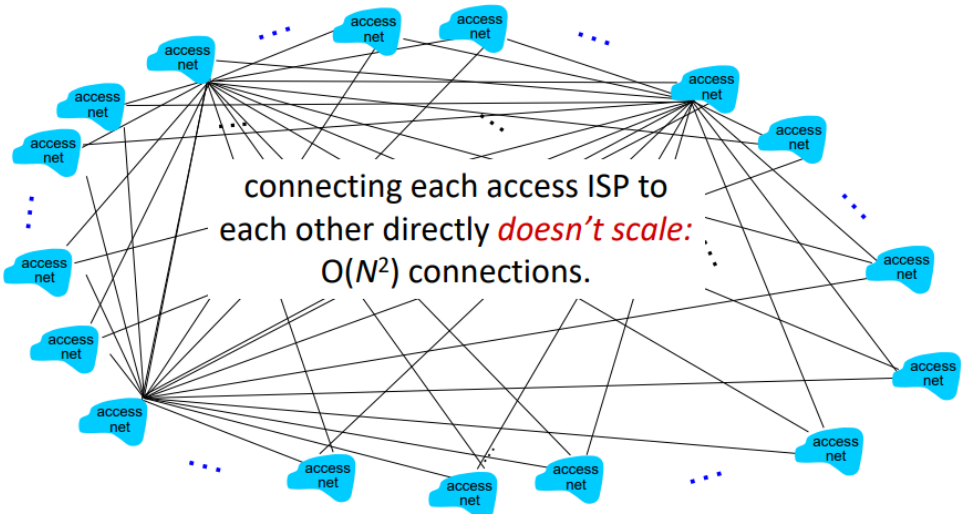
esistono delle tecniche che permettono di **emulare la commutazione di circuito sulle reti a commutazione di pacchetto**, tuttavia sono casi parecchio difficili da gestire.

1.5 Struttura di internet nel dettaglio

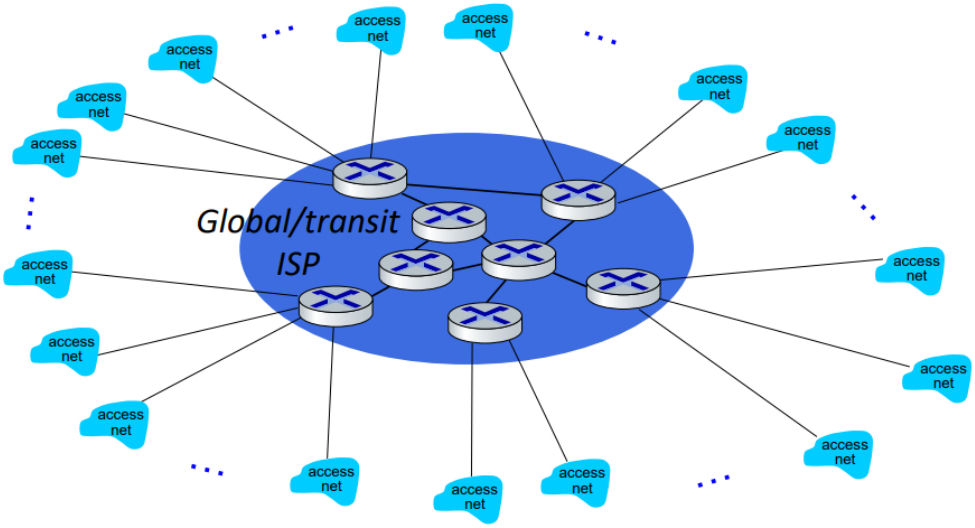
Gli host sono connessi a internet tramite le **reti di accesso** fornite da ISPs. Le reti di accesso devono per forza essere **interconnesse** per garantire che possa avvenire la comunicazione tra qualsiasi due host della rete (internet è una rete **connessa**, cioè ogni nodo è raggiungibile da tutti gli altri nodi). Il risultato dell'evoluzione di internet è una **struttura gerarchica**; questa evoluzione, tuttavia, non è stata guidata da **necessità di tipo tecnico** ma di tipo **economico e politico**. Vediamo quindi la struttura di internet passo passo: al mondo abbiamo **milioni** di reti di accesso



L'idea più semplice per interconnetterle è **connettere ogni ISP agli altri in maniera diretta**, tuttavia...



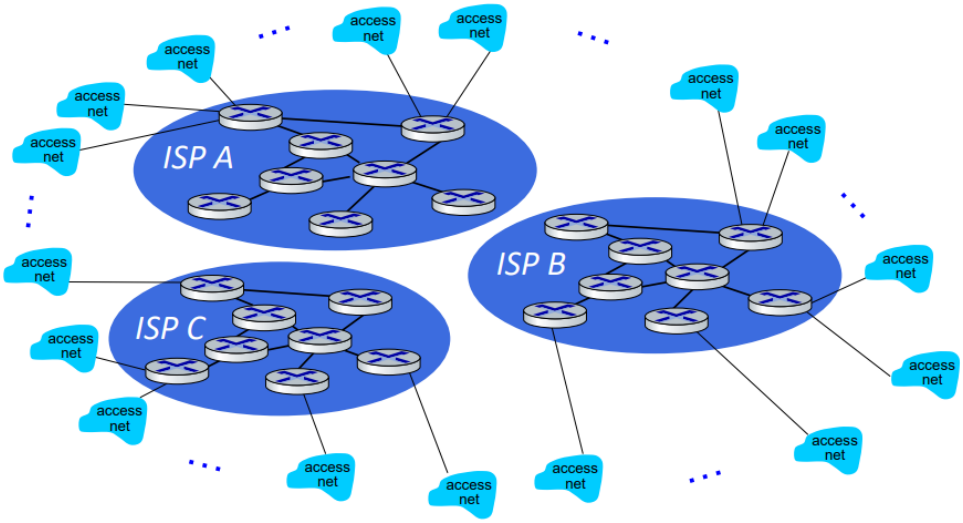
Una soluzione alternativa è quella di avere un **ISP globale** (o di transito) con una rete geograficamente estesa in tutto il mondo, i cui router sono **interconnessi** e che fornisce connettività alle reti di accesso:



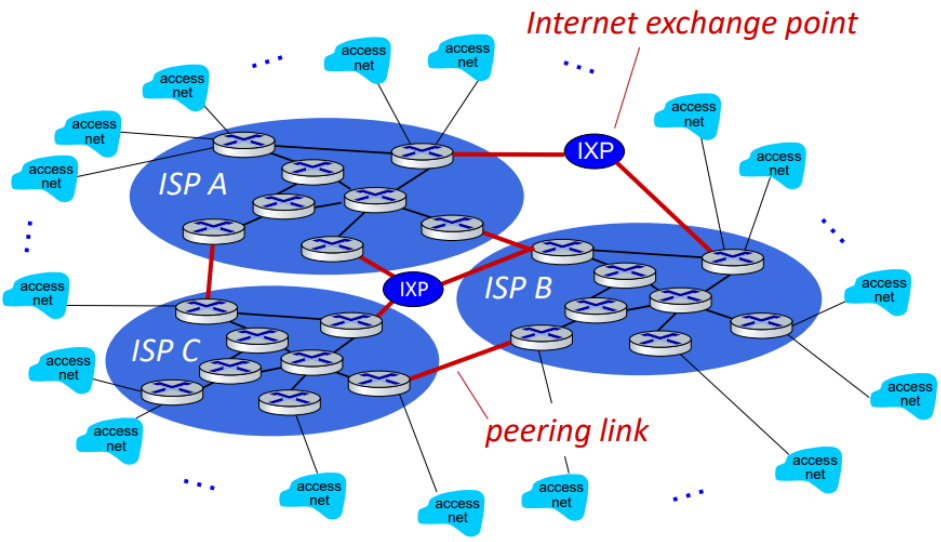
Questo approccio tuttavia ha dei problemi:

- Richiede una rete estesa **su tutto il globo**, in modo che possa servire ogni rete di accesso
- Richiede un accordo economico tra gli ISP locali e quello globale
- Perché ci si dovrebbe limitare ad un solo ISP globale? In effetti, ci potrebbe essere concorrenza fra essi

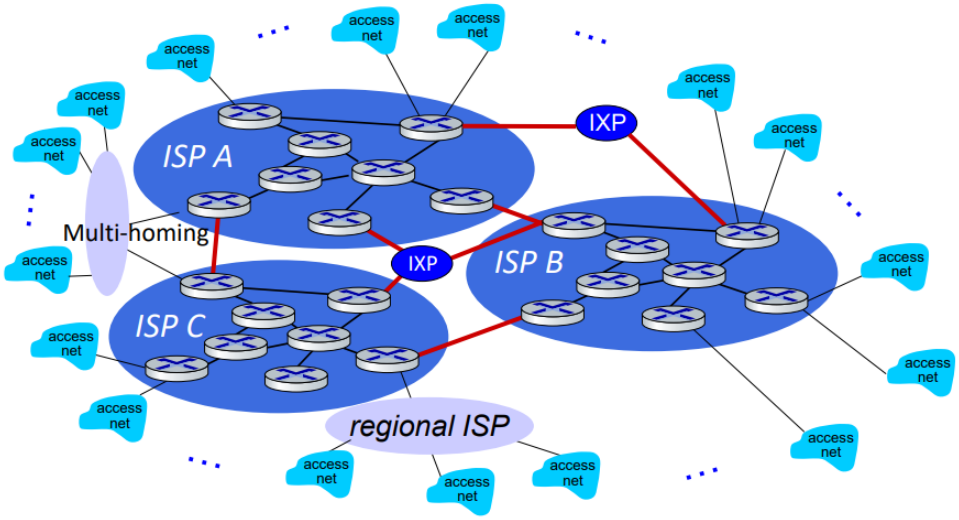
In virtù dell'ultimo punto sopra, si è andata a creare una situazione di questo tipo:



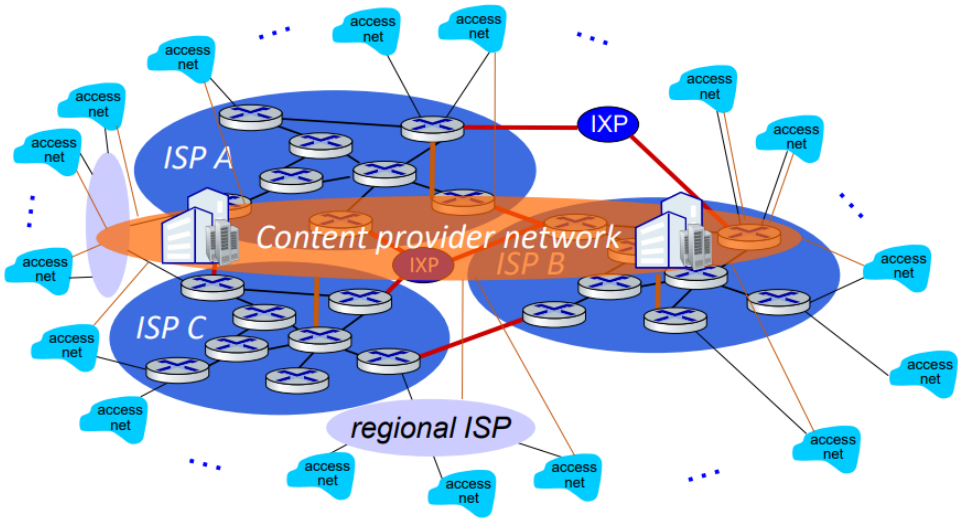
si sono andati quindi a creare degli ISPs **geograficamente estesi** che coprono determinate aree del globo. Questo genere di ISPs prendono il nome di **Tier 1 ISPs**. A questo punto, gli ISPs globali devono creare dei **peering links** (chiamati così perché **non sono il frutto di un accordo economico ma di un accordo tra pari**) per creare un'interconnessione tra di loro:



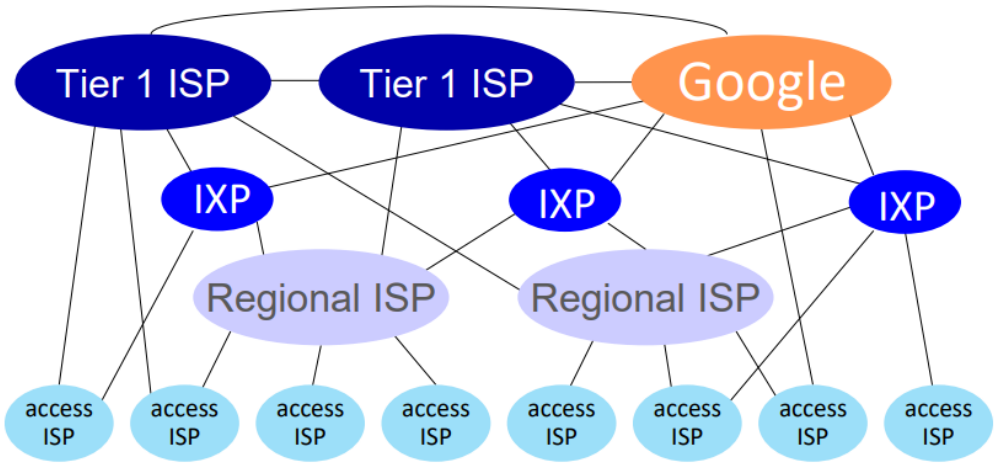
Tuttavia, c'è anche un'altra possibilità, cioè quella di avere degli **Internet Exchange Points** (IXPs), i quali sono dei nodi a cui gli ISPs globali si connettono e che garantiscono la connessione tra di essi. Tuttavia, tipicamente le reti di acceso non si **interfacciano direttamente con gli ISPs globali** ma passano tramite **ISPs regionali**, che hanno una diffusione più capillare sul territorio. Altro tipo modo in cui le reti di accesso accedono agli ISPs globali è tramite **multi-homing**, cioè una rete di accesso potrebbe avere **più collegamenti verso un ISPs globale o regionale**; questo approccio garantisce **resilienza**



Infine, abbiamo le **reti dei content provider**, che hanno lo scopo di **fornire contenuto agli utenti**. Per farlo, molto spesso, esse **bypassano gli ISPs globali** e si interfacciano direttamente con le reti di accesso.



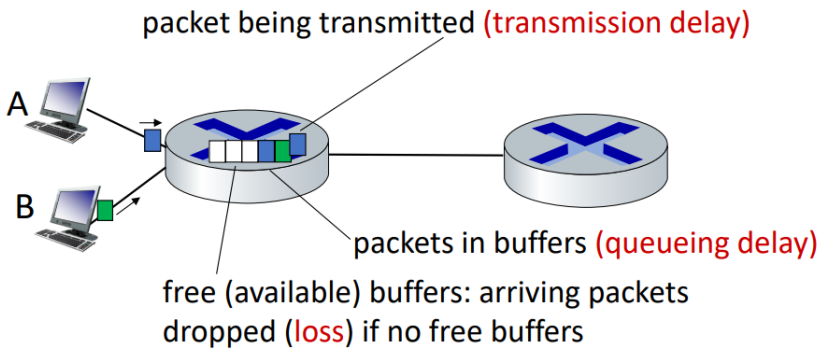
Nulla però vieta alle reti dei content provider di accedere anche alle reti degli ISPs globali. Diamo una versione più generale della gerarchia:



Tuttavia, le connessioni possibili **possono anche non limitarsi a quelle mostrare in figura**.

1.6 Metriche di performance nelle reti

Il ritardo e la perdita sono due delle metriche fondamentali per capire la prestazione della rete. Abbiamo già parlato di **ritardo di accodamento** e di **ritardo di trasmissione**, tuttavia essi sono solo due dei tipi di ritardo che si possono verificare.



Tuttavia ce ne sono delle altre, vediamole tutte:

- **Ritardo di elaborazione** (d_{proc}): quando un pacchetto arriva ad un nodo, devono essere effettuate delle operazioni fondamentali, per esempio **l'analisi della tabella di inoltro per capire su quale collegamento trasmettere il pacchetto** (lookup). Altra operazione è il **controllo dei bit di errore**: a volte, per colpa di distorsioni o "rumore" sul canale, certi bit di un pacchetto possono essere cambiati (da 0 a 1 e viceversa); il pacchetto quindi risulta malformato. Esistono meccanismi che permettono di **individuare se c'è stato un errore** (e anche di ricostruire il pacchetto originale). Questi metodi sono detti di **controllo e correzione dell'errore** e richiedono del tempo per essere eseguiti, tendenzialmente nell'ordine dei **microsecondi** (10^-6s).

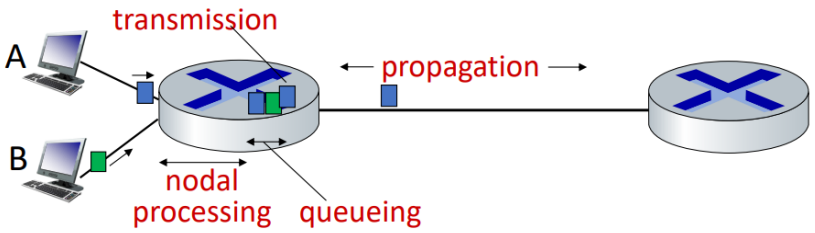
- **Ritardo di accodamento** (d_{queue}): è il tempo che il pacchetto attende per essere trasmesso. Dipende dal livello di congestione della rete. È tipicamente nell'ordine dei millisecondi

- **Ritardo di trasmissione** (d_{trans}): Se abbiamo un pacchetto di lunghezza L che vogliamo trasferire su un collegamento con tasso di trasmissione R, allora il ritardo di trasmissione è dato da:

$$d_{trans} = \frac{L}{R}$$

- **Ritardo di propagazione** (d_{prop}): Il ritardo di propagazione è il tempo che necessita un pacchetto per **propagarsi da un capo all'altro del collegamento**. Questo ritardo dipende dalla lunghezza del collegamento fisico d (espressa in **metri** (m)) e dalla **velocità di propagazione** s, che è nell'ordine di grandezza della **velocità della luce**: $\sim 2 \cdot 10^8\ m/sec$, e si calcola nel seguente modo:

$$d_{prop} = \frac{d}{s}$$



Il **ritardo totale del nodo** (ritardo **end-to-end**) d_{nodal} è calcolato come la somma di tutti ritardi visti sopra:

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

esso è uno delle **metriche più importanti** e mi permette di valutare le prestazioni della rete. Per calcolare il ritardo ent-to-end bisognerebbe anche tener conto dei **ritardi introdotti dai sistemi periferici**, che tuttavia non considereremo per questi appunti.

1.6.1 Ritardo di accodamento e intensità di traffico

Il ritardo di accodamento ha una caratteristica particolare: mentre tutti gli altri ritardi sono **analoghi** se considero lo stesso pacchetto trasmesso in due momenti diversi, **per il ritardo di accodamento non è così**. Il ritardo di accodamento dipende dallo stato della rete durante la trasmissione. Questo ritardo di solito viene studiato tramite **metodi statistici**: in particolare, il ritardo di accodamento dipende da un valore che viene chiamato **intensità di traffico** ("traffic intensity"):

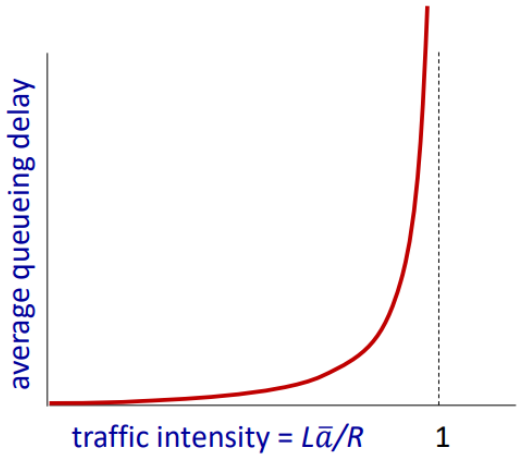
$$\frac{L \cdot \overline{a}}{R} = \frac{\text{tasso di arrivo dei pacchetti (bit/s)}}{\text{tasso di uscita dei pacchetti (bit/s)}}$$

Dove:

- $\overline{a}$ è il **tasso di arrivo medio dei pacchetti** (pacchetti/s)
- L è la **lunghezza** dei pacchetti (bit/packet)
- R è la **larghezza di banda** del collegamento (bit/s)

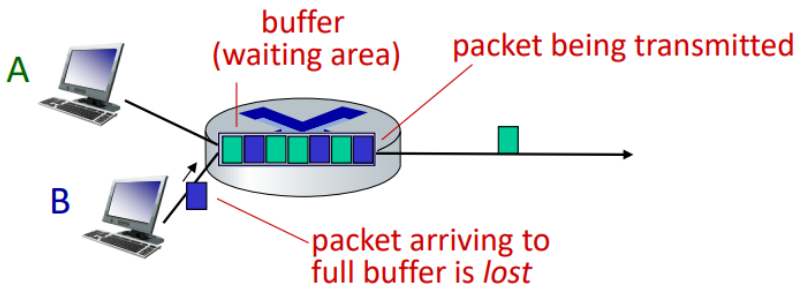
L'intensità di traffico è **adimensionale** ed è una quantità non negativa:

- Se l'intensità di traffico è **vicina a 0**, allora il ritardo di accodamento medio sarà piccolo
- Se l'intensità di traffico è **vicina a 1**, allora il ritardo di accodamento medio è grande
- Se l'intensità di traffico è **maggiore di 1**, allora il tasso di arrivo dei pacchetti **è troppo alto rispetto alla capacità della rete di smaltirli**, quindi il tempo di accodamento medio è **potenzialmente infinito**



1.6.2 Perdita di pacchetti

Un'altra metrica di performance fondamentale è la **perdita di pacchetti**; infatti, idealmente essa dovrebbe essere **nulla**. La perdita di pacchetti si verifica quando si ha un buffer overflow sul buffer di accodamento di un nodo. La perdita di pacchetti può essere mitigata tramite **meccanismi di ritrasmissione dei pacchetti** da parte del nodo sorgente (tuttavia non è obbligatorio che avvenga).



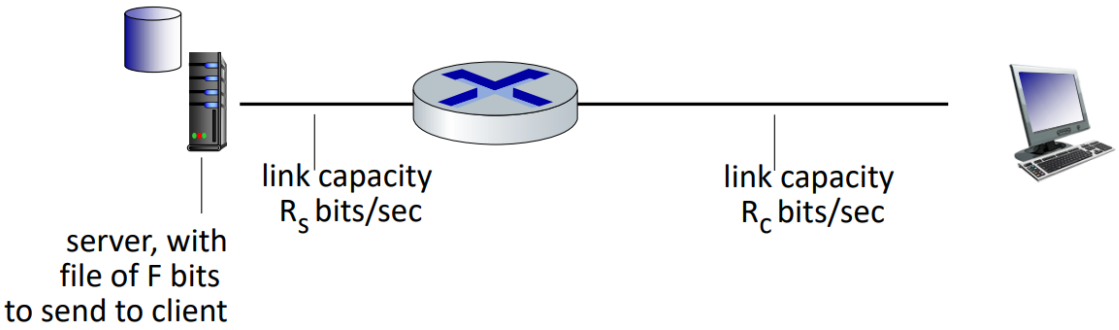
Oltre alla perdita per accodamento, **i pacchetti possono essere scartati dai commutatori** se ci sono stati troppi errori durante la trasmissione oppure se non è previsto un meccanismo di correzione degli errori.

1.6.3 Throughput

Il **throughput** è il **tasso a cui i bits vengono inviati dal mittente al destinatario**; si calcola in bit/s e viene a volte chiamato **throughput end-to-end**. Come si fa a calcolare il throughput?

- **Throughput istantaneo**: tasso di invio **ad un certo punto nel tempo**
- **Throughput medio**: tasso di invio medio **su un lungo periodo di tempo**; si può ottenere facendo la media di tutti i valori istantanei o nel seguente modo: supponiamo di voler trasferire un file F dal punto A al punto B; allora il throughput sarà la dimensione del file diviso il tempo che ha impiegato il file ad essere trasferito

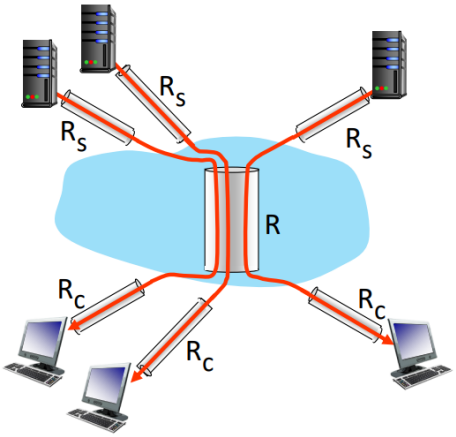
Il throughput è una metrica di prestazione "quanto bene" la rete sta andando nella comunicazione tra un mittente ed un destinatario.



Si possono verificare vari casi:

- Se $R_s < R_c$ allora il throughput medio sarà R_s. In questo caso, **non si causerà mai accodamento nel commutatore**
- Se $R_s > R_c$ allora il throughput medio sarà R_c. In questo caso, potrebbe essere possibile che i **pacchetti si accodano nel commutatore**

In ogni caso, il throughput è vincolato dal collegamento con capacità minore, il quale prende il nome di **bottleneck link**. Vediamo questo ulteriore scenario: supponiamo di avere 10 client e 10 server e che essi, per comunicare, passino attraverso un collegamento all'interno della network core con larghezza di banda R:



In questo caso, il throughput end-to-end per connessione risulta il minimo tra R_c, R_s, e $R/10$, cioè $min\{R_C, R_s, R/10\}$. Nella pratica, tuttavia, la rete di core ha larghezze di banda **molto maggiori rispetto ai collegamenti degli hosts**, quindi $R/10$, il più delle volte, sarà comunque molto maggiore di R_s e R_c. Ciò significa che, molto spesso, il bottleneck è dato dai collegamenti di accesso.

1.7 Strato protocollare e modelli di servizio

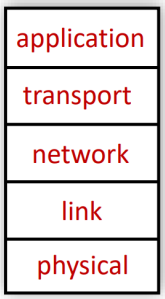
Le reti sono dei sistemi molto complessi con diverse componenti che interagiscono fra loro

- Hosts
- Routers
- Collegamenti di diverso tipo e scopo
- Applicazioni
- Protocolli
- Hardware e Software

C'è allora un modo intelligente per **strutturare e discutere le reti**? Un modo è una **struttura a strati**, in cui ogni servizio viene implementato tramite **azioni interne allo strato** e ogni servizio fa affidamento ai servizi **appartenenti al livello sottostante per implementare le proprie operazioni e garantirne il funzionamento**. Inoltre, i vari strati sono **indipendenti l'uno dall'altro**, nel senso che, in caso di modifiche a come vengono implementati i servizi in uno strato, gli altri strati **non vengono influenzati dalla modifica**. L'approccio a strati quindi permette di approcciarsi alla modellazione e alla discussione di sistemi complessi:

- La sua **struttura esplicita** permette l'identificazione e le **relazioni fra le varia componenti del sistema**. Crea inoltre un **modello di riferimento a strati** per la discussione
- Permette la **modularizzazione** del sistema e quindi rende più semplice l'aggiornamento e la manutenzione di questo
  - I cambiamenti nell'implementazione dei servizi di uno strato sono **trasparenti al resto del sistema** e quindi non lo influenzano

Ciò che è stato definito per internet prende il nome di **pila (o stack) protocollare di internet** e prevede l'esistenza di 5 diversi strati, numerati dal basso verso l'alto:

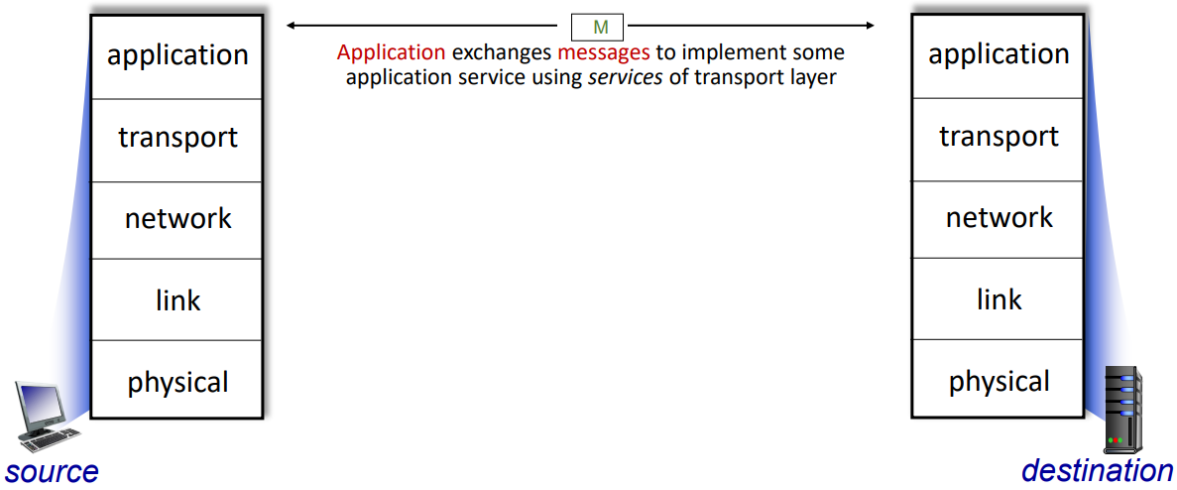


Vediamoli nel dettaglio:

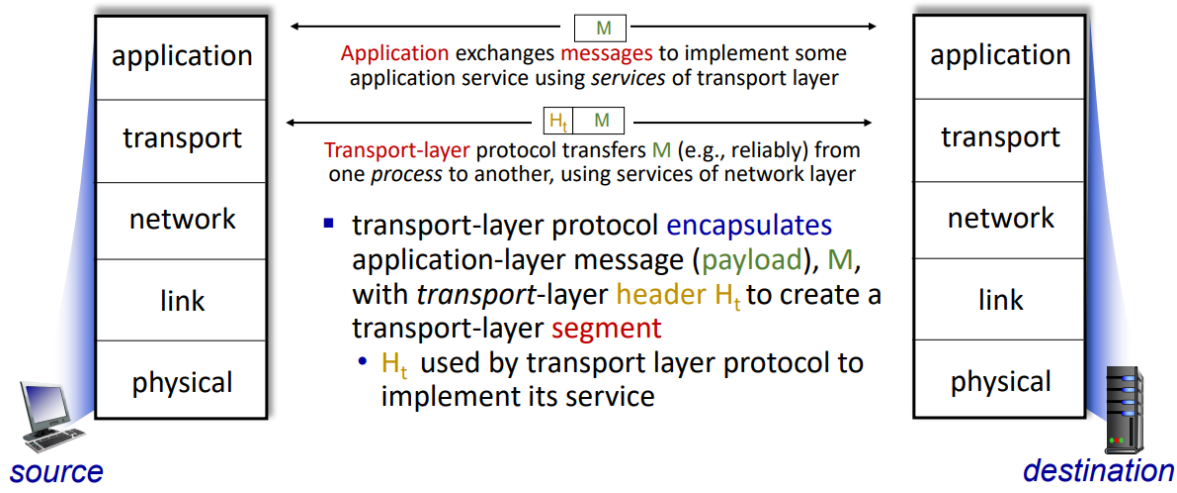
- **LIVELLO 5: APPLICATIVO**: Tutti i protocolli volti al supporto alle applicazioni di rete (HTTP, SMTP, DNS ecc..)
- **LIVELLO 4: TRASPORTO**: Ha il compito di effettuare un trasferimento di dati tra macchine che si trovano su reti differenti. Esempio di protocolli a questo livello sono TCP e UDP
- **LIVELLO 3: RETE**: Offre il servizio fondamentale di **instradamento** tra sorgente e destinazione. Protocollo fondamentale a questo livello è **Internet Protocol** (IP)
- **LIVELLO 2: COLLEGAMENTO**: Ha il compito di trasferire i dati tra **due entità di rete adiacenti**, cioè su un collegamento. Protocolli a questo livello sono, per esempio, Ethernet e 802.11(WiFi).
- **LIVELLO 1: FISICO**: Trasporto fisico dei bit su un cavo (o canale radio)

1.7.1 Servizi, Stratificazione e Incapsulamento

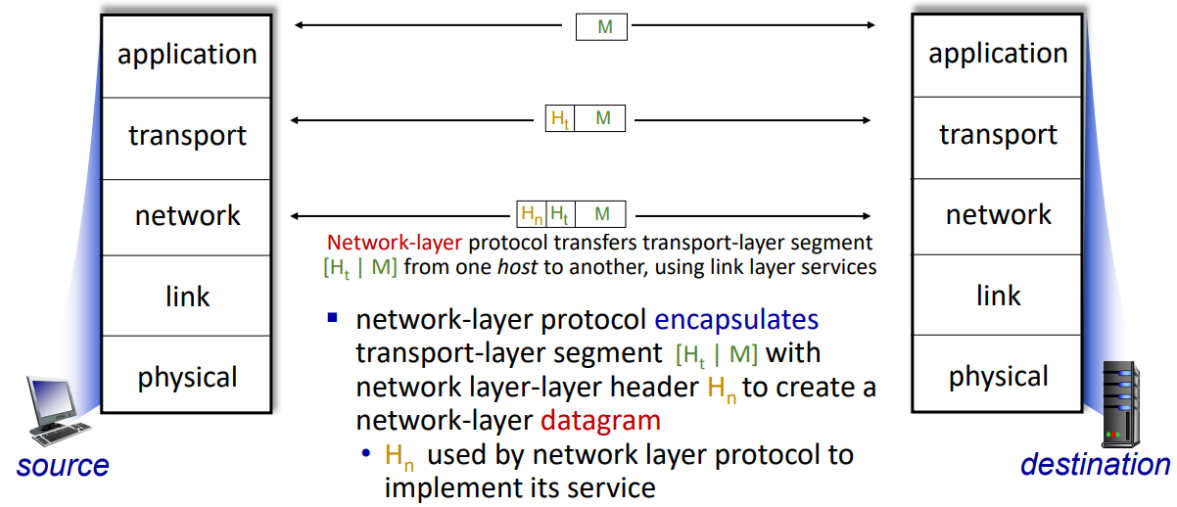
Come si può andare ad implementare i vari protocolli che garantiscono il funzionamento dei servizi ad ogni livello? Un concetto fondamentale da introdurre a questo scopo è quello dell'**incapsulamento**: le applicazioni si devono scambiare messaggi per realizzare i propri servizi, e per farlo devono basarsi sui servizi offerti dal **livello di trasporto**



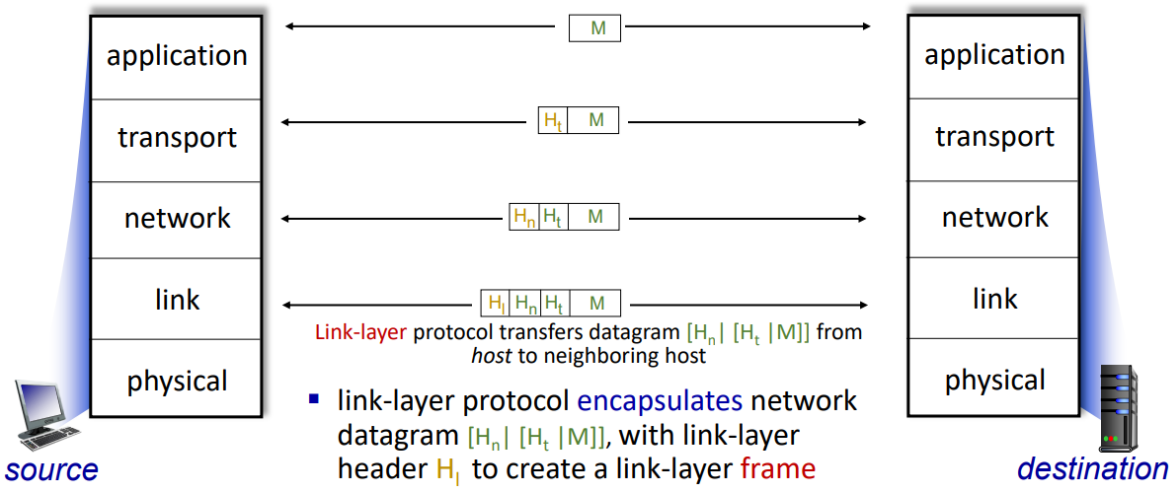
Il messaggio viene quindi passato al livello di trasporto, il quale **aggiunge un'intestazione (operazione di incapsulamento)** che va ad implementare il servizio a livello di trasporto:



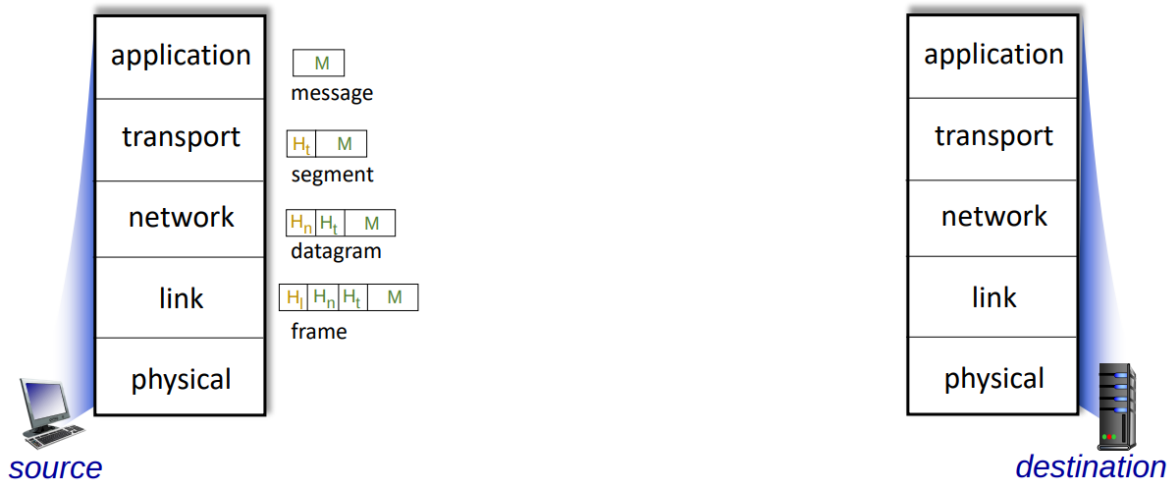
A sua volta, il livello di trasporto usa **i servizi del livello di rete**, quindi passerò il nuovo **segmento** al livello di rete, il quale aggiungerà un'ulteriore intestazione:



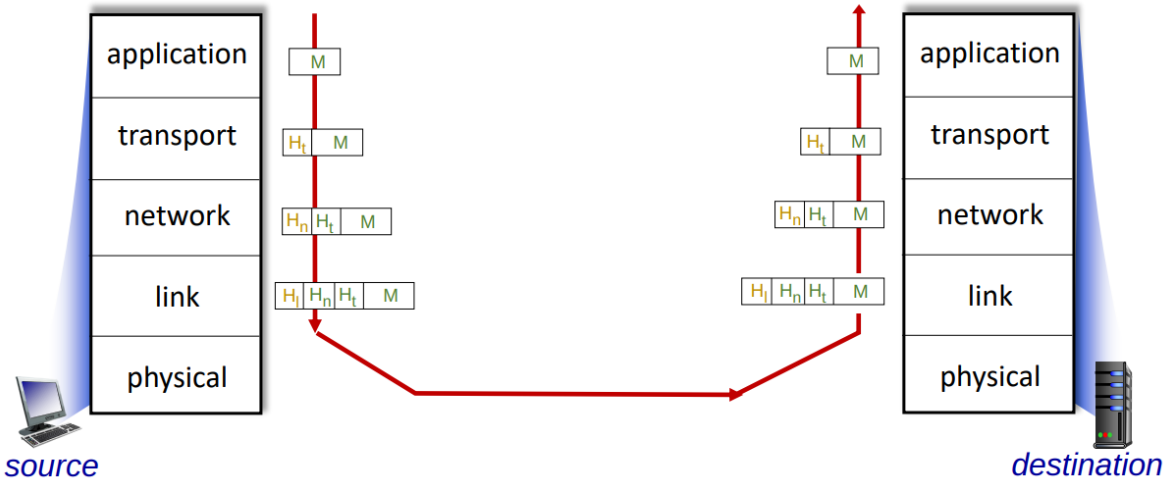
Da notare che a questo livello **il payload contiene anche l'header del trasporto**, e sarà così anche per le prossime operazioni. Lo strato di rete si basa sui **servizi offerti dal livello di collegamento** per implementare i propri, quindi passa il nuovo **datagramma** ottenuto al livello di collegamento:



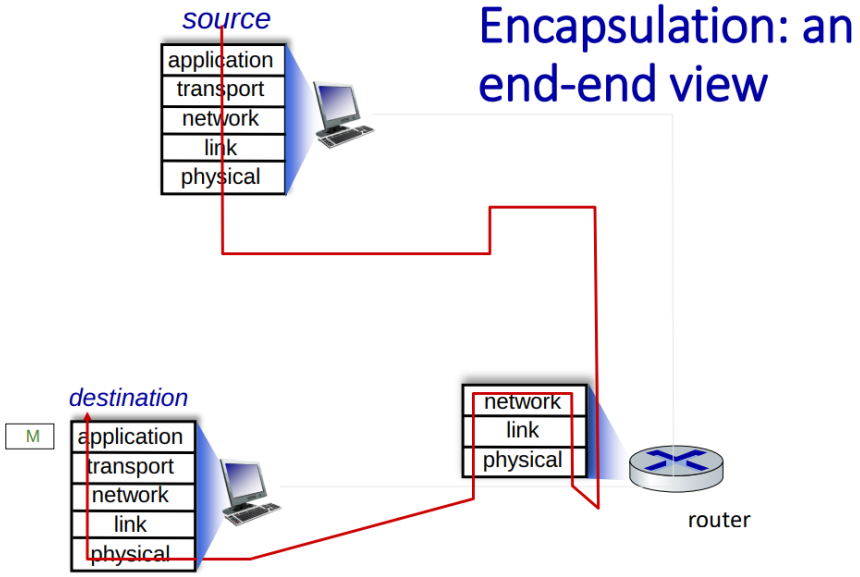
Infine, il pacchetto verrà passato al livello fisico e quindi trasmesso al destinatario. Come abbiamo già accennato, i pacchetti hanno nome diverso in strati diversi:



Man mano che si scende nella pila protocollare, vengono aggiunti nuovi bit di intestazione, i quali servono per **far funzionare i servizi di rete**. L'insieme di tutti i bit aggiunti da ogni strato prende il nome di **overhead** del pacchetto. Poiché il throughput viene misurato **a livello applicativo**, non si avrà mai, a livello pratico, **una misura di throughput massima** poiché non vengono contati i bit di overhead. Dopo l'operazione di **incapsulamento** del messaggio e di trasmissione su cavo fisico, avviene sul sistema ricevente un'operazione di **decapsulamento** del messaggio:



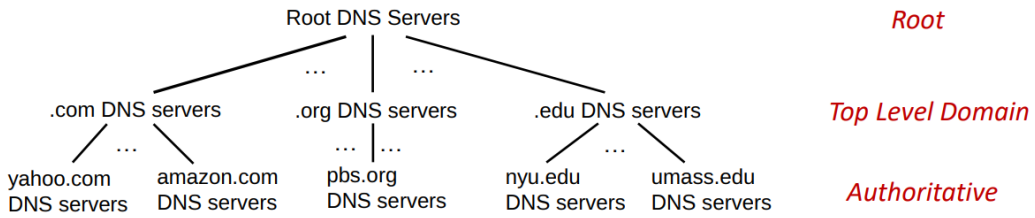
I sistemi periferici sono sempre in grado di interpretare tutti gli strati della pila protocollare; questo però non è vero per i **commutatori**, che "parlano" fino al **livello 2** se sono degli **switch** e fino al **livello 3** se sono dei **router**:



2 Strato applicativo: Domain Name System

Lo strato applicativo ha la funzione di interfacciare e fornire servizi di connettività per i **processi delle applicazioni**. Il numero di protocolli presenti in questo strato è altissimo; in questi appunti, tuttavia, ci concentreremo sul protocollo **DNS**, il quale è una funzionalità chiave di internet, in particolare se si pensa ad esso come **fornitore di servizi web**. Quando inseriamo il **nome** di un sito nel browser, ciò che è avviene è lo scatenamento di un meccanismo che permette al computer di ottenere **l'indirizzo IP** (quindi l'indirizzo a livello di rete) associato al server che deve fornire la pagina web richiesta. Per ottenere questo indirizzo IP, non si può pensare di conoscere ogni indirizzo esistente, quindi è necessario un meccanismo che ci permette di reperirlo: il **Domain Name System** (DNS). Il DNS permette di effettuare la traduzione dall'**host**

name (i nomi "human readable" dei siti) all'indirizzo IP del server associato che dovrà fornire al richiedente il contenuto. Come funziona quindi la traduzione? Essa sfrutta la **gerarchia** di un numero molto elevato di **server DNS** (che prendono anche il nome di **name servers**), i quali andranno a **risolvere** (tradurre) gli host name in indirizzi IP. Poiché il DNS è un protocollo a livello applicativo, la complessità di questa operazione si trova **interamente ai confini della rete**. Il DNS funziona nel seguente modo:



Client wants IP address for www.amazon.com; 1st approximation:

- client contacts root DNS server to find .com DNS server
- client contacts .com DNS server to get amazon.com DNS server
- client contacts amazon.com DNS server to get IP address for www.amazon.com

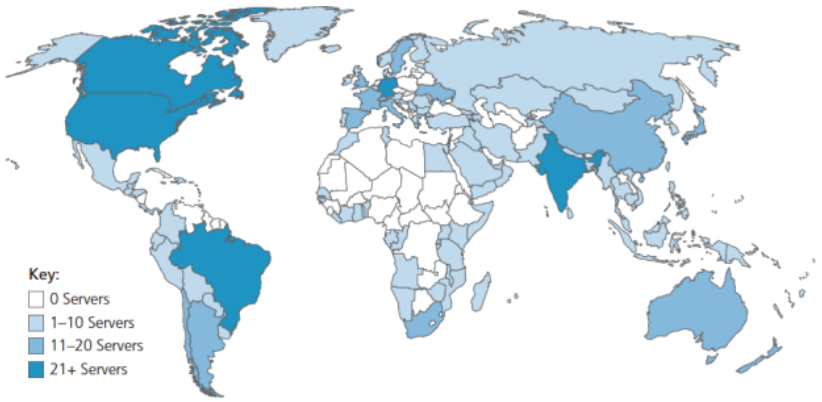
abbiamo quindi un sistema distribuito che si articola su **3 livelli**:

- **Root DNS servers**: sono la radice dell'albero gerarchico
- **Top Level Domain (TLD) DNS Servers**: sono responsabili della risoluzione dei **Top Level Domains**(TLDs); come per esempio .com, .net, .it, ...
- **Authoritative DNS Servers**: Responsabili di risolvere gli indirizzi per i vari domini autoritativi (es. amazon.com, google.com ecc...)

Questi 3 livelli collaborano tra loro per risolvere il nome richiesto.

2.1 Root DNS Servers

I root DNS server stanno alla radice del sistema DNS e senza la loro esistenza **è fondamentale per l'esistenza del web**. In tutto, esistono 13 root name servers "logici", gestiti da **12 provider differenti**; a loro volta ognuno di questi root dns server è replicato più volte nel globo. È fondamentale avere repliche dei root name server geograficamente sparse che possono essere acceduti dai vari luogo del mondo.



2.2 Top Level DNS servers e Authoritative DNS servers

Ogni TLD DNS server è responsabile per la risoluzione dei TLD, inclusi quelli nazionali (.it, .de. .pl, ...). I DNS server autoritativi invece è importate specificare che **possono essere mantenuti dall'organizzazione che detiene il dominio** oppure **possono essere mantenuti da un service provider**. Il loro compito è di risolvere le richieste relative a tutte le pagine appartenenti al loro dominio.

2.3 Local DNS name servers

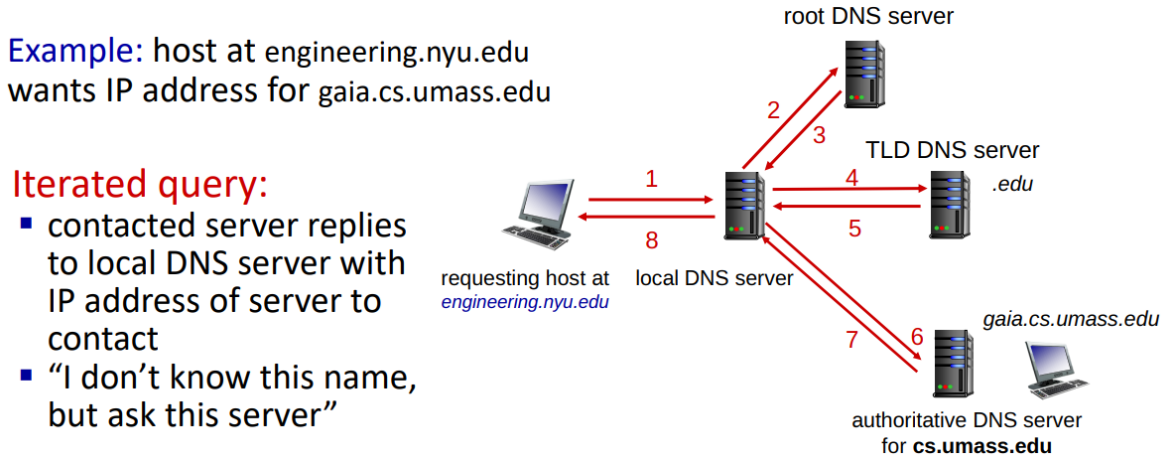
Quando vi è la necessità di contattare il DNS per risolvere un dominio,la prima cosa che viene fatta è contattare il **DNS server locale**, a cui viene demandata di fatto questa risoluzione. Il DNS server locale ha un ruolo fondamentale: esistono DNS server locali forniti da ISP, che di solito sono l'opzione di default, tuttavia un utente può benissimo cambiare il proprio DNS server locale ad uno fornito da un'organizzazione terza. Il DNS server locale tendenzialmente fornisce due servizi:

- Mantiene una cache delle associazioni nome-indirizzo IP. Questa cache viene costruita a partire dalle varie richieste dei sistemi terminali e di solito le associazioni vengono mantenute per 2 giorni. Il DNS server locale quindi risponderà direttamente alle richieste del client, senza quindi interpellare la gerarchia del DNS, se il nome richiesto è presente nella sua cache.
- In caso il nome richiesto non sia presente in cache, il DNS server locale procederà ad inoltrare la richiesta alla gerarchia del DNS; una volta ottenuta risposta, procederà a comunicarla al client

Ogni richiesta al DNS prende il nome di **query**.

2.4 DNS name resolution: Iterated query

Ci sono due modi per effettuare query al DNS. Il modo che prende il nome di **query iterativa** è quello che viene usato più spesso. In questo metodo, il local DNS server ha **il maggior carico** dal punto di vista delle richieste da servire.



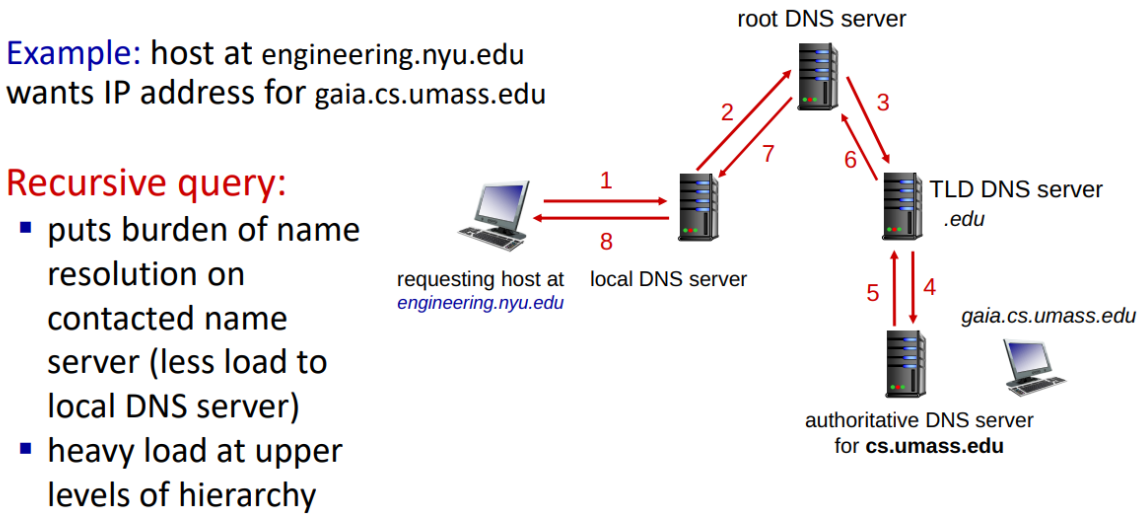
Analizziamo nel dettaglio ogni passaggio:

1. L'host chiede al DNS server locale di risolvere un certo nome in un indirizzo IP
2. Il DNS server locale chiede al DNS server root di risolvere il nome
3. Il DNS server root indica al server DNS server locale quale TLD DNS server può risolvere la sua richiesta
4. Il DNS server locale chiede al TLD DNS server indicato di risolvere il nome
5. Il TLD DNS server indica al server DNS locale quale DNS server autoritativo può risolvere la sua richiesta
6. Il DNS server locale chiede al server DNS autoritativo di risolvere il nome
7. Il DNS server autoritativo restituisce l'indirizzo IP associato al nome indicato
8. Il DNS server locale restituisce l'indirizzo IP al client

Ogni passaggio di questo processo è inoltre **trasparente all'utente**.

2.5 DNS name resolution: recursive query

In una query ricorsiva, la risoluzione viene effettuata, in modo gerarchico, dal DNS server di livello superiore. Il peso delle richieste quindi grava sulla **struttura gerarchica del DNS**.



Vediamo ogni passo in dettaglio:

1. Il client chiede al DNS server locale di risolvere il nome
2. Il DNS server locale chiede al DNS server di root di risolvere il nome
3. Il DNS server di root chiede al TLD DNS Server corrispondente di risolvere il nome

4. Il TLD DNS server chiede al DNS server autoritativo corrispondente di risolvere il nome
5. Il DNS server autoritativo restituisce l'IP richiesto al TLD DNS server
6. Il TLD DNS server restituisce la risposta ottenuta al root DNS server
7. Il root DNS server restituisce la risposta ottenuta al DNS server locale
8. Il DNS server locale restituisce la risposta ottenuta al client

Solitamente, questo meccanismo non si utilizza poiché tendenzialmente i TLD DNS server, per loro natura, si trovano a gestire un alta quantità di traffico, quindi un meccanismo di risposta ricorsivo porrebbe ulteriore peso su di essi. Si tende quindi a cercare di mantenere il peso sui DNS server locali, i quali tendenzialmente gestiscono una quantità di traffico molto minore.

3 Livello di trasporto

I servizi del livello di trasporto offrono:

- **comunicazione logica** tra processi su host diversi, cioè processi su macchine diversi possono comunicare come se fossero direttamente connessi fra loro
- Le azioni effettuate dai protocolli di trasporto nei sistemi periferici sono:
  - **Mittente**: il livello di trasporto deve prendere i messaggi applicativi, eventualmente **dividerli in parti più piccole**, incapsularli in quelli che prendono il nome di **segmenti**
  - **Destinatario**: il livello di trasporto **riassembla i segmenti** per ricostruire il messaggio originale e poi passa il messaggio ricostruito al livello applicativo

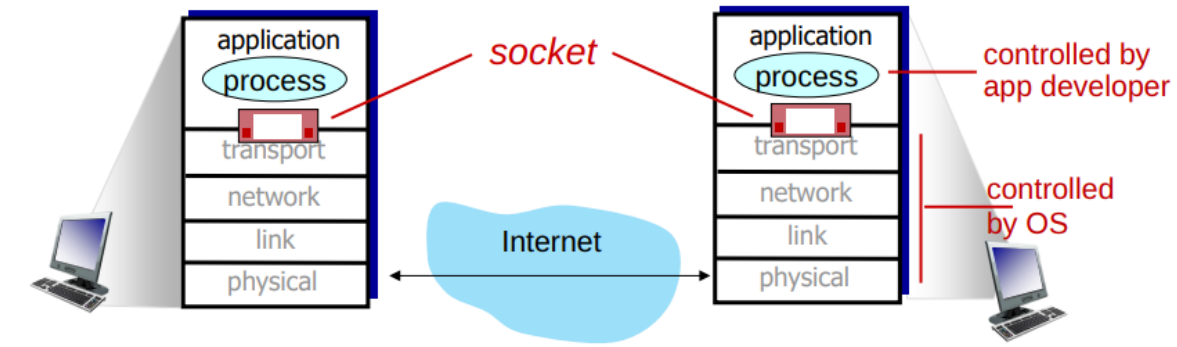
Come si differenzia il servizio di comunicazione tra processi offerto dal livello di trasporto dal servizio offerto dal livello di rete?

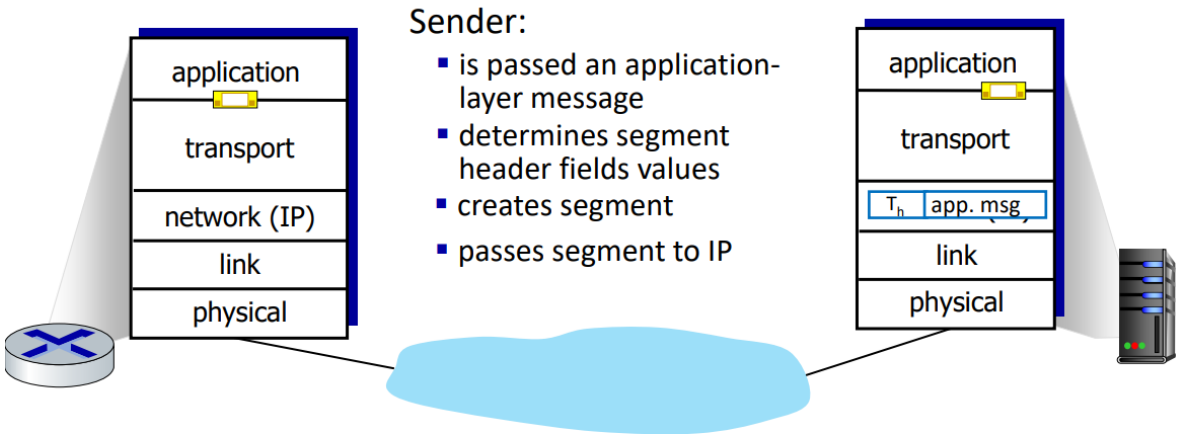
- A **livello di rete**, abbiamo una comunicazione logica tra **hosts**
- A **livello di trasporto**, estende la comunicazione logica offerta dal livello di rete, migliorandola a una comunicazione tra **processi su host differenti**

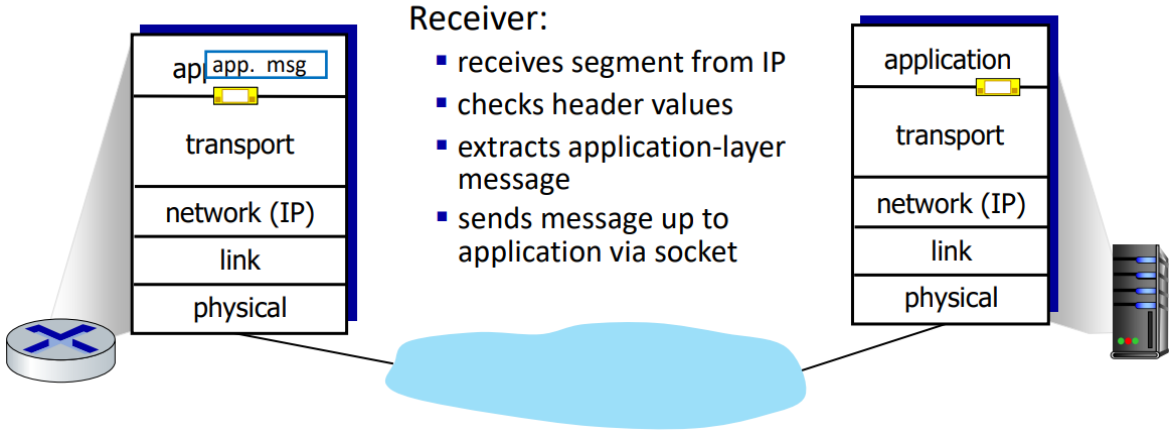
3.1 Sockets

Una **socket** è un "varco" che permette ai processi a livello applicativo di comunicare con il livello di trasporto e di inviare i messaggi verso destinatari usando lo stack protocollare di internet. Quando quindi un processo deve mandare un messaggio in rete, **lo fa passare attraverso una socket**; vengono poi sfruttati i servizi offerti dal livello di trasporto per fare in modo che il messaggio sia recapitato **ad un'altra socket dal lato del destinatario** (ci sono quindi almeno sempre due socket in una comunicazione tra processi). Dal punto di vista formale, una socket è **un'interfaccia** che permette di interfacciare il livello applicativo con il livello di trasporto. La cosa particolare

della socket è che, da un lato la parte della socket che si **interfaccia con i processi applicativi** può essere controllata da chi sviluppa l'applicazione (può aprirla, chiuderla, ...), dall'altro la parte che si interfaccia con il livello di trasporto è **controllata dal sistema operativo**, il quale implementa lo stack protocollare.







3.2 I principali protocolli di trasporto

Vi sono due principali protocolli di trasporto:

- **Transmission Control Protocol (TCP)**: Fornisce un servizio di **consegna dei dati affidabile e in ordine**. Offre meccanismi di **controllo della congestione, controllo di flusso** e di **inizializzazione della connessione** (TCP

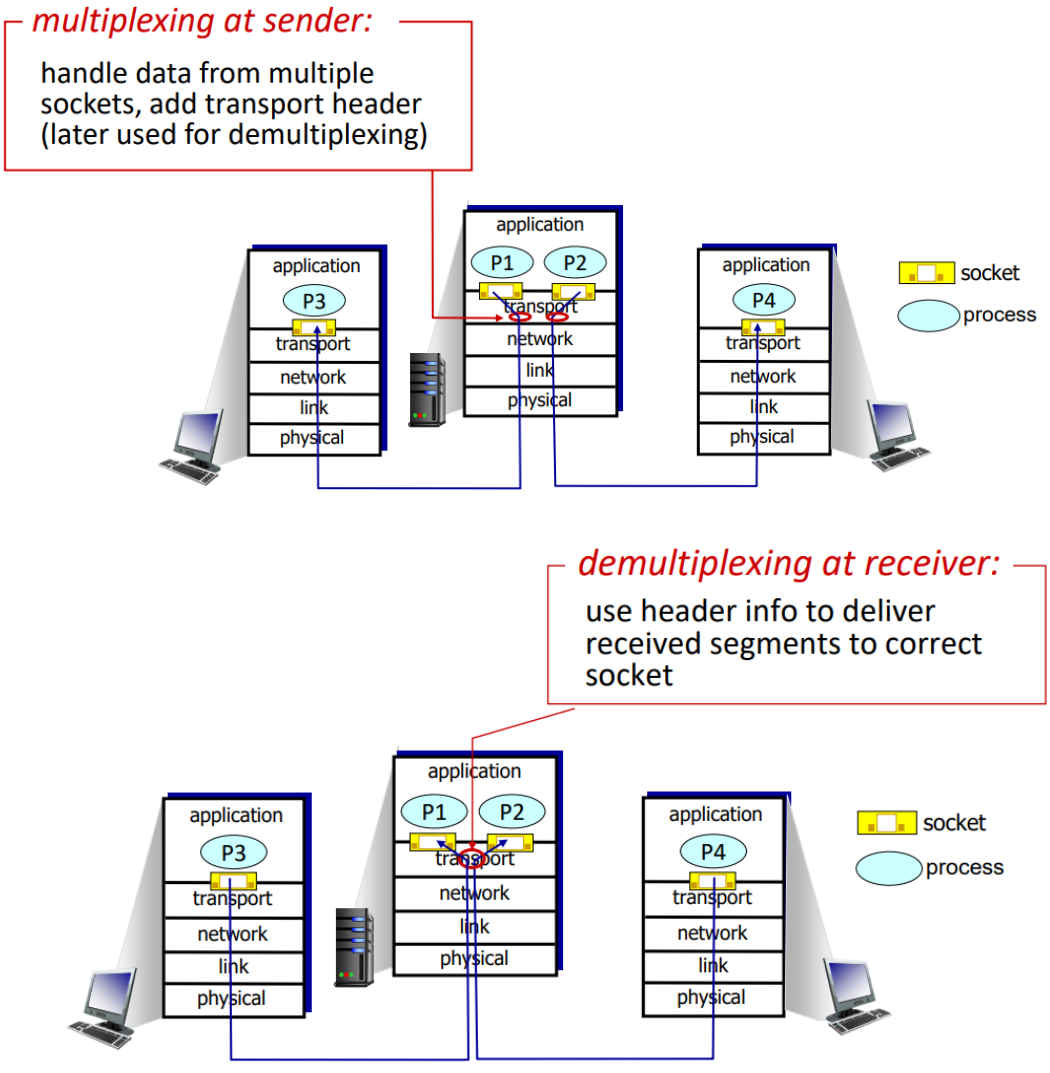
è un protocollo **connection oriented**). TCP è adatto alle applicazioni dette **elastiche**, cioè poco sensibili al ritardo ma **molto sensibile alla perdita di pacchetti**

- **User Datagram Protocol (UDP)**: Protocollo di trasporto "**best-effort**" il quale è un'estensione "senza fronzoli" del protocollo IP. Il protocollo è di tipo **connectionless** ed è adatto alle applicazioni dette **real-time**: poco sensibili alla perdita di pacchetti (in un certo grado) ma **molto sensibili al ritardo**

Nessuno dei due protocolli **garantisce qualcosa in termini di ritardo**, cioè nessuno dei due garantisce che i pacchetti saranno consegnati con un ritardo massimo di n secondi. Allo stesso modo, nessuno dei due protocolli garantisce nulla riguardo alla **banda**: nessuno dei due garantisce che la comunicazione avvenga a certi bit/s.

3.3 Multiplexing e Demultiplexing

Quando si parla di Multiplexing in generale si parla di un'operazione che porta ad unire **più flussi** in un unico flusso. Nel caso specifico del livello di trasporto, dobbiamo garantire che il servizio di multiplexing/demultiplexing sia **garantito** poiché mi permette di **estendere logicamente la comunicazione ai processi**.



quindi, l'operazione di multiplazione **porta ad aggiungere un header ai messaggi provenienti dai processi**, in modo tale che essi siano **distinguibili** una volta che arrivano al livello di rete; il quale **non ha conoscenza di quale processo ha generato quale segmento** ma si preoccupa solamente di consegnare il pacchetto **all'host corretto**. L'informazione che permette di distinguere i pacchetti è la **porta di destinazione**. Il livello di trasporto si vedrà recapitati segmenti, dal livello di rete, che avranno nel loro header dell'informazione relativa alla porta di destinazione; l'operazione di **demultiplazione** permette, a livello di trasporto, di inoltrare **verso la socket corretta** segmenti diretti verso **processi differenti**. In particolare, l'operazione di **demultiplazione** funziona nel seguente modo:

- L'host riceve i **datagrammi IP**:
  - Ogni datagramma ha un indirizzo IP sorgente e un indirizzo IP destinatario
  - Ogni datagramma trasporta un **segmento del livello di trasporto**
  - Ogni segmento ha una **numero di porta sorgente e il numero della porta di destinazione**
- L'host mittente usa **l'indirizzo IP e il numero di porta** per inviare i segmenti alla socket appropriata

3.3.1 Connectionless demultiplexing

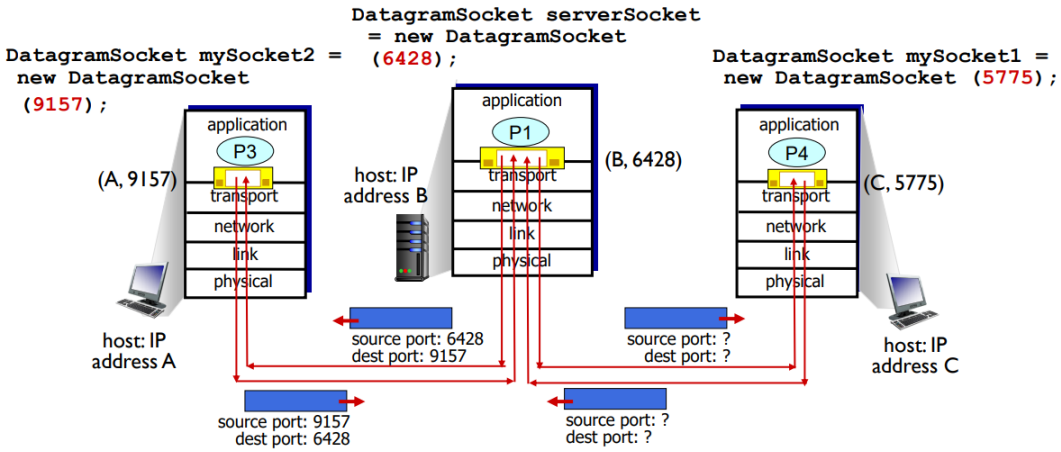
Quando si va a creare una socket, **bisogna specificare attraverso su quale porta essa è raggiungibile**. La socket di destinazione è **univocamente identificata** da una coppia di valori:

1. **L'indirizzo IP di destinazione**
2. **Il numero della porta di destinazione**

Quando l'host destinatario riceve un segmento UDP:

- Controlla il numero della porta di destinazione nel segmento
- Invia il segmento UDP alla socket con quella porta

I datagrammi IP/UDP con **la stessa porta di destinazione e stesso indirizzo IP di destinazione**, anche se da indirizzi IP sorgenti differenti e/o con differenti numeri di porta sorgente, saranno **inviati alla stessa socket** dell'host destinatario.



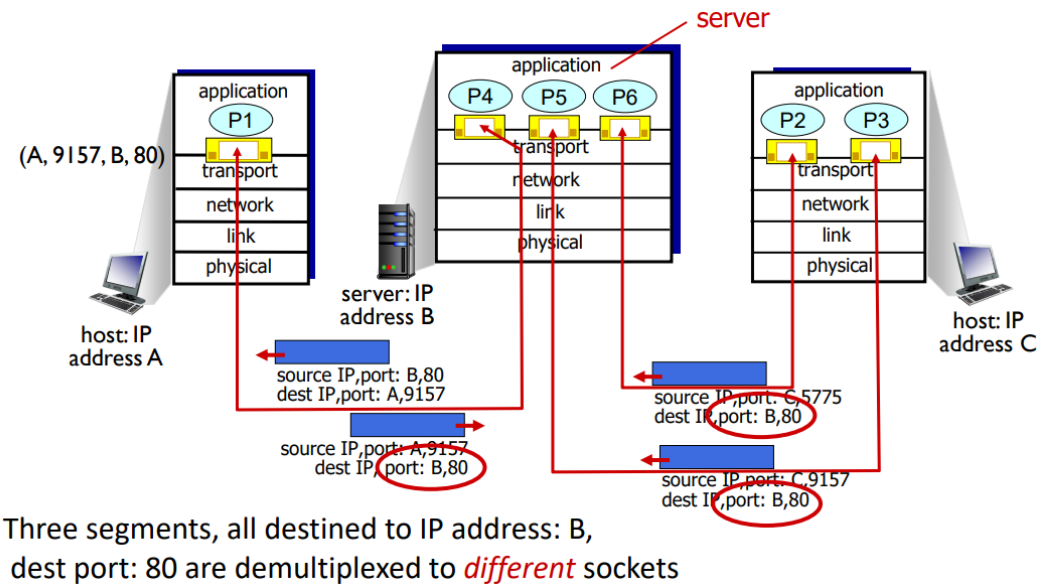
3.3.2 Connection-oriented demultiplexing

A differenza dei protocolli connectionless, una socket è qui identificata **da una quadrupla**:

- **Indirizzo IP sorgente**
- **Numero di porta sorgente**
- **Indirizzo IP destinatario**
- **Numero della porta di destinazione**

Quando viene effettuata la demultiplazione, allora **tutti e 4 i valori** verranno usati per inviare il pacchetto alla socket corretta. Ciò significa che qui si avrà **una socket dedicata ad ogni connessione**. I server **possono supportare più socke TCP simultaneamente** e:

- Ogni socket è identificata da una sua quadrupla
- Ad ogni socket è **associato un diverso client comunicante**



3.4 User Datagram Protocol (UDP)

User Datagram Protocol (UDP) è, di fatto, un'estensione "senza fronzoli" e "minimale" del protocollo IP. UDP fornisce un servizio di trasporto **"best effort"**, quindi i segmenti di UDP potrebbero essere:

- Persi
- Consegnati in disordine all'applicazione

È un **protocollo connectionless**:

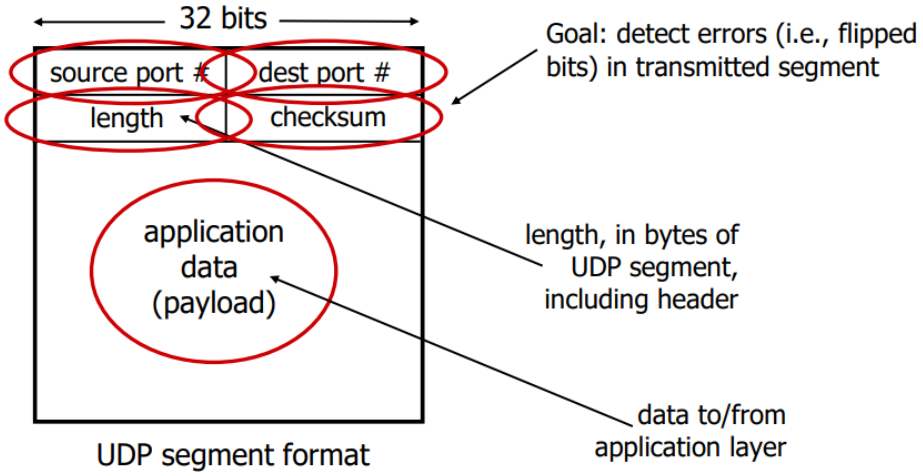
- Nessun "handshaking" (inizializzazione della connessione) tra il mittente e il destinatario

- Ogni segmento UDP viene gestito indipendentemente dagli altri

Perché è necessaria l'esistenza di UPD?

- Nessuna necessità di stabilire una connessione (il che può aggiungere ritardi)
- È **semplice**: nessuno stato di connessione al mittente e al destinatario
- **Dimensioni dell'intestazione piccole**
- **Nessun controllo di congestione**
  - UDP può **trasmettere quanto veloce si vuole**
  - Può funzionare in casi di congestione della rete (più o meno, comunque fatica in caso di congestione molto elevata)

L'header di UDP ha la seguente struttura:

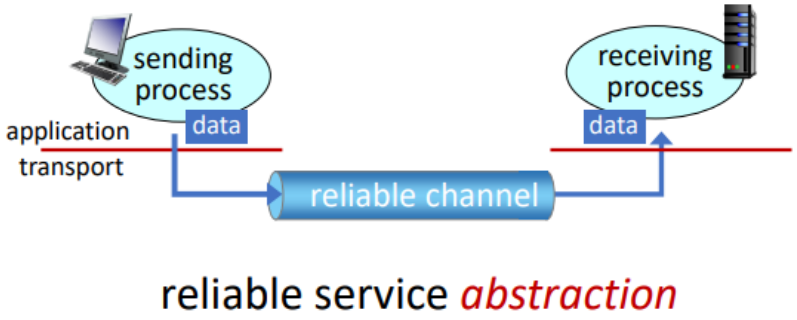


Facciamo qualche considerazione:

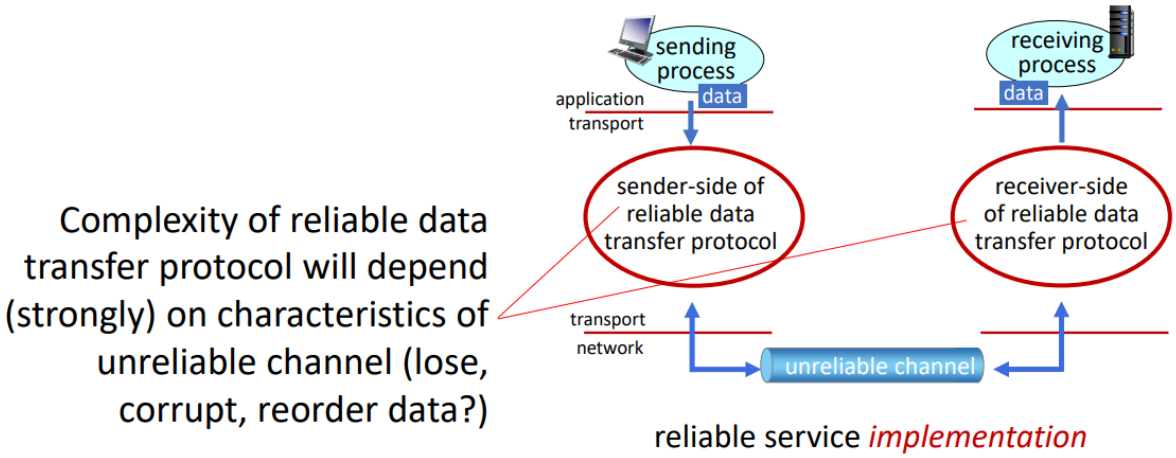
- L'header è quindi di **64 bit**, ed ogni campo è lungo **16 bit**.
- Posso avere $2^{16} - 1$ possibili porti di sorgente e destinazione (la porta con tutti 0 non è tendenzialmente una porta valida)
- Includendo l'header, al più posso avere $2^{16} - 1$ byte differenti in un segmento UDP
- Il checksum si calcola nel seguente modo: si considerino gruppi di 16 bit del segmento; si sommino modulo 2 e si faccia il complemento di essa. Lato destinatario, si effettua la stessa operazione e poi si somma con il checksum: se si ottengono tutti 0 allora non ci sono stati errori.

3.5 Principi del trasferimento dati affidabile

Per servizio di dati affidabile si intende un servizio di trasporto in cui i dati **vengono consegnati senza perdite e senza errori**. Dal punto di vista astratto vogliamo avere la seguente situazione:



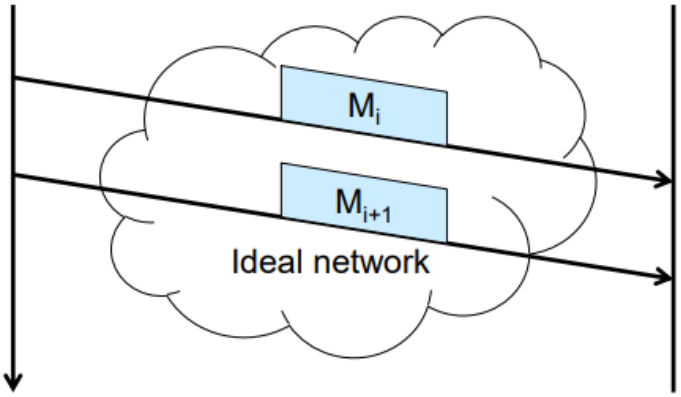
Tuttavia, nella realtà **il canale è sempre inaffidabile per definizione**: il protocollo IP è un protocollo "best-effort" e quindi non offre garanzie sulla consegna, dovremo quindi **andare ad implementare dei meccanismi di trasporto che rendano la consegna affidabile**:



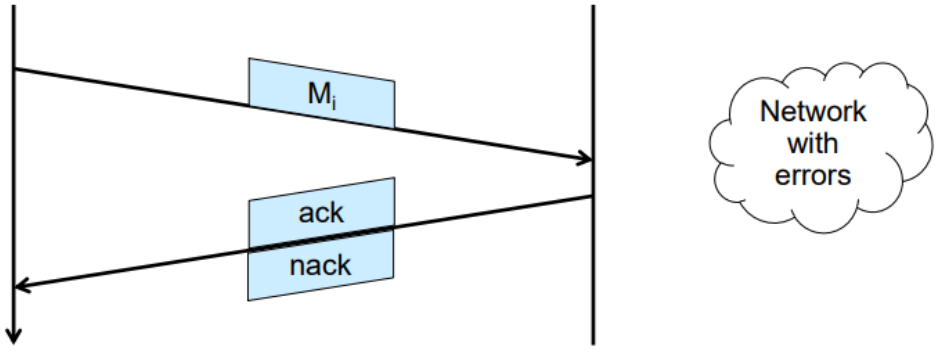
Abbiamo poi un ulteriore problema: il mittente e il destinatario **non conosco lo stato l'uno dell'altro** a meno che **non sia comunicato tramite messaggio**. Consideriamo una **rete ideale**, cioè che non introduce:

- Bit di errore
- Scarto di segmenti
- Segmenti fuori sequenza

Il livello di trasporto **allora non dovrà correggere nulla e il protocollo stesso sarà triviale**: il mittente invia i segmenti in sequenza (uno dopo l'altro) e il ricevente li riceve senza nessun ulteriore controllo



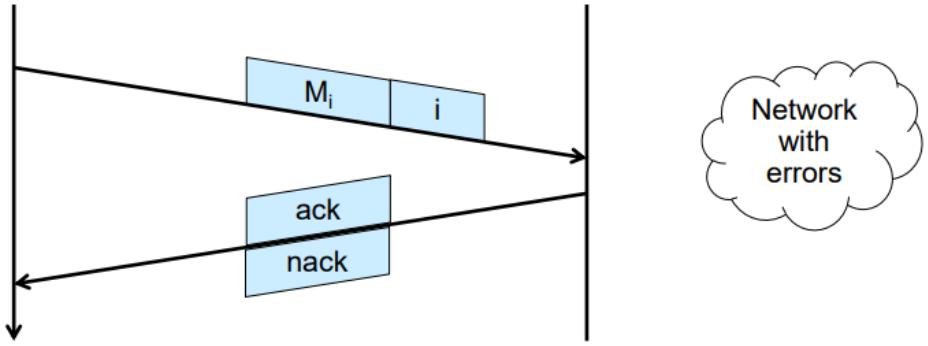
Sfortunatamente, le reti ideali non esistono. In una rete con errori, è possibile introdurre **riconoscimenti positivi** (ack) e **riconoscimenti negativi** (nack).



Un algoritmo di invio molto semplice allora potrà essere il seguente:

if *ack* **then**
  M_{i+1}
else
  if *nack* **then**
    M_i
  end
  if *?* **then** `// problema!`
    something
  end
end

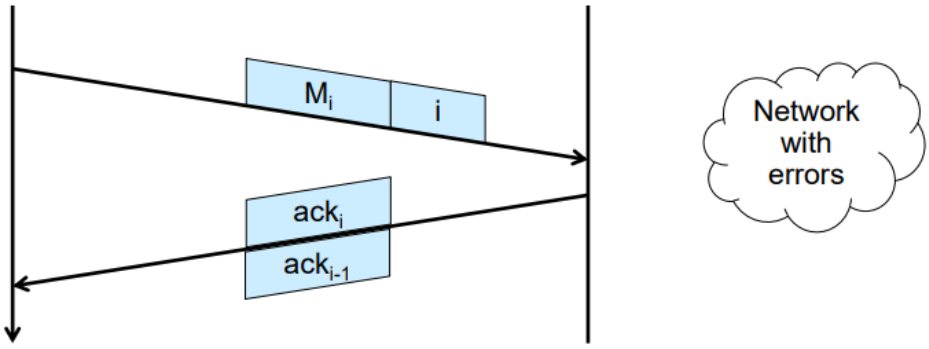
Se la rete è **soggetta ad errori** anche **gli ack possono essere soggetti ad errori**, quindi stiamo ipotizzando, con questo algoritmo, che **la perdita o la corruzione di ack non avvenga**, cosa che in realtà può accadere nelle reti reali. Un modo per mitigare questo problema è **ritrasmettere il messaggio se l'ack è corrotto**. Tuttavia, neanche questo funziona! Non ho un meccanismo per capire se **un messaggio è duplicato o meno**. Possiamo mitigare ulteriormente questo problema nel seguente modo: se i **segmenti sono numerati con un numero di sequenza nell'intestazione**, non c'è il rischio di non sapere se un messaggio è duplicato o meno



L'algoritmo di invio allora diventa:

if *ack* **then**
| M_{i+1}
else
| M_i `// è stato ricevuto un nack`
end

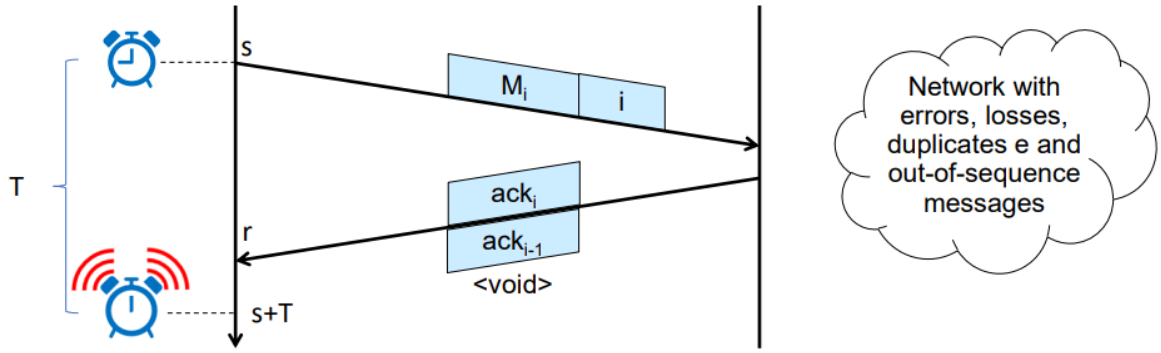
Un protocollo di questo tipo **funzionerebbe in una rete che introduce errori**. Sfortunatamente, le reti non solo introducono errori ma **anche perdite di pacchetti!**. Prima di passare a definire un protocollo resistente anche alle perdite, parliamo di **efficienza di un protocollo**: quando si definisce un nuovo protocollo, si deve cercare di limitare **il numero di messaggi diversi possibili in esso**, quindi se oltre a numerare i segmenti **numeriamo anche gli ack**



allora dovrò inserire una **regola aggiuntiva** che indica che, se non si è ricevuto il messaggio atteso, al posto di inviare un ack con numero di sequenza i dovrò inviare un ack con numero di sequenza $i-1$. L'algoritmo quindi diventa:

if *ack* **then**
| M_{i+1}
else
| M_i `// è stato ricevuto un` ack_i
end

quindi, si andrà a ritrasmettere un messaggio se **si riceve un ack duplicato**. Per rendere il protocollo resistente alle **perdite**: quando ho una perdita, può capitare che **il pacchetto inviato sia stato perso** oppure che **l'ack vada perso**; allora viene introdotto **un timer**:

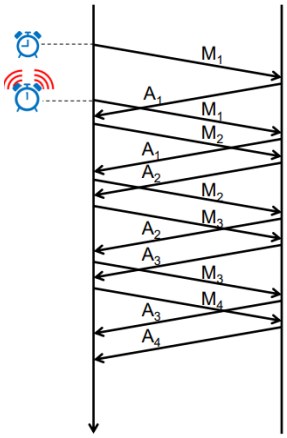


nel momento in cui **il timer scade** e non ho ancora ricevuto un ack, **rinvio lo stesso messaggio**. L'algoritmo di invio diventerà:

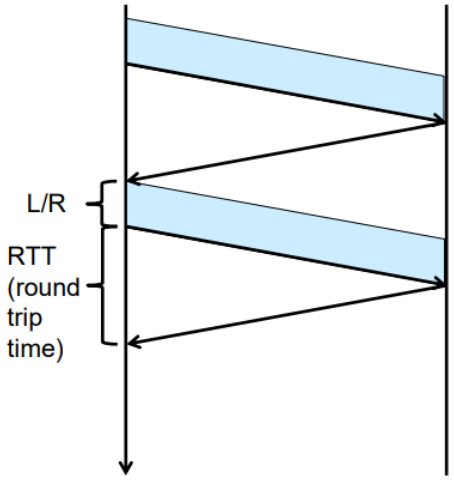
if $s+T$ *is reached* **then**
  M_i
  set$(s+T)+T$
else
  if ack_i **then** // $s+T$ `non è stato ancora raggiunto` (r)
    M_{i+1}
    set$(r+T)$
  else
    M_i
    set$(r+T)$
  end
end

Questo tipo di implementazione è la più sofistica di quelli che vengono detti **protocolli stop-and-wait**. Un protocollo stop-and-wait **funziona in una rete reale**. Tuttavia, vi sono delle problematiche:

- L'impostazione della lunghezza del timer è un problema complesso
- Un timer **troppo corto** potrebbe causare **ritrasmissioni inutili**: nell'esempio fatto sopra, ci potrebbero essere lunghe sequenze di **ritrasmissioni inutili**
- Un timer **troppo lungo** ferma per troppo tempo la trasmissione in caso di perdita



Dovrò quindi avere un timer almeno lungo quanto il periodo di tempo che intercorre tra la trasmissione del messaggio e il ricevimento di un ack, questo valore prende il nome di **Round-Trip Time** (RTT); il problema quindi passa al cercare di fare una stima accurata del RTT. La problematica principale di un protocollo stop-and-wait è la seguente: supponiamo di essere nel caso in cui il **ritardo di trasmissione** non è più trascurabile



i parallelogrammi sopra rappresentano il ritardo di trasmissione, il quale vale L/R. Assumiamo, per semplicità, che gli ack abbiano un ritardo di trasmissione trascurabile. Se siamo in un protocollo stop-and-wait, in questo caso esso è **altamente inefficiente**. Una misura dell'efficienza del protocollo è la **utilità** (utility), la quale si calcola nel seguente modo:

$$U = \frac{\frac{L}{R}}{RTT + \frac{L}{R}}$$