

Reti

Matteo Genovese

September 2024

Contents

1 Reti di calcolatori e Internet	3
1.1 Cos'è internet?	3
1.1.1 Definizioni varie	3
1.2 Il nucleo della rete	4
1.2.1 Comutazione di pacchetto	4
1.2.2 Comutazione di circuito	6
1.3 Ritardi, perdite e throughput nelle reti a commutazione di pacchetto	6
1.3.1 Panoramica del ritardo nelle reti a commutazione di pacchetto	7
1.3.2 Ritardo end-to-end	7
1.3.3 Throughput nelle reti di calcolatori	8
1.4 Livelli dei protocolli e loro modelli di servizio	8
1.4.1 Architettura a livelli	8
1.4.2 Incapsulamento	10
2 Livello di applicazione	12
2.1 Princìpi delle applicazioni di rete	12
2.1.1 Architetture delle applicazioni di rete	12
2.1.2 Processi comunicanti	12
2.1.3 Servizi di trasporto disponibili per le applicazioni	13
2.1.4 Servizi di trasporto offerti da Internet	14
2.1.5 Protocolli a livello di applicazione	15
2.2 Web e HTTP	15
2.2.1 Panoramica di HTTP	15
2.2.2 Connessioni persistenti e non persistenti	15
2.2.3 Formato dei messaggi HTTP	17
2.2.4 Cookie	19
2.2.5 Web caching (proxy server)	19
2.3 Posta elettronica	20
2.3.1 SMTP	21
2.4 DNS	22

2.4.1	Gestione gerarchica DNS	22
2.4.2	DNS locale	23
2.4.3	Record DNS	24
3	Livello di trasporto	26
3.1	Introduzione e servizi a livello di trasporto	26
3.1.1	Protocolli utilizzati	26
3.2	Multiplexing e demultiplexing	26
3.3	Trasporto senza connessione: UDP	28
3.4	Principi del trasferimento dati affidabile	29
3.4.1	Protocolli con pipeline	36
3.4.2	Go-Back-N	36
3.4.3	Selective Repeat	38
3.5	TCP: trasporto orientato alla connessione	40
3.5.1	Struttura dei segmenti	40
3.5.2	Gestione numeri di sequenza e riscontro del TCP	41
3.5.3	Gestione del timer nel TCP	42
3.5.4	Trasferimento dati affidabile del TCP	43
3.5.5	Controllo del flusso	44
3.6	Principi del controllo di congestione	46
3.7	Controllo di congestione	49
3.8	Programmazione delle socket	52
3.8.1	Programmazione socket TCP	53
3.8.2	Programmazione socket UDP	55
4	Livello rete	56
4.1	Architettura del router	56
4.1.1	Tabelle di inoltro	57
4.2	Protocollo internet: IP	57
4.3	IPv4, Protocollo IP versione 4	58
4.3.1	Formato dei datagrammi	58
4.3.2	Frammentazione dei datagrammi IP	59
4.3.3	Sottorete	59
4.3.4	Assegnazione indirizzi internet CIDR	60
4.3.5	Indirizzamento - convenzioni	60
4.3.6	Netmask	60
4.3.7	Inoltro dei pacchetti: Host	61
4.3.8	Inoltro dei pacchetti: Router	61
4.3.9	Come ottenere un blocco di indirizzi IP	61
4.3.10	Come ottenere un singolo indirizzo	61
4.3.11	DHCP	62
4.3.12	NAT	62
4.4	IPv6	63
4.4.1	Formato dei datagrammi	63
5	Livello di rete (Piano di controllo)	65

1 Reti di calcolatori e Internet

1.1 Cos'è internet?

Internet è una rete globale di calcolatori interconnessi, spesso descritta come una 'rete di reti', che comunica utilizzando un insieme comune di protocolli, principalmente la suite TCP/IP.

1.1.1 Definizioni varie

host o *sistemi periferici* sistema terminale della rete dove risiedono e vengono eseguite le applicazioni.

rete di collegamenti

o *communication link* il mezzo fisico (es. cavo coassiale, fibra ottica, onde radio) attraverso cui i dati vengono trasmessi tra i nodi della rete.

commutatori di pacchetti

o *packet switch* dispositivi di rete che inoltrano i pacchetti di dati da un collegamento in ingresso a un collegamento in uscita.

velocità di trasmissione

o *transmission rate* velocità con cui i vari tipi di collegamenti si scambiano dati, misurata in **bit/secondo (bps)** e che rappresenta la capacità del collegamento..

pacchetto

o *packet* unità di dati formata a livello di rete (livello IP), contenente una porzione di dati (che a livello di trasporto, con TCP, è chiamata **segmento**) e un'intestazione con informazioni di controllo.

commutatore di pacchetto

dispositivo che riceve un **pacchetto** su un collegamento in ingresso, lo memorizza temporaneamente e poi lo ritrasmette su un collegamento in uscita. I due tipi principali sono i **router**, usati nel nucleo della rete per l'instradamento tra reti diverse, e i **commutatori a livello di collegamento (link-layer switch)**, usati nelle reti locali per la commutazione all'interno della stessa rete.

percorso

o *route* o *path*, sequenza di collegamenti e di commutatori di pacchetto attraversata dal singolo pacchetto.

Internet ServiceProvider (ISP)

un'organizzazione che possiede e gestisce un insieme di commutatori di pacchetto e di collegamenti, fornendo ai sistemi periferici e ad altre reti l'accesso a Internet.

applicazioni distribuite

o *distributed applications*, più sistemi periferici che si scambiano reciprocamente dati. Vengono eseguite sui sistemi periferici (*host*).

interfaccia socket

o *socket interface* un’interfaccia di programmazione (API) che specifica come un programma eseguito su un sistema periferico può richiedere ai servizi di rete di Internet di recapitare dati a un programma eseguito su un altro sistema periferico.

protocollo

è un insieme di regole rigido, definisce il formato, l’ordine dei messaggi scambiati tra due o più entità in comunicazione, così come le azioni intraprese in fase di trasmissione e/o di ricezione di un messaggio o di un altro evento.

client e server

host che richiedono dei servizi (client) e *host* che si occupano di erogare dei servizi (server). Questi ultimi sono spesso collocati in potenti **data center** per garantire alta disponibilità e prestazioni.

reti di accesso

o *access network* la rete che connette fisicamente i sistemi periferici al primo **router** (chiamato **router di bordo**, o *edge router*) sul percorso verso la rete Internet del provider.

1.2 Il nucleo della rete

Il nucleo della rete è una maglia complessa di commutatori di pacchetti e collegamenti ad alta velocità che interconnettono i sistemi periferici di Internet. La funzione principale del nucleo della rete è quella di instradare i dati tra i sistemi periferici.

1.2.1 Comutazione di pacchetto

Le *applicazioni distribuite* scambiano **messaggi**. Per facilitare la trasmissione e la gestione della rete, la sorgente divide i messaggi più lunghi in unità più piccole chiamate **pacchetti**, che viaggiano attraverso i *commutatori di pacchetto* per raggiungere la destinazione. Ogni pacchetto viene trasmesso sul collegamento fisico alla velocità di trasmissione R (misurata in bit al secondo, bps). Un pacchetto di L bit impiegherà quindi L/R secondi per essere trasmesso completamente sul collegamento.

Una delle tecnologie fondamentali utilizzate dai *commutatori di pacchetto* è la **trasmissione store and forward**. Secondo questo principio, il commutatore deve ricevere **completamente** l’intero **pacchetto** prima di iniziare a trasmettere il primo bit verso il collegamento di uscita.

Consideriamo un pacchetto di L bit che viene trasmesso dall’*host* sorgente al primo *router* sul percorso. Se la velocità di trasmissione del collegamento è

R , il tempo necessario per trasmettere il pacchetto è L/R secondi. Utilizzando la trasmissione store and forward, il router riceverà l'intero pacchetto all'istante L/R . Supponendo che anche il collegamento tra il router e l'*host* di destinazione abbia una velocità di trasmissione R , il router inizierà a trasmettere il pacchetto verso la destinazione. L'*host* di destinazione riceverà l'intero pacchetto all'istante L/R (trasmissione host-router) + L/R (trasmissione router-host) = $2L/R$.

Generalizzando, considerando un percorso con N collegamenti, ognuno con una velocità di trasmissione R , e quindi $N - 1$ router intermedi che utilizzano la trasmissione store and forward, il ritardo totale di trasmissione attraverso il percorso (tralasciando altri tipi di ritardi come la propagazione) sarà:

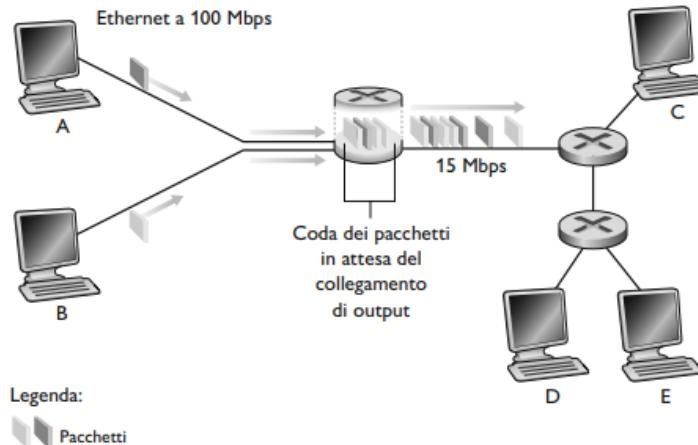
$$\text{delay} = N \frac{L}{R}$$

Se la sorgente deve inviare P pacchetti in sequenza sullo stesso percorso, il tempo necessario affinché l'ultimo pacchetto raggiunga la destinazione sarà approssimativamente $P \cdot N \frac{L}{R}$, assumendo che ogni pacchetto venga trasmesso solo dopo che il precedente è stato completamente trasmesso.

Nota: Se i pacchetti vengono inviati in modo continuo (*pipelining*), l'analisi del ritardo diventa più complessa. Il ritardo per il primo pacchetto rimane $N \frac{L}{R}$, ma i pacchetti successivi arriveranno a intervalli di $\frac{L}{R}$.

Tralasciando però i ritardi di propagazione, che rappresentano il tempo impiegato dal segnale per viaggiare attraverso il mezzo fisico.

I commutatori di pacchetto hanno più collegamenti, e per ogni collegamento di output è presente un **buffer di output** o **coda di output** per organizzare i pacchetti da inviare su quel collegamento. Questo comporta che i pacchetti subiscono un **ritardo di accodamento**, il pacchetto deve aspettare che si "liberi il passaggio" per essere trasmesso. Questo ritardo è variabile e dipende dal traffico della rete in un dato momento. Nel caso in cui il buffer sia pieno, avendo una dimensione prestabilita, il pacchetto verrà perso (*packet loss*), verrà eliminato o il pacchetto in arrivo o uno di quelli in coda, dipende dal progettista di rete.

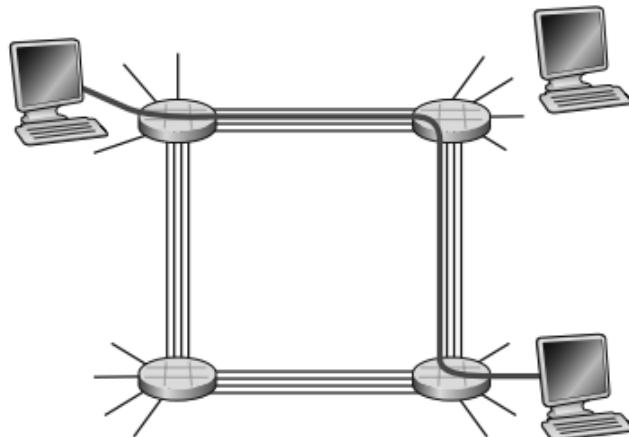


1.2.2 Comutazione di circuito

Esiste un altro metodo per scambiare messaggi, ossia la **commutazione di circuito**. Questo garantisce che le risorse per consentire lo scambio di dati sono **dedicate** (o **riservate**), sono quindi esclusivamente allocate per la **comunicazione**.

Bisogna stabilire un collegamento tra mittente e destinatario, lo chiameremo **circuito**, attraverso una fase di segnalazione tra i commutatori. A questo circuito verrà riservata una **velocità di trasmissione costante** (pari a una frazione della capacità del canale).

Ogni *host* della rete è connesso a un commutatore che a sua volta è connesso con gli altri *host* nella rete. Quando un host dovrà comunicare con un altro host, verrà instaurata una **connessione punto a punto** (*connessione end to end*) **logica** a loro dedicata.



Un circuito viene implementato tramite **multiplexing a divisione di frequenza** (*frequency-division multiplexing*) o **multiplexing a divisione di tempo** (*time-division multiplexing*).

Con il **FDM** (multiplexing a divisione di frequenza), lo spettro di frequenza disponibile viene diviso in bande di frequenza più piccole, ognuna delle quali viene dedicata a una connessione. A ciascuna connessione viene quindi dedicata un'**ampiezza di banda** (*bandwidth*) specifica.

Con il **TDM** (multiplexing a divisione di tempo), il tempo viene diviso in intervalli (slot di tempo) e a ciascuna connessione viene assegnato uno slot di tempo in cui può trasmettere i dati. Gli slot di tempo vengono assegnati in modo ciclico.

1.3 Ritardi, perdite e throughput nelle reti a comunicazione di pacchetto

Per le leggi fisiche non è possibile scambiare dati istantaneamente. Le reti introducono ritardi, perdono pacchetti e limitano il **throughput** (la quantità

di dati al secondo che può essere trasferita tra due sistemi periferici) a causa di fattori come la congestione della rete e le limitazioni fisiche dei collegamenti.

Esistono modi per affrontare questo problema.

1.3.1 Panoramica del ritardo nelle reti a commutazione di pacchetto

I principali tipi di ritardi sono:

- **ritardo di elaborazione** (*processing delay*): include il tempo per esaminare l'intestazione del pacchetto e determinare dove trasmetterlo e il tempo per controllare se ci sono errori a livello di bit.
- **ritardo di accodamento** (*queuing delay*): il pacchetto in coda attende la trasmissione sul collegamento nel caso in cui il buffer non sia libero. Questo ritardo è variabile e dipende dalla congestione della rete. Nel caso in cui il buffer sia libero, il ritardo è nullo.
- **ritardo di trasmissione** (*transmission delay*): utilizzando la politica **FIFO**. Avendo L bit da trasmettere con R bps di velocità di trasmissione, avremmo un ritardo pari a L/R , cioè il tempo richiesto per la trasmissione di tutti i bit nel collegamento.
- **ritardo di propagazione** (*propagation delay*): tempo che il pacchetto impiega una volta immesso sul collegamento, viaggia a una velocità di propagazione del collegamento v per una distanza d , quindi il ritardo sarà d/v .

che sommati formano il **ritardo totale di nodo** (*node delay*), ovvero la somma del ritardo di elaborazione, accodamento, trasmissione e propagazione:

$$d_{nodo} = d_{elaborazione} + d_{accodamento} + d_{trasmissione} + d_{propagazione}$$

1.3.2 Ritardo end-to-end

Ipotizziamo di avere una rete con $N - 1$ router tra l'host sorgente e l'host di destinazione. Inoltre, supponiamo di avere una rete non congestionata, quindi con ritardo di accodamento nullo. Questa è una *semplificazione* che ci permette di calcolare il ritardo end-to-end in modo più semplice. In questo caso, il **ritardo dalla sorgente alla destinazione** (*end-to-end delay*) è dato dalla somma dei ritardi di nodo su tutti i N collegamenti del percorso, e può essere calcolato con la formula:

$$d_{end-to-end} = N \cdot (d_{elaborazione} + d_{trasmissione} + d_{propagazione})$$

dove N rappresenta il numero di **collegamenti** (*hop*) nel percorso tra la sorgente e la destinazione. È importante notare che, in scenari reali, il ritardo di accodamento è spesso un fattore significativo e non può essere trascurato.

1.3.3 Throughput nelle reti di calcolatori

Definiamo **throughput istantaneo** la frequenza (bit/tempo) alla quale i file sono trasferiti tra mittente e destinatario **in un determinato istante**.

Definiamo **throughput medio** la frequenza (bit/tempo) alla quale i file sono trasferiti tra mittente e destinatario **in un periodo di tempo**.

In una rete, il **throughput** è spesso limitato dal **collegamento collo di bottiglia**, ovvero il collegamento con la velocità di trasmissione più bassa lungo il percorso tra mittente e destinatario. In presenza di più router tra sorgente e destinazione, il throughput sarà determinato dalla velocità del collegamento più lento.

Consideriamo ora i seguenti scenari, dove R_s è la velocità di trasmissione del collegamento tra il server e il router, e R_c è la velocità di trasmissione del collegamento tra il router e il client:

- **Caso 1:** $R_s < R_c$: In questo caso, il collegamento collo di bottiglia è il collegamento tra il server e il router. Il throughput sarà limitato da R_s , quindi il throughput massimo raggiungibile sarà pari a R_s .
- **Caso 2:** $R_s > R_c$: In questo caso, il collegamento collo di bottiglia è il collegamento tra il router e il client. Il throughput sarà limitato da R_c , quindi il throughput massimo raggiungibile sarà pari a R_c .
- **Caso 3:** $R_s = R_c$: In questo caso, non c'è un collo di bottiglia evidente. Il throughput massimo raggiungibile sarà pari a R_s (o R_c , dato che sono uguali).

È importante notare che, in scenari reali, il throughput può essere influenzato anche da altri fattori, come la congestione della rete e la presenza di altri flussi di dati.

1.4 Livelli dei protocolli e loro modelli di servizio

1.4.1 Architettura a livelli

L'**architettura di internet** è stata progettata a **livelli o strati (layer)**, ciascun protocollo e funzione appartiene a un livello. Questa architettura ci permette di utilizzare i servizi di un livello superiore, senza doverci preoccupare dei dettagli di implementazione del livello inferiore. Questo è il concetto di **modello di servizio (service model)** di un livello, dove ogni livello offre un insieme di servizi ben definiti al livello superiore, nascondendo la complessità del livello sottostante.

Un livello di protocolli può essere implementato sia a livello **hardware** che **software**. Per esempio, a **livello di applicazione o trasporto** troviamo protocolli implementati via software, mentre a **livello fisico e data link** abbiamo dei collegamenti fisici, quindi sono implementati via hardware. Il **livello di rete** ha un'implementazione **mista**, con alcune funzioni implementate via software e altre via hardware.

Un protocollo di livello n lo possiamo trovare su più sistemi periferici, commutatori di pacchetto e altri componenti della rete. In ogni componente della rete è presente un' *entità di protocollo* di livello n che implementa una parte del protocollo.

La modularità di questa architettura, basata sul principio dell'astrazione, rende più facile aggiornare la componentistica e i protocolli di un livello senza influenzare gli altri livelli. I protocolli dei vari livelli sono detti **pila di protocolli** (*protocol stack*), una pila **gerarchica** di protocolli dove ogni livello si basa sui servizi forniti dal livello sottostante. Esaminiamoli con un approccio **top-down**.

Livello di applicazione

Il **livello di applicazione** (*application layer*) è la sede delle applicazioni di rete e dei relativi protocolli (per internet: HTTP, SMTP, FTP). Anche il protocollo DNS fa parte di questo livello. I protocolli di questo livello definiscono come le applicazioni comunicano tra loro.

È distribuito su più sistemi periferici: un'applicazione di un sistema periferico scambia **messaggi** con un'altra applicazione di un sistema periferico tramite i protocolli di questo livello.

Livello di trasporto

Il **livello di trasporto** (*transport layer*) trasferisce i messaggi del livello applicazione tra punti periferici gestiti da applicazione. I protocolli che troviamo sono TCP (connection-oriented) e UDP (connectionless). In questo livello chiameremo **segmenti** i pacchetti.

Livello di rete

Il **livello di rete** (*network layer*) si occupa di trasferire i pacchetti a livello di rete da un host a un altro. I pacchetti in questo livello vengono chiamati **datagrammi**. Questo livello riceve dal livello di trasporto il segmento e un indirizzo IP di consegna. Il livello di rete comprende il protocollo IP (sia versione 4 che 6), inoltre comprende i protocolli di instradamento.

Viene anche chiamato **livello IP**, poiché il protocollo IP è il collante di internet.

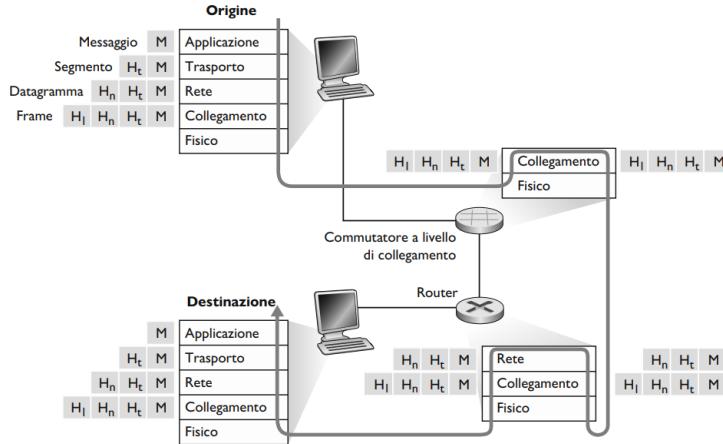
Livello di collegamento

Il **livello di collegamento** (*data link layer*) instrada un datagramma attraverso una serie di router tra sorgente e destinazione. Riceve il datagramma dal livello di rete, il suo compito sarà di trasportarlo al prossimo nodo (un singolo 'hop'), poi il nodo successivo lo passerà al livello di rete.

I protocolli di questo livello possono essere: Ethernet o Wi-Fi. In questo livello i pacchetti vengono chiamati **frame**.

Livello di fisico

Il **livello fisico (physical layer)** trasferisce i singoli bit del frame da un nodo a quello successivo. Dipende dal livello collegamento e dal mezzo trasmisivo (fibra ottica o rame), occupandosi della trasmissione fisica dei bit.



1.4.2 Incapsulamento

L'**incapsulamento** è il processo mediante il quale i dati vengono avvolti con informazioni di intestazione aggiuntive man mano che si spostano verso il basso nella pila protocollo. Ogni livello aggiunge la propria intestazione, che contiene informazioni di controllo rilevanti per la funzione di quel livello. Questo processo è cruciale per consentire la comunicazione tra i diversi livelli e attraverso reti diverse.

Payload: Il *payload* di un pacchetto è la parte di dati che viene trasportata, ovvero i dati effettivi che il livello superiore vuole trasmettere. Ad esempio, il payload di un segmento TCP è il messaggio del livello applicazione, mentre il payload di un datagramma IP è il segmento TCP.

- **Origine dei Dati al Livello di Applicazione:** Il processo inizia al livello di applicazione, dove vengono generati i dati dell'utente (es. un messaggio email, una richiesta di pagina web). Questi dati sono spesso chiamati "messaggio" e costituiscono il payload per i livelli inferiori.
- **Incapsulamento al Livello di Trasporto:** Il livello di applicazione passa il messaggio al livello di trasporto. Il livello di trasporto aggiunge la propria intestazione al messaggio, creando un "segmento". Questa intestazione contiene informazioni come i numeri di porta, utilizzati per identificare l'applicazione specifica sugli host mittente e destinatario. Il livello di trasporto potrebbe anche aggiungere numeri di sequenza e checksum per una consegna affidabile dei dati (nel caso di TCP). Il payload di questo segmento è il messaggio del livello applicazione.

- **Incapsulamento al Livello di Rete:** Il segmento viene quindi passato al livello di rete. Il livello di rete aggiunge la propria intestazione, creando un "datagramma". Questa intestazione contiene gli indirizzi IP di origine e destinazione, utilizzati per instradare il datagramma attraverso la rete. Il payload di questo datagramma è il segmento del livello di trasporto.
- **Incapsulamento al Livello di Collegamento:** Il datagramma viene passato al livello di collegamento. Il livello di collegamento aggiunge la propria intestazione e coda, creando un "frame". Questa intestazione contiene informazioni come gli indirizzi MAC, utilizzati per consegnare il frame attraverso un singolo collegamento. La coda spesso contiene un checksum per il rilevamento degli errori. Il payload di questo frame è il datagramma del livello di rete.
- **Trasmissione al Livello Fisico:** Infine, il frame viene passato al livello fisico, che trasmette i bit del frame attraverso il mezzo fisico (es. cavo di rame, fibra ottica, segnale wireless).
- **Decapsulamento alla Destinazione:** Quando i dati raggiungono la destinazione, il processo viene invertito. Ogni livello all'estremità ricevente rimuove la propria intestazione corrispondente, passando i dati al livello superiore successivo. Questo processo è chiamato decapsulamento.

In sostanza, l'incapsulamento è il processo di avvolgere i dati con intestazioni man mano che si spostano verso il basso nella pila protocollare, consentendo la comunicazione tra i livelli e attraverso le reti.

2 Livello di applicazione

2.1 Princìpi delle applicazioni di rete

2.1.1 Architetture delle applicazioni di rete

Lo sviluppatore di applicazioni deve progettare l'**architettura dell'applicazione** (*application architecture*). Ci sono due tipi di architettura: **client-server** o **P2P** (*peer-to-peer*).

L'architettura **client-server** si basa su un *server* che sarà sempre attivo, con un indirizzo IP statico e sarà connesso con altri server, mentre il *client* è colui che **inizia** la comunicazione con il server. Non succederà mai che il server proverà a contattare il client, e i client avranno degli indirizzi IP dinamici. Costi alti per via dell'installazione e manutenzione. Nel caso in cui il server cada e non sarà più raggiungibile tutta la rete cadrà, quindi abbiamo un **singolo punto di fallimento** che è il server.

L'architettura **P2P pura** si basa sulla comunicazione tra i vari *peer*, dove ogni peer agisce sia come *client* che come *server*. Non esiste un *host* sempre attivo. È un'architettura scalabile, poiché le risorse sono distribuite tra i vari peer, ma difficile da gestire per la sicurezza (tutti gli host devono essere protetti adeguatamente, se uno solo non è protetto tutta la rete è insicura) visto che manca un punto centrale di controllo, e ogni peer è un potenziale punto di fallimento.

Esiste un'architettura **ibrida**, quindi un mix di architettura client-server e P2P. Il server serve come mezzo di ricerca, fungendo da directory o tracker. I peer mandano una richiesta al server che la inoltra agli altri peer, mettendo poi in comunicazione i peer nella modalità P2P.

2.1.2 Processi comunicanti

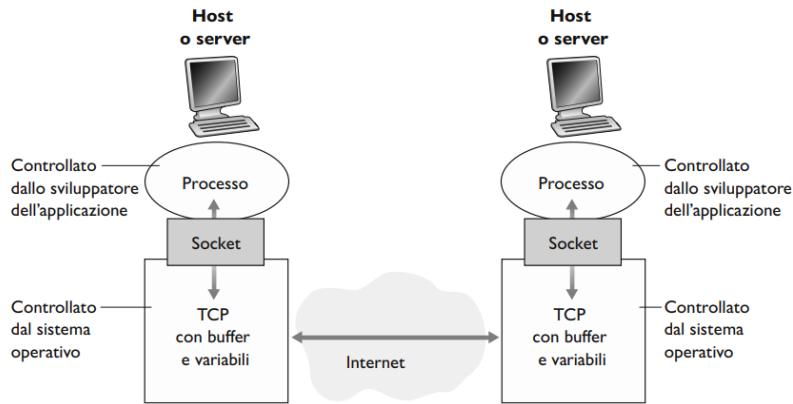
Nei sistemi operativi i programmi prendono nome di **processi comunicanti**.

- **Processo:** programma in esecuzione su un host. Più processi sullo stesso host comunicano tramite **schemi interprocesso**, mentre processi su host differenti comunicano tramite scambio di messaggi.
- **Processo client:** processo che dà inizio alla comunicazione.
- **Processo server:** processo che attende di essere contattato.
- **Socket:** Il processo comunica tramite un socket, un'interfaccia che mette in comunicazione il livello del processo applicativo e il livello trasporto. È equiparabile a una porta logica.
- **API:** Application Programming Interface, definisce come un'applicazione accede ai servizi di trasporto. È un'interfaccia di programmazione.

Le applicazioni con architettura P2P hanno sia processi client che server. Il progettista dell'applicazione può scegliere il protocollo di trasporto e alcuni parametri a livello di trasporto.

Per identificare il processo ricevente bisogna avere due informazioni:

- **Indirizzo dell'host:** specificato dal loro **indirizzo IP**, un indirizzo logico di 32 bit che identifica univocamente l'host.
- **Identificatore del processo ricevente sull'host di destinazione:** la sua **socket**, il **numero di porta di destinazione** svolge questo compito, identificando un processo specifico sull'host.



2.1.3 Servizi di trasporto disponibili per le applicazioni

L'applicazione lato mittente trasmette i messaggi tramite la socket. Lato ricevente, il protocollo a livello di trasporto deve consegnare i messaggi alla socket del processo ricevente, fornendo una *comunicazione logica* tra le applicazioni.

Esistono vari servizi offerti dai protocolli a livello di trasporto, tra cui: trasferimento dati affidabile, throughput, temporizzazione e sicurezza.

Trasferimento dati affidabile

Molte applicazioni hanno la necessità di ricevere ogni pacchetto che gli viene mandato, poiché la perdita di pacchetto potrebbe causargli dei danni, per esempio economici. Hanno bisogno di un protocollo con un servizio di consegna garantita dei dati, cioè il **trasferimento affidabile dei dati** (*reliable data transfer*), che si ottiene tramite meccanismi come acknowledgment e ritrasmissione.

Se il protocollo a livello di trasporto fornisce questo servizio, il processo mittente manderà i dati sapendo che arriveranno tutti a destinazione.

Alcune applicazioni, per esempio quelle multimediali, accettano la possibilità di perdita di dati poiché preferiscono la maggiore velocità di trasmissione, si dicono **applicazioni che tollerano le perdite** (*loss-tolerant applications*), ovvero applicazioni che possono tollerare una certa perdita di dati senza impatti significativi sulla loro funzionalità.

Throughput

Garantire il servizio di **throughput** significa garantire una certa quantità di banda per il collegamento, ovvero una certa **velocità** con cui i dati vengono trasferiti. Queste applicazioni vengono chiamate **applicazioni sensibili alla banda** (*bandwidth-sensitive applications*), ovvero applicazioni che richiedono un **throughput minimo garantito**.

Esistono delle applicazioni che non hanno bisogno di una quantità di throughput garantita, ma sono elastiche, da qui il nome **applicazioni elastiche** (*elastic applications*), ovvero applicazioni che si adattano al throughput disponibile in quell'istante.

Temporizzazione

Per le applicazioni in tempo reale è necessaria una comunicazione quasi istantanea, hanno bisogno della garanzia che la trasmissione avvenga sotto un determinato tempo. Queste garanzie di temporizzazione sono spesso difficili da ottenere nella pratica.

Sicurezza

Infine, un protocollo a livello di trasporto può garantire la sicurezza dei dati mediante, per esempio, la crittografia. I servizi di sicurezza possono includere confidenzialità, integrità e autenticazione.

2.1.4 Servizi di trasporto offerti da Internet

Esistono due protocolli di trasporto per le applicazioni di internet:

- **TCP**: prevede una connessione e un trasporto affidabile dei dati. **Servizio orientato alla connessione** (*connection-oriented service*): fa in modo che client e server si scambino informazioni di controllo a livello di trasporto prima che i messaggi a livello di applicazione comincino a fluire. Questa procedura, denominata **handshaking**, stabilisce uno *stato* sia sul client che sul server, pre-allertandoli per lo scambio di messaggi. Dopo la fase di *handshaking* si dice che esiste una **connessione TCP** tra le socket di client e host, una connessione di tipo *full-duplex* (i processi possono scambiare messaggi contemporaneamente in entrambe le direzioni). **Servizio di trasferimento affidabile** (*reliable data transfer service*): il protocollo TCP garantisce ai processi il flusso di dati senza errori, utilizzando meccanismi come acknowledgment, ritrasmissione e numeri di sequenza. TCP evita la congestione della rete, **attivamente** strozzando il flusso quando è eccessivo.
- **UDP**: protocollo minimale, è "senza connessione" (non ha bisogno di *handshaking*), ciò non garantisce l'affidabilità della connessione. UDP non garantisce la consegna, l'ordine o l'assenza di duplicati. Non ha un

meccanismo di gestione della congestione, manda il flusso di dati al livello di rete a qualsiasi velocità, lasciando la gestione della congestione all'applicazione.

2.1.5 Protocolli a livello di applicazione

I protocolli a livello di applicazione devono stabilire:

- i tipi di messaggi scambiati (richiesta o risposta), definendo lo **scopo** del messaggio.
- la sintassi dei vari tipi di messaggio, definendo il **formato** del messaggio.
- la semantica dei campi, definendo il **significato** dei campi all'interno del messaggio.
- le regole per regolare quando e come un processo invia e risponde ai messaggi, definendo la **sequenza** dei messaggi e le **azioni** intraprese dai processi comunicanti.

2.2 Web e HTTP

Il **Web** è *on demand*, ciò significa che si può avere quello che si vuole quando si vuole, accedendo a risorse tramite **URL** (*Uniform Resource Locator*).

2.2.1 Panoramica di HTTP

HTTP (*HyperText Transfer Protocol*) è un protocollo a livello di applicazione *request-response* che utilizza **TCP** come protocollo di trasporto, implementato a due programmi, client e server. Una **pagina web** è un documento costituito da *più* oggetti, un **oggetto** è un file indirizzabile tramite un **URL**. In un *URL* ci sono tre componenti: il protocollo (es. ‘*http://*’), il nome dell'host e il percorso dell'oggetto. I **browser** implementano il protocollo *HTTP* lato client, *richiedendo* oggetti. I **server web** implementano il protocollo *HTTP* lato server (Apache è un esempio), *fornendo* oggetti e saranno sempre attivi con un indirizzo IP statico. *HTTP* è un protocollo **senza memoria di stato** (*stateless protocol*), ovvero i server HTTP non mantengono alcuna informazione sulle richieste passate dei client.

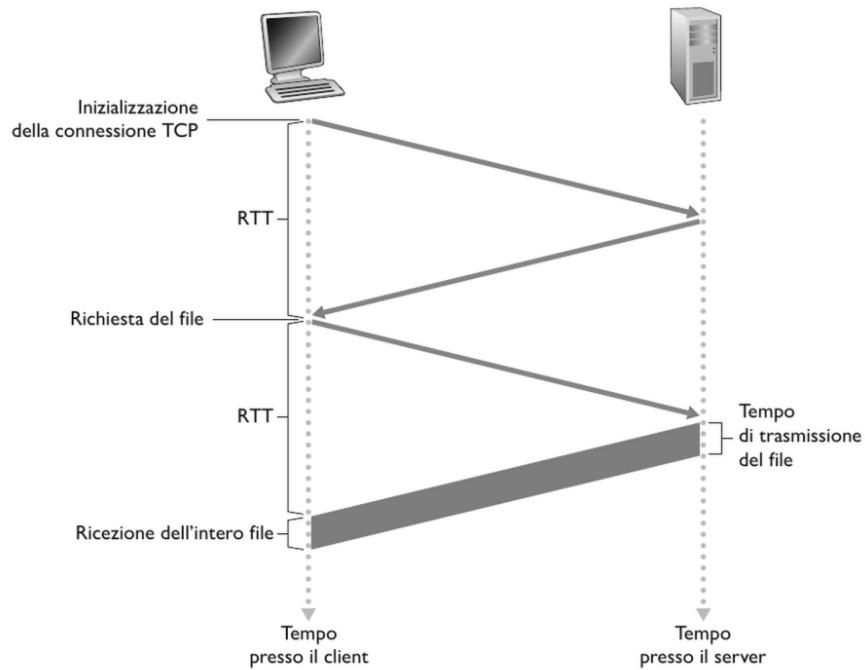
2.2.2 Connessioni persistenti e non persistenti

Si possono creare due tipi di connessioni:

- **non persistenti**: ogni coppia richiesta e risposta deve essere inviata su una connessione TCP *separata*. C'è un overhead nell'aprire una nuova connessione TCP per ogni coppia richiesta-risposta.
- **persistenti**: tutte le comunicazioni sono mandate sulla stessa connessione TCP (scelta di default per HTTP), riutilizzando la stessa connessione TCP e riducendo l'overhead.

HTTP con connessioni non persistenti

Nelle *connessioni non persistenti TCP* ogni connessione viene chiusa dopo l'invio dell'oggetto da parte del server, quindi ogni connessione trasporterà un solo messaggio di richiesta e solo uno di risposta. Esistono browser che possono aprire più connessioni TCP parallelamente per velocizzare. Il **round-trip time (RTT)** è il tempo impiegato da un pacchetto per viaggiare dal client al server e tornare al client. È una *misura* della latenza di rete. Include i ritardi di propagazione, di accodamento nei router e nei commutatori intermedi e di elaborazione del pacchetto.



Quando un utente clicca su un collegamento ipertestuale il browser inizializza una connessione TCP con il web server, iniziando così un **handshake a tre vie** (three-way handshake): il client invia un piccolo segmento TCP al server (SYN), il server manda una conferma sempre con un piccolo segmento TCP (SYN-ACK), e il client manda una conferma di ritorno al server (ACK). Questo handshake serve a stabilire la connessione TCP. Con queste prime due operazioni di handshake a tre vie calcoliamo il *RTT*. Il client ora invia un messaggio di richiesta HTTP insieme alla conferma di avvenuta ricezione (*acknowledgment - ACK*), a messaggio ricevuto dal server il server procede a inviare il file al client. Il tempo di risposta totale sarà di 2 RTT (un RTT per l'handshake TCP e un RTT per la richiesta-risposta HTTP) più il tempo di trasmissione del file dal server al client.

HTTP con connessioni persistenti

Nelle connessioni persistenti il server lascia aperta la connessione dopo la prima coppia di richiesta-risposta col client, tutte le altre coppie verranno trasmesse sulla stessa connessione, riutilizzando la stessa connessione TCP e riducendo l'overhead. Una delle caratteristiche di queste connessioni è il *pipelining*, la capacità di poter effettuare *più* richieste senza aspettare la risposta delle richieste in corso. La connessione si chiuderà dopo un lasso di tempo configurato in cui la connessione è stata inattiva.

2.2.3 Formato dei messaggi HTTP

Esistono due formati di messaggi HTTP, basati su testo, per richiesta e risposta.

Messaggio di richiesta HTTP

```
GET /somedir/page.html HTTP/1.1
Host: www.someurl.com
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

Questa è una richiesta HTTP, può avere un numero indefinito di righe, la riga fondamentale è la prima che è la **riga di richiesta**, le successive sono **righe di intestazione**.

La riga di richiesta ha 3 campi: metodo (GET, POST, DATA, PUT, DELETE), che specifica l'*azione* da eseguire sulla risorsa, l'URL e la versione di HTTP. **GET** è il metodo più utilizzato nel web, si usa per *recuperare* un oggetto tramite l'URL.

Notiamo la riga "Connection: close", con questa riga il browser comunica al server che non si deve occupare di connessioni persistenti, deve chiudere la connessione dopo aver inviato l'oggetto. Questo header è tipico delle connessioni non persistenti.

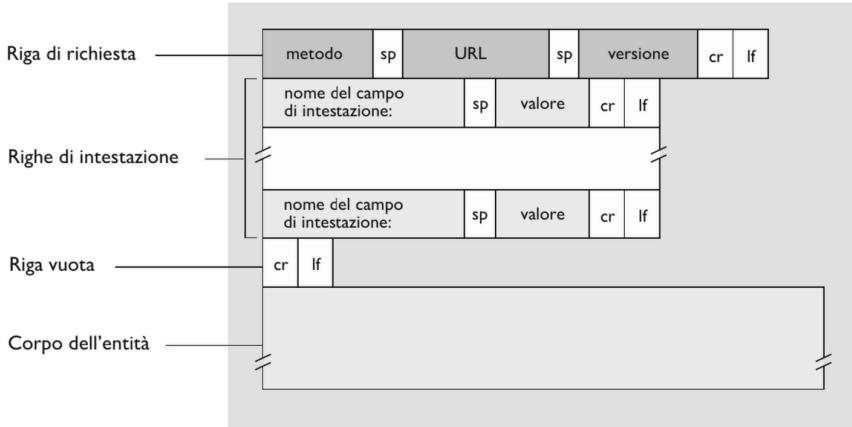
Alla fine della richiesta troviamo un *corpo* del messaggio, vuoto in caso di metodo GET, utilizzato in caso di metodo POST per inviare dati al server (es. dati di un form).

Il metodo **POST** viene utilizzato per *inviare* form compilati dall'utente al server, si può utilizzare anche il metodo GET per questo scopo ma includendo questi dati nell'URL della pagina richiesta.

Il metodo **HEAD** viene utilizzato dagli sviluppatori, è come il metodo *GET* ma si riceve solo la risposta HTTP, senza ricevere l'oggetto, quindi si recuperano i *metadata* di una risorsa.

Il metodo **PUT** viene utilizzato per caricare dal client dei file sul server.

Il metodo **DELETE** viene utilizzato per cancellare file sul server (spesso disabilitato per ragioni di sicurezza insieme al metodo PUT).



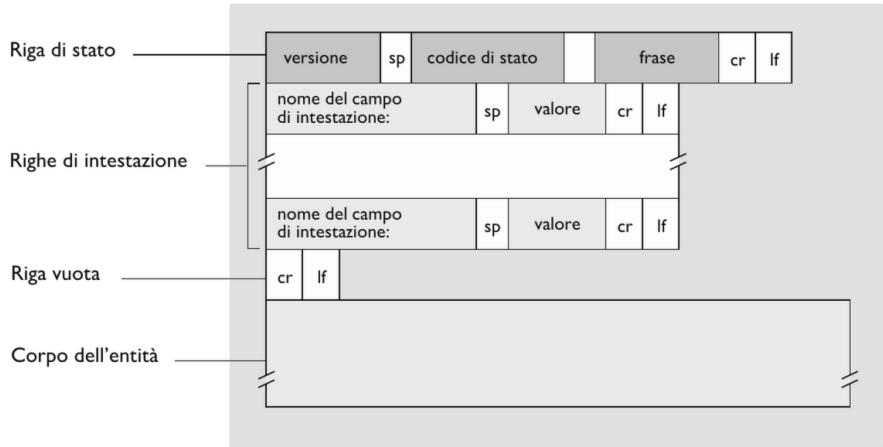
Messaggio di risposta HTTP

```
HTTP/1.1 200 OK Connection: close
Date: Thu, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
(data data data data ....)
```

Abbiamo: una **riga di stato** iniziale (contenente 3 campi, versione di protocollo, codice di stato HTTP che indica l'*esito* della richiesta e il corrispettivo messaggio), sei **righe di intestazione** e il **corpo dell'oggetto** finale (fulcro del messaggio, contiene l'oggetto richiesto).

Esistono vari codici di stato, questi i più comuni:

- **200 - OK:** la richiesta ha avuto successo e in risposta si invia l'informazione.
- **301 - Moved Permanently:** l'oggetto richiesto è stato trasferito in modo permanente; il nuovo URL è specificato nell'intestazione Location: del messaggio di risposta. Il client recupererà automaticamente il nuovo URL.
- **400 - Bad Request:** si tratta di un codice di errore generico che indica che la richiesta non è stata compresa dal server.
- **404 - Not Found:** il documento richiesto non esiste sul server.
- **505 - HTTP Version Not Supported:** il server non dispone della versione di protocollo HTTP richiesta.



2.2.4 Cookie

HTTP è un protocollo *stateless*, un elemento utile per i webserver sono i **cookie**, un identificativo per l'utente che mantiene le informazioni sul server, per esempio l'*autenticazione*, e che permette di *mantenere lo stato* tra più richieste HTTP.

È formato da 4 componenti, tra cui:

- Riga di intestazione nel messaggio di risposta HTTP (**Set-cookie**: numero identificativo), usata per *impostare* il cookie.
- Riga di intestazione nel messaggio di richiesta HTTP (**Cookie**: numero identificativo), usata per *inviare* il cookie al server.
- File cookie, mantenuto *localmente* sul sistema terminale dell'utente e gestito dal browser del client.
- Database sul webserver che mantiene l'identificativo dei cookie, usato per *associare* i cookie con i dati dell'utente.

I cookie possono anche essere usati per creare un livello di sessione utente al di sopra di HTTP che è privo di stato, permettendo di mantenere informazioni sull'utente durante la sua navigazione.

2.2.5 Web caching (proxy server)

Una **web cache**, cioè un **proxy server**, agisce come *intermediario* e gestisce le richieste HTTP al posto del web server effettivo. All'interno del proxy server troviamo delle *copie* dei siti già richiesti da altri web server, si trovano all'interno della sua cache.

Il meccanismo di funzionamento è:

1. Il browser stabilisce una connessione TCP con il proxy server e invia una richiesta HTTP per l'oggetto specificato.

2. Abbiamo due casi possibili ora:

- (a) Il proxy server ha l'oggetto richiesto nella propria cache, inoltra direttamente l'oggetto al browser che ne ha fatto richiesta.
- (b) Il proxy server non ha l'oggetto richiesto, apre una connessione TCP verso il server in cui risiede l'oggetto richiesto, effettua una richiesta HTTP al server e poi inoltra l'oggetto al browser.

3. Il proxy riceve l'oggetto e salva la copia sulla sua cache.

Notiamo che il proxy server è contemporaneamente sia *client* (quando richiede oggetti al server di origine) che *server* (quando fornisce oggetti al client). Il suo utilizzo riduce notevolmente i tempi di caricamento dei siti, il traffico di rete e il carico del server.

Uno dei problemi che si possono venire a creare è che l'oggetto richiesto al proxy server potrebbe non essere aggiornato, portando a visualizzare informazioni obsolete. Per ovviare a questo problema è stato implementato il **GET condizionale** (*conditional GET*), è come una richiesta GET HTTP tradizionale, in più si aggiunge una riga di intestazione **if-modified-since:**, che include un *timestamp*. Se il server riceve una richiesta GET condizionale, confronta il timestamp nell'header ‘*if-modified-since*’ con il timestamp dell'ultima modifica dell'oggetto. Se l'oggetto non è stato modificato dopo il timestamp specificato, il server risponde con un codice di stato **304 Not Modified** senza inviare l'oggetto. Se l'oggetto è stato modificato, il server risponde con un codice di stato **200 OK** e invia l'oggetto aggiornato.

2.3 Posta elettronica

La posta elettronica è un mezzo di comunicazione asincrono, ciò significa che non richiede che mittente e destinatario siano online contemporaneamente, tre componenti principali: gli **user agent** (o agenti utente), i **mail server** (server di posta) e il **protocollo SMTP (Simple Mail Transfer Protocol)**. L'*user agent* (un'applicazione usata per comporre, inviare e ricevere email) invia il messaggio al proprio *mail server* (il distributore del servizio che memorizza e inoltra i messaggi) che invierà la mail al *mail server* del destinatario. Componenti della posta elettronica:

- **casella di posta:** contenitore dei messaggi in arrivo, collocata in un *mail server*. Ogni utente ha una casella di posta *unica* su un mail server.
- **Coda di messaggi:** mail che devono arrivare al destinatario, memorizzate *temporaneamente* in attesa di consegna.
- **Protocollo SMTP (Simple Mail Transfer Protocol):** regola la comunicazione tra i *mail server* e tra gli *user agent* e i proprio *mail server*, usato per *trasferire* i messaggi. Principale protocollo a livello di applicazione per la posta elettronica, utilizza *TCP*. Quando un server invia posta a un altro, agisce come client SMTP; quando invece la riceve, funziona come

server SMTP. Il protocollo SMTP regola la comunicazione tra i *mail server* e tra gli *user agent* e i proprio *mail server*.

2.3.1 SMTP

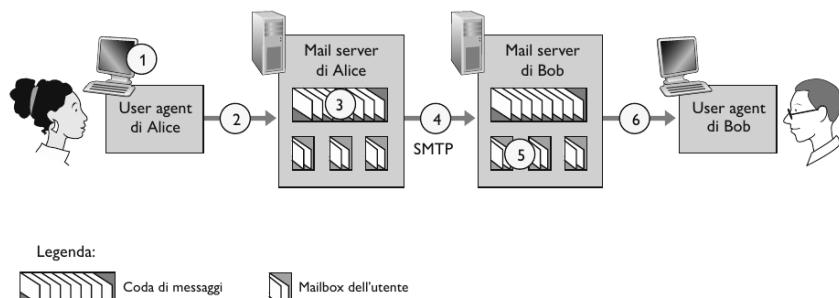
Il protocollo **SMTP** utilizza TCP, la porta standard è la 25, è un *protocollo con stato*, ciò significa che mantiene lo stato *tra più comandi* all'interno della stessa connessione, codificato in ASCII-7bit per i *messaggi di controllo*, mentre il corpo del messaggio può usare altre codifiche.

Si prevedono 3 fasi diverse:

- Prima fase: handshake, che include lo scambio di comandi specifici come HELO o EHLO.
- Seconda fase: scambio effettivo dei messaggi, che include comandi come MAIL FROM (per specificare il mittente), RCPT TO (per specificare il destinatario) e DATA (per inviare il corpo del messaggio).
- Terza fase: chiusura della connessione, che include il comando QUIT.

Scenario di funzionamento del protocollo

1. L'user agent mittente compone l'indirizzo mail del destinatario.
2. L'user agent invia il messaggio al mail server del mittente.
3. Il mail server mittente apre una connessione TCP con il mail server del destinatario sulla porta 25, agendo come client SMTP.
4. Il client SMTP (mail server mittente) invia il messaggio tramite la connessione TCP e lo colloca nella coda dei messaggi.
5. Il mail server del destinatario, agendo come server SMTP, riceve il messaggio e lo invia nella mail box.
6. L'user agent destinatario può ora leggere il messaggio.



Si utilizzano dei mail server poiché nel caso in cui le mail non possono essere consegnate in un preciso momento o ci sono errori, il mail server può fare vari tentativi (in base alla configurazione fatta) per consegnare il messaggio, utilizzando una *coda* per memorizzare i messaggi e ritentare la consegna in un secondo momento.

Oltre ai comandi di controllo, SMTP definisce anche il formato del messaggio email, che include header (es. From, To, Subject) e un corpo. La fine del messaggio email è indicata da un punto su una riga vuota. Per recuperare le email dalla casella di posta, gli user agent utilizzano protocolli come **POP3** (Post Office Protocol version 3), che permette di scaricare le email dal server e, optionalmente, di cancellarle dal server (download-and-delete) o di mantenerle (download-and-keep), o **IMAP** (Internet Message Access Protocol), che permette di gestire le mailbox direttamente sul server, consentendo di visualizzare le email, creare cartelle, spostare messaggi e altre operazioni senza dover scaricare tutte le email sul dispositivo locale.

2.4 DNS

Il **DNS** è un protocollo, ma in realtà è un sistema gerarchico e distribuito, *Domain Name System*. Il web è identificato da un nome simbolico, il nome dell'host, che è *leggibile dagli umani*, ma il modo univoco per identificare il server è il suo indirizzo IP (32 bit), che è *leggibile dalle macchine*. È un sistema distribuito, esistono vari server, organizzati in modo *gerarchico*, che conoscono l'associazione fra nome host e indirizzo IP. È un protocollo a livello di applicazione, i vari servizi che offre sono:

- Traduzione degli hostname in indirizzi IP, che è la funzione *principale* del DNS.
- Host aliasing, più nomi per lo stesso indirizzo IP (stesso host). Esiste un **nome canonico** della macchina che la identifica, esistono anche degli **alias** che sono altri nomi per la stessa macchina, usati per *convenienza* e *flessibilità*.
- Possibilità di gestire la posta elettronica in server diversi ma con lo stesso hostname (esempio mail@unipa.it e unipa.it), usando i record DNS per *mappare* i nomi di dominio ai mail server.
- Possibilità di distribuire il carico, più macchine possono gestire lo stesso server, ci saranno più server (e più indirizzi IP) per un unico hostname, e DNS può distribuire il traffico su *più server*.

2.4.1 Gestione gerarchica DNS

Abbiamo un sistema gerarchico, progettato per *distribuire il carico e migliorare la scalabilità*, più in alto abbiamo i **server radice**, sotto i **server TLD (top-level domain)** e infine i **server autoritativi (o di competenza)**.

I *server autoritativi* sono i server che posseggono le traduzioni, sono di società che posseggono host Internet, devono fornire i record DNS di pubblico dominio che mappano i nomi di tali host in indirizzi IP. Sono la *fente finale* dei record DNS per un dominio specifico.

I *server TLD* sono responsabili dei domini ad alto livello (.com, .co.uk, .it...), vengono gestiti da nazioni o da aziende e sono responsabili di *delegare l'autorità* ai server autoritativi.

I *server radice* sono il vertice della gerarchia e sono *essenziali* per il funzionamento del DNS. Sono 13 nel mondo (numero limitato), e verranno contattati da un **DNS locale**.

Il client contatterà sempre il server radice, che darà indicazioni su dove trovare i server TLD interessati che diranno al client qual è il server autoritativo responsabile per l'hostname scelto. In realtà, il client contatta un **DNS locale** (o *ricorsivo*) che poi effettua le query iterative o ricorsive.

2.4.2 DNS locale

Una macchina fuori dalla gerarchia, che si occuperà di fare da client per le comunicazioni tra il client reale e l'host radice, funzionando da intermediario. I DNS locali agiscono come *resolver ricorsivi*, effettuando le query per conto dei client.

Ogni ISP ha un **DNS locale** e lui opera da proxy, inoltre la query in una gerarchia di server DNS. Gli ISP forniscono DNS locali ai loro clienti. I DNS locali fanno *caching* dei record DNS per migliorare le prestazioni.

Approccio con **expiring date** (o TTL - Time-To-Live) per il mantenimento delle informazioni, usato per *invalidare* i record in cache.

Esempio di query iterativa

1. Il **client web richiedente** chiede al **client DNS proprio** (che è tipicamente parte del sistema operativo) di tradurre un hostname.
2. Il **client DNS** parla col **server DNS locale**.
3. Il **server DNS locale** richiede informazioni al **server DNS radice**.
4. Il **server DNS radice** fornisce un riferimento, al **server TLD**, al **server DNS locale**.
5. Il **server DNS locale** richiede informazioni al **server DNS TLD**.
6. Il **server DNS TLD** fornisce un riferimento, al **server di competenza**, al **server DNS locale**.
7. Il **server DNS locale** richiede informazioni al **server di competenza**.
8. Il **server di competenza** fornisce l'IP corrispondente dell'hostname al **server DNS locale**.

9. Il **server DNS locale**, ricevendo l'informazione dal **server di competenza**, la manda al **client dns richiedente** che trasferirà l'informazione al **client web richiedente** che potrà ora usare l'IP per connettersi al **server web richiesto**.

DNS Caching

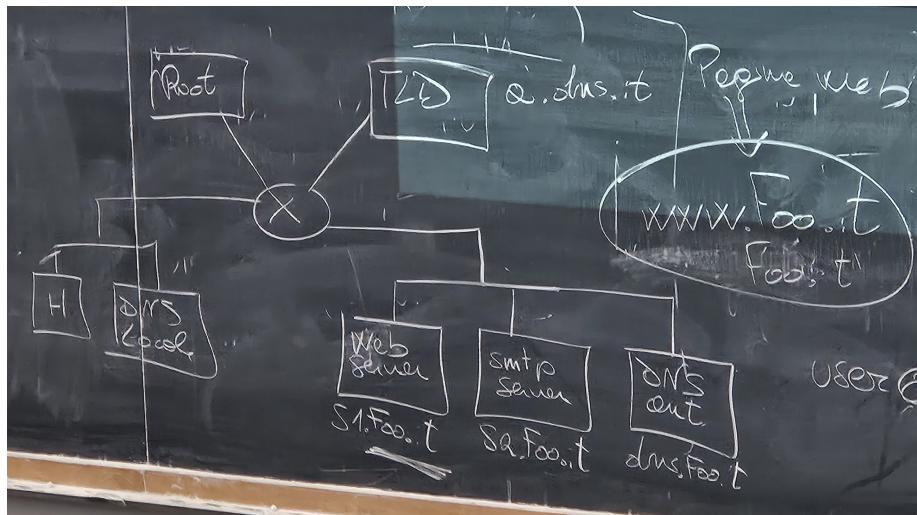
Per migliorare le prestazioni e ridurre il carico sui server DNS, i risultati delle query DNS vengono memorizzati in una **cache DNS**. Questa cache può essere presente sia nei *server DNS locali* che nei *client DNS*. Quando un client richiede la traduzione di un hostname, il DNS locale controlla prima la sua cache. Se il record DNS è presente nella cache e non è scaduto (in base al valore *ttl*), il DNS locale restituisce direttamente la risposta al client, evitando di dover effettuare una nuova query nella gerarchia DNS. Questo meccanismo di caching riduce significativamente il tempo di risposta e il traffico di rete.

2.4.3 Record DNS

Formato **RR (Record di risorsa)**: (name, value, type, ttl), dove *ttl* rappresenta il *time-to-live* del record. 4 tipi di type:

- **Type=A**: name = nome host, value = indirizzo *IPv4*.
- **Type=NS**: name = dominio, value = nome del server di competenza, usato per *delegare l'autorità* ad altri server DNS.
- **Type=CNAME (canonical name)**: name = *nome alias* di un nome *nome canonico*, value = *nome canonico*, usato per creare *alias* per gli hostname.
- **Type=MX**: value = nome del server di posta di *name*, name = nome server, usato per *intradare le email* al mail server appropriato.

Esempio popolazione DB server



DB del server Radice

Name	Value	Type
.it	a.dns.it	NS
a.dns.it	22.4.9.10	A

Il server radice punta al server TLD per il dominio ‘.it’.

DB del server TLD

Name	Value	Type
foo.it	dns.foo.it	NS
dns.foo.it	147.163.2.1	A

Il server TLD punta al server autoritativo per il dominio ‘foo.it’.

DB del server DNS autoritativo

Name	Value	Type
www.foo.it	s1.foo.it	CNAME
foo.it	s1.foo.it	CNAME
s1.foo.it	143.163.2.2	A
foo.it	s2.foo.it	MX
s2.foo.it	143.163.2.3	A

Il server autoritativo contiene gli indirizzi IP degli host nel dominio ‘foo.it’, incluso il mail server. Notiamo che i record CNAME creano alias per gli hostname e che i record MX specificano il mail server per il dominio.

3 Livello di trasporto

3.1 Introduzione e servizi a livello di trasporto

Strumento che instaura una **connessione logica** *end-to-end* tra i processi applicativi dei vari host, *non è un collegamento fisico*.

Il **livello di trasporto** è in esecuzione sui **sistemi terminali**, più precisamente nel loro *sistema operativo*, non sui router. Durante l'invio, al *mittente*, scinde i messaggi in vari **segmenti**, passandoli al livello inferiore (livello di rete), mentre durante la ricezione, al *destinatario*, riassembla il messaggio. Il **livello di trasporto** è un **daemon**, ovvero un *processo in background* che fornisce servizi, che mette in comunicazione logica i processi, usa i servizi del livello di rete, aspetta dal livello applicativo il messaggio da inviare e a chi inviarlo, fornendo servizi al *livello applicativo*.

Il **livello di rete** mette in comunicazione logica gli host, quindi una comunicazione *host-to-host*.

3.1.1 Protocolli utilizzati

I due protocolli utilizzati sono **TCP** e **UDP**.

Il protocollo TCP offre vari servizi, tra cui: controllo di congestione e controllo di flusso, questo garantirà un trasferimento dati *affidabile, ordinato e connection-oriented*, ma avrà ritardi nel trasporto a causa di un maggiore *overhead* e *potenziali ritardi*.

Il protocollo UDP essendo non orientato alla connessione è meno affidabile, non avendo nemmeno i controlli garantiti dal TCP, ma guadagna in velocità, non garantendo il corretto ordine di ricezione dei pacchetti e la ricezione di tutti i pacchetti, offrendo un trasferimento dati *non affidabile, non ordinato e connectionless*, con un *minore overhead* e una *trasmissione più veloce*.

Entrambi i protocolli non garantiscono servizi di *garanzia su ritardi* (visto che non possiamo prevedere il tempo di accodamento, il pacchetto potrebbe perdersi ed essere infinito bloccando tutta la connessione) e *garanzia su ampiezza di banda*, non fornendo garanzie *assolute* su ritardo o ampiezza di banda.

Il passaggio di consegna host-to-host a consegna process-to-process viene chiamato **multiplexing/demultiplexing a livello di trasporto** (*transport layer multiplexing/demultiplexing*).

3.2 Multiplexing e demultiplexing

L'operazione di **multiplexing** è l'operazione di invio, prende i dati da inviare dai vari processi applicativi, *raccogliendo dati da più processi applicativi*, incapsula il pacchetto con l'intestazione e invia il pacchetto.

L'operazione di **demultiplexing** è l'operazione di ricezione, prende i vari pacchetti, li ricompatta con le indicazioni dell'intestazione e li manda alla **socket** (un *endpoint logico* per la comunicazione tra processi, canale di comunicazione virtuale tra *livello applicazione* e *livello di trasporto*) corretta, utilizzando l'intestazione per *consegnare i dati al processo applicativo corretto*.

Nell'intestazione a livello di trasporto abbiamo bisogno di minimo: l'etichetta numerica della socket associata ai processi di mittente e destinatario, quindi il numero porta d'origine e di destinazione. Il resto dei campi, come i numeri di sequenza e i checksum (nel caso di TCP), dipende dal protocollo scelto.

I protocolli **standard** hanno delle porte precise, motivo per cui già sappiamo qual è la porta di destinazione, usate per *servizi ben noti*.

L'host usa gli indirizzi IP e i numeri di porta per inviare i vari segmenti, identificando *univocamente* un endpoint di comunicazione.

Quindi nel segmento a livello di trasporto avremo il campo con il **numero di porta di origine** e il campo col **numero di porta di destinazione**, i numeri di porta vanno da 0 a 65535. Da 0 a 1023 abbiamo le **porte note** (*well-known port number*) e sono riservati per i protocolli applicativi ben noti.

Multiplexing e demultiplexing non orientati alla connessione (UDP)

Il multiplexing e il demultiplexing UDP funzionano nel seguente modo:

- Le socket UDP sono create dalle applicazioni, con il livello di trasporto che assegna automaticamente un numero di porta o con l'applicazione che si lega a una porta specifica.
- Il lato client di un'applicazione UDP tipicamente lascia che il livello di trasporto assegna automaticamente il numero di porta, mentre il lato server si lega a una porta specifica.
- Durante il multiplexing, il livello di trasporto del mittente crea un segmento UDP che include i dati applicativi, i numeri di porta di origine e destinazione, e altri campi. Il segmento viene poi passato al livello di rete.
- Durante il demultiplexing, il livello di trasporto del destinatario esamina il numero di porta di destinazione del segmento UDP e lo consegna alla socket appropriata, e quindi al processo applicativo corretto.
- Una socket UDP è identificata univocamente da una coppia composta da indirizzo IP e numero di porta di destinazione. Segmenti con diversi indirizzi IP o numeri di porta di origine, ma con lo stesso indirizzo IP e numero di porta di destinazione, vengono diretti alla stessa socket.
- Il numero di porta di origine in un segmento UDP funge da "indirizzo di ritorno". Quando il destinatario vuole inviare una risposta, usa il numero di porta di origine del segmento ricevuto come numero di porta di destinazione del segmento di risposta.

Multiplexing e demultiplexing orientati alla connessione (TCP)

Il demultiplexing TCP si basa sull'identificazione delle socket TCP tramite una quaterna: indirizzo IP di origine, numero di porta di origine, indirizzo IP di destinazione e numero di porta di destinazione. Questa quaterna identifica

univocamente una connessione TCP. A differenza di UDP, segmenti TCP con indirizzi IP o numeri di porta di origine diversi vengono diretti a socket differenti, anche se l'indirizzo IP e la porta di destinazione sono uguali (eccetto per i segmenti di richiesta di connessione).

- Un server TCP ha una "socket di benvenuto" in ascolto su una porta specifica, usata per accettare nuove connessioni.
- Un client TCP crea una socket e invia un segmento con il flag SYN impostato a 1 per stabilire la connessione. Il sistema operativo del client sceglie un numero di porta di origine.
- Il server, ricevuta la richiesta di connessione, crea una nuova socket di connessione e memorizza la quaterna.
- Tutti i segmenti successivi con la stessa quaterna vengono diretti alla socket di connessione corrispondente.
- Un server può gestire più connessioni TCP concorrenti, ognuna identificata da una quaterna univoca.

Anche se più client usano la stessa porta di origine, il server può distinguere le connessioni grazie ai diversi indirizzi IP di origine.

3.3 Trasporto senza connessione: UDP

Un protocollo senza connessione, ciò significa che UDP non stabilisce una connessione prima di inviare i dati, i segmenti (NON SONO DATAGRAMMI) UDP (User Datagram Protocol) possono essere perduti o consegnati in ordine errato. UDP non garantisce la consegna, l'ordine o l'assenza di duplicati.

L'intestazione UDP è formata da numero porta origine, numero porta di destinazione, lunghezza in byte del segmento UDP con intestazione e il checksum (che è *opzionale* in IPv4, aggiunge bit alla fine per controllare se viene corrotto il pacchetto), tutti tasselli da 16 bit, totale di 8 Byte.

Source Port Number (2 bytes - 16 bit)	Destination Port Number (2 bytes - 16 bit)
Length (2 bytes - 16 bit)	Checksum (2 bytes - 16 bit)
Payload Data (If Any) (variable length)	

UDP viene usato nei protocolli **DNS** e **SNMP**, viene utilizzato nelle applicazioni multimediali, che sono *sensibili alla latenza* e tollerano la perdita di dati.

Checksum UDP

Serve a rilevare gli *errori di bit* nel segmento trasmesso, controlla se ci sono bit alternati nella checksum confrontandolo con il checksum prima del trasporto.

Operazioni del mittente:

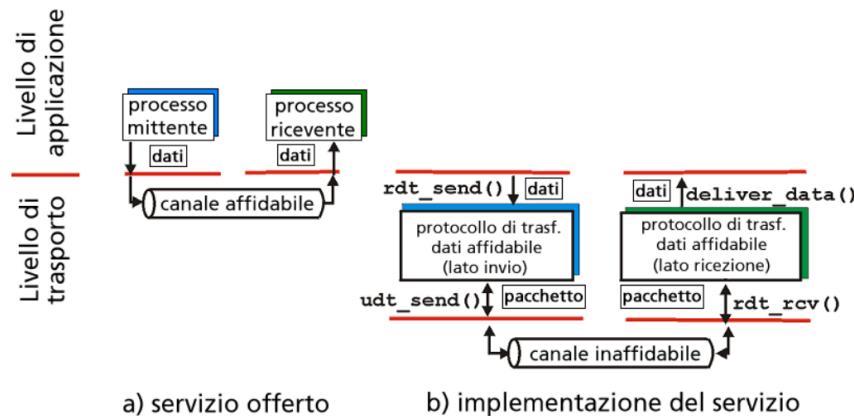
- Somma tutte le parole (tutti i campi presenti nel campo UDP, compreso di intestazione) tradotte in binari, usando l'*aritmetica del complemento a uno*. Nel caso in cui ci sia un riporto (17 bit) lo sommo al bit meno significativo, quindi il primo bit viene sommato al diciassettesimo bit, questa operazione è chiamata *end-around carry*, così che ora il pacchetto è lungo 16 bit.
- La checksum è il complemento a 1 della somma (gli 1 diventano 0 e gli 0 diventano 1).
- Il client calcola il checksum sull'*intero segmento UDP*, inclusi header e dati, e lo mette nell'intestazione.

L'host calcola il checksum e lo controlla con quello dentro l'intestazione, se rileva una discrepanza, ovvero se il checksum *ricalcolato* non corrisponde a quello ricevuto, scarta il pacchetto.

Errori multipli possono annullare bit corrotti, essendo somma binaria, se due bit opposti si corrompono il risultato non cambia. Il checksum non è quindi *infallibile* e può non rilevare alcuni errori.

3.4 Principi del trasferimento dati affidabile

Il servizio che offre il livello di trasporto è di un **canale affidabile**, ma l'implementazione del servizio utilizza un **canale inaffidabile** realizzato dal **livello di rete**. Il livello di trasporto deve realizzare il collegamento e rendere affidabile il canale messo a disposizione dal livello di rete.



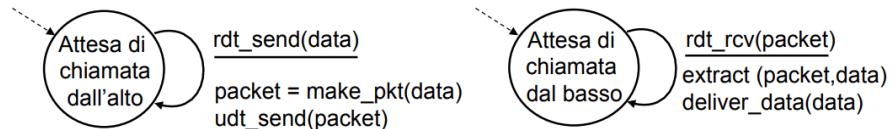
rdt: reliable data transfer - udt: unreliable data transfer

Il livello di trasporto, quindi, implementa un protocollo di trasferimento dati affidabile (rdt) per fornire un canale affidabile al livello applicativo, nonostante l'inaffidabilità del livello di rete. Il protocollo rdt utilizza funzioni come `rdt_send()` per inviare dati e `deliver_data()` per consegnare i dati al livello applicativo. Per comunicare attraverso il canale inaffidabile, il protocollo rdt utilizza funzioni come `udt_send()` per inviare pacchetti e `rdt_rcv()` per riceverli.

Rdt1.0: Mondo ideale

In un mondo ideale dove il livello rete offre un **canale affidabile**, il livello di trasporto esegue solo queste operazioni:

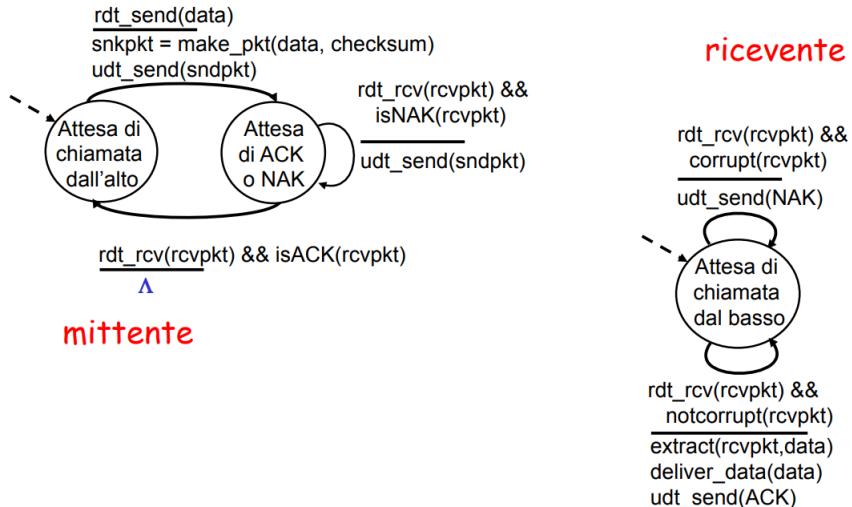
- L'host riceve i dati da inviare e il destinatario dal livello applicativo.
- Crea i pacchetti da inviare.
- Invia i dati al destinatario tramite il livello di rete.
- ...
- Il client riceve i dati dal livello di rete.
- Estrae i dati.
- Invia i dati al livello applicativo.



Il protocollo **rdt1.0** rappresenta un caso ideale in cui il livello di rete fornisce un canale di comunicazione perfetto. Il mittente, in attesa di una chiamata dal livello applicativo, invia i dati ricevuti tramite la funzione `rdt_send(data)`, creando un pacchetto con `make_pkt(data)` e inviandolo con `udt_send(packet)`. Il destinatario, in attesa di una chiamata dal livello di rete, riceve il pacchetto con `rdt_rcv(packet)`, estrae i dati con `extract(packet,data)` e li consegna al livello applicativo con `deliver_data(data)`.

Rdt2.0: canale con errori nei bit

Il *livello di trasporto* riceve i dati dal *livello applicativo*, crea i pacchetti e aggiunge all'intestazione la checksum a ogni pacchetto. Invia il pacchetto al destinatario che ricalcolerà il checksum e controllerà se è corretto. Nel caso in cui sia corretto, manderà un messaggio di **ACK** (conferma di ricezione) al mittente e manderà il pacchetto al *livello applicativo* del destinatario. Altrimenti, manderà un messaggio di **NAK** (notifica pacchetto corrotto) al mittente che dovrà rimandare lo stesso pacchetto prima di procedere a inviare i restanti.

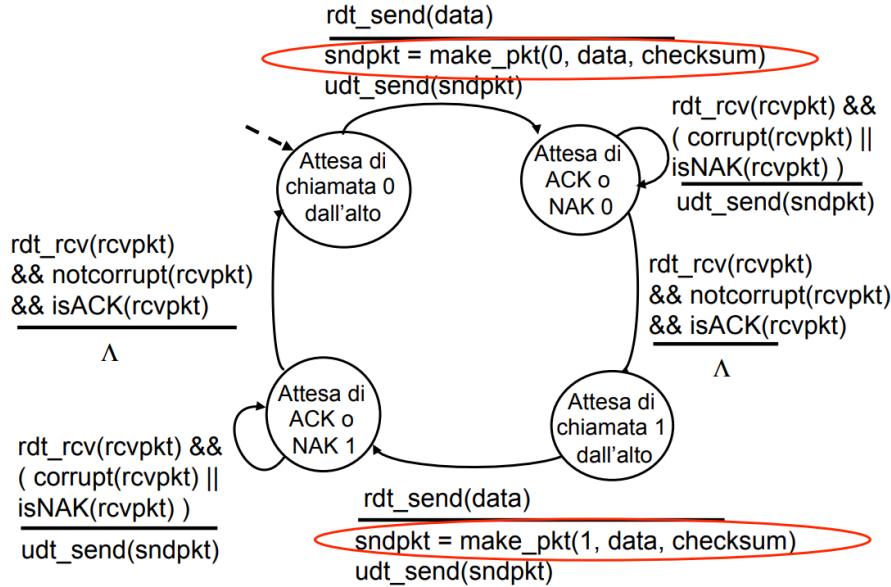


Il protocollo rdt2.0 introduce la gestione degli errori di bit. Il mittente, in attesa di una chiamata dal livello applicativo, crea un pacchetto con checksum tramite `make_pkt(data, checksum)` e lo invia con `udt_send(sndpkt)`. Il mittente poi attende un ACK o un NAK. Se riceve un NAK o un pacchetto corrotto (tramite `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`), ritrasmette il pacchetto. Se riceve un ACK (tramite `rdt_rcv(rcvpkt) && isACK(rcvpkt)`), torna allo stato di attesa. Il destinatario, in attesa di una chiamata dal livello di rete, riceve un pacchetto con `rdt_rcv(rcvpkt)`. Se il pacchetto è corrotto (tramite `corrupt(rcvpkt)`), invia un NAK con `udt_send(NAK)`. Se il pacchetto non è corrotto (tramite `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`), estrae i dati con `extract(rcvpkt,data)`, li consegna al livello applicativo con `deliver_data(data)` e invia un ACK con `udt_send(ACK)`.

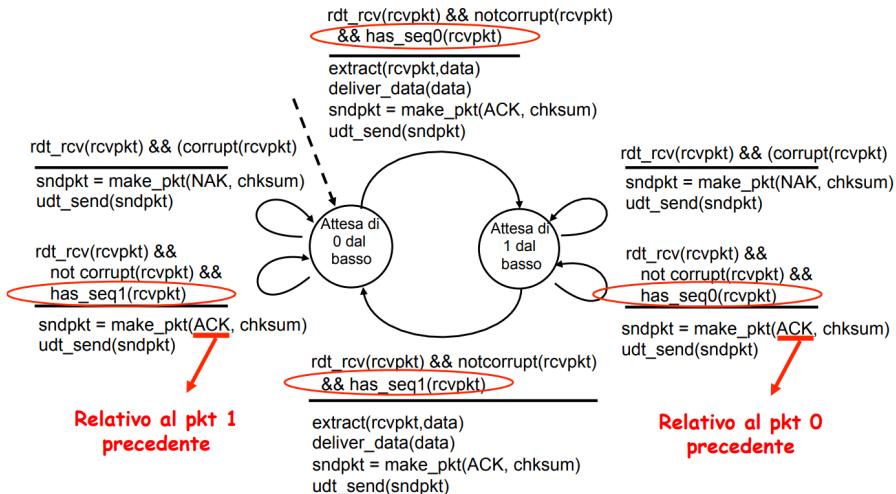
Rdt2.1: il mittente gestisce gli ACK o NAK alterati

Un problema di questo metodo è il caso in cui i pacchetti di *ACK* o *NAK* vengano corrotti, quindi il mittente non sa la risposta corretta del destinatario. Ritrasmettere è un'opzione, ma si possono avere dei **duplicati**. Dobbiamo risolvere il problema dei *duplicati*, aggiungiamo il **numero di sequenza** a ogni pacchetto. Il ricevente scarterà il pacchetto duplicato nel caso in cui il mittente rimandi lo stesso pacchetto anche avendo già mandato un *ACK*, avendo già memorizzato il *numero di sequenza*.

Schema automa a stati finiti mittente:



Schema automa a stati finiti ricevente:



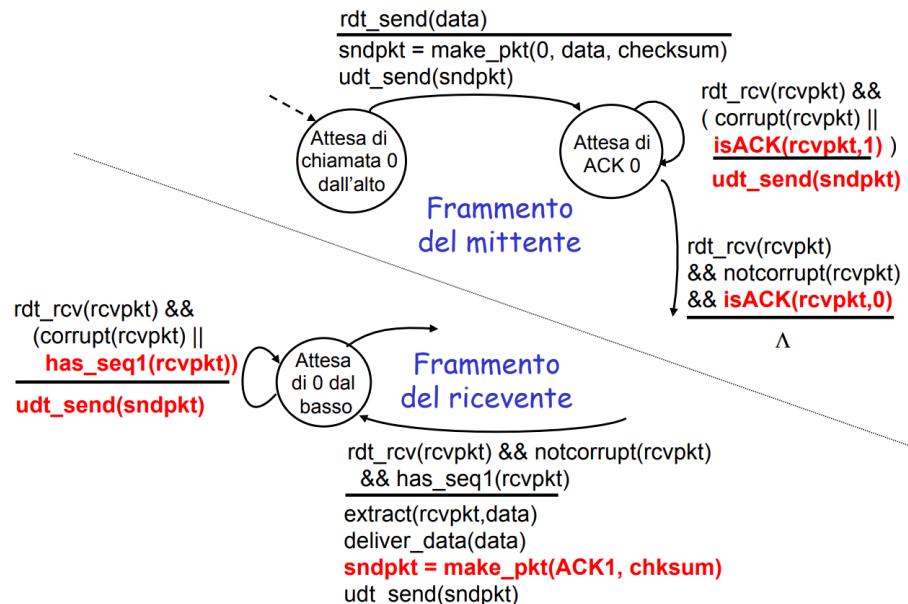
Il protocollo rdt2.1 introduce i numeri di sequenza per gestire i duplicati. Il mittente, in attesa di una chiamata dal livello applicativo, crea un pacchetto con numero di sequenza (0 o 1) e checksum tramite `make_pkt(seq, data, checksum)` e lo invia con `udt_send(sndpkt)`. Il mittente poi attende un ACK o un NAK.

Se riceve un NAK o un pacchetto corrotto (tramite `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))`), ritrasmette il pacchetto. Se riceve un ACK non corrotto (tramite `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`), passa all'altro stato di attesa.

Il destinatario, in attesa di una chiamata dal livello di rete, riceve un pacchetto con `rdt_rcv(rcvpkt)`. Se il pacchetto è corrotto (tramite `corrupt(rcvpkt)`), invia un NAK con `udt_send(NAK)`. Se il pacchetto non è corrotto, controlla il numero di sequenza. Se il numero di sequenza è quello atteso (tramite `has_seq0(rcvpkt)` o `has_seq1(rcvpkt)`), estrae i dati con `extract(rcvpkt,data)`, li consegna al livello applicativo con `deliver_data(data)` e invia un ACK con `udt_send(ACK)`, passando all'altro stato di attesa. Se il numero di sequenza non è quello atteso, scarta il pacchetto e invia un ACK per il pacchetto precedente.

Rdt2.2: Protocollo senza NAK

Si utilizza il numero di sequenza opposto al numero di sequenza del pacchetto che stiamo visualizzando come *NAK*. Se invio il pacchetto con **numero di sequenza = 0** e il destinatario non capisce, manderà come messaggio un **ACK con numero di sequenza 1**, dando al mittente un ACK con un altro numero di sequenza come *NAK*.



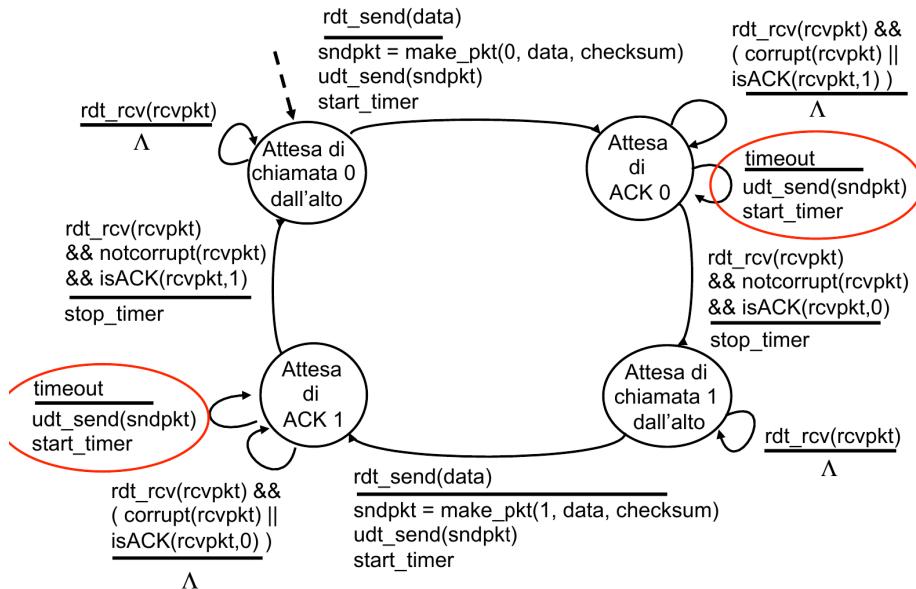
Il protocollo `rdt2.2` elimina l'uso esplicito dei NAK. Il mittente, in attesa di una chiamata dal livello applicativo, crea un pacchetto con numero di sequenza 0 e checksum tramite `make_pkt(0, data, checksum)` e lo invia con `udt_send(sndpkt)`.

Il mittente poi attende un ACK. Se riceve un pacchetto corrotto o un ACK con numero di sequenza 1 (tramite `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))`), ritrasmette il pacchetto. Se riceve un ACK non corrotto con numero di sequenza 0 (tramite `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)`), torna allo stato di attesa.

Il destinatario, in attesa di una chiamata dal livello di rete, riceve un pacchetto con `rdt_rcv(rcvpkt)`. Se il pacchetto è corrotto o ha il numero di sequenza errato (tramite `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))` quando è in attesa di 0 o tramite `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq0(rcvpkt))` quando è in attesa di 1), invia un ACK con il numero di sequenza opposto con `udt_send(sndpkt)`. Se il pacchetto non è corrotto e ha il numero di sequenza atteso (tramite `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)` quando è in attesa di 0 o tramite `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)` quando è in attesa di 1), estrae i dati con `extract(rcvpkt,data)`, li consegna al livello applicativo con `deliver_data(data)` e invia un ACK con il numero di sequenza atteso con `udt_send(sndpkt)`, passando all'altro stato di attesa.

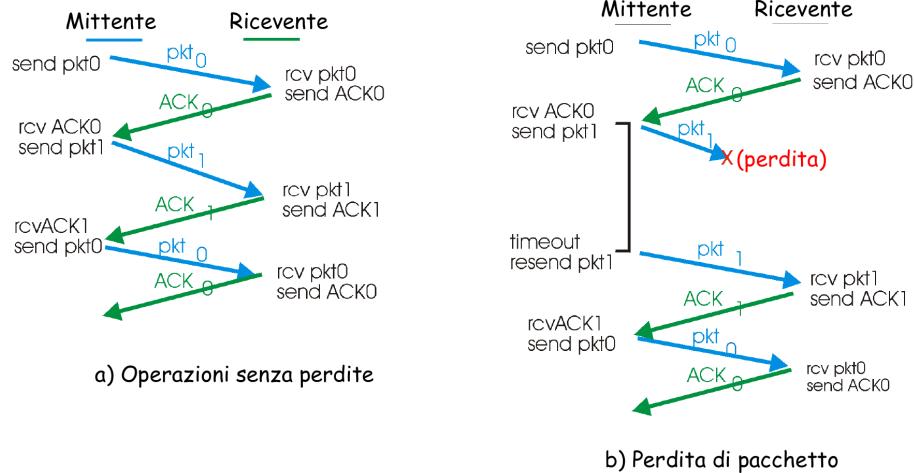
RDT3.0: canali con errori e perdite

Si aggiunge un timer di attesa per la ricezione di un **ACK**, così in caso di pacchetto perso il mittente ritrasmetterà il pacchetto. Nel caso in cui sia solo il ritardo, il mittente invierà il pacchetto, ma il duplicato verrà gestito tramite i numeri di sequenza.

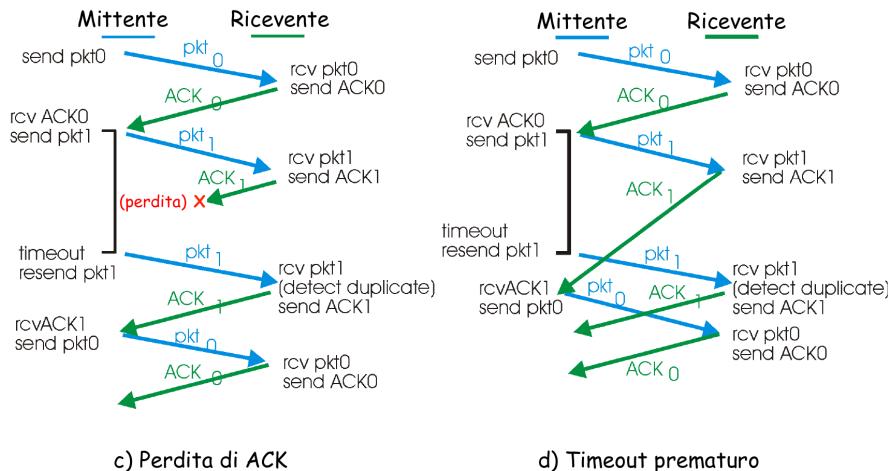


l'automa a stati finiti del ricevente è uguale a quello del **RTD2.2**

RDT3.0: Perdita di pacchetto



RDT3.0: Perdita di ACK



Il protocollo rdt3.0 introduce un timer per gestire la perdita di pacchetti. Il mittente, in attesa di una chiamata dal livello applicativo, crea un pacchetto con numero di sequenza (0 o 1) e checksum tramite `make_pkt(0, data, checksum)` o `make_pkt(1, data, checksum)`, lo invia con `udt_send(sndpkt)` e avvia un timer con `start_timer`. Il mittente poi attende un ACK. Se riceve un pacchetto corrotto o un ACK con numero di sequenza errato (tramite `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))` o `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 0))`), ritrasmette il pacchetto e riavvia il timer. Se riceve un ACK non corrotto con il numero di sequenza corretto (tramite `rdt_rcv(rcvpkt)`)

`&& notcorrupt(rcvpkt) && isACK(rcvpkt, 0) o rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)),` ferma il timer con `stop_timer` e passa all'altro stato di attesa. Se il timer scade (tramite `timeout`), ritrasmette il pacchetto e riavvia il timer. Il ricevente ha lo stesso automa a stati finiti del protocollo `rdt2.2`.

Il **RDT3.0** utilizza un algoritmo **STOP and WAIT** ma è molto lento, utilizziamo le **pipeline** per velocizzare il sistema.

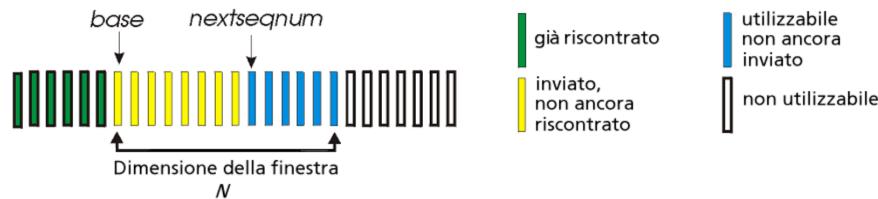
3.4.1 Protocolli con pipeline

Utilizzeremo due tipi di meccanismi, sono opposti come filosofia.

3.4.2 Go-Back-N

- Il mittente può avere fino a N pacchetti senza ACK in pipeline (finestra scorrevole).
- Il ricevente ha un buffer di dimensione 1 e mantiene solo il numero di sequenza atteso.
- Il ricevente invia solo **ACK cumulativi**, non dà l'ACK di un pacchetto se c'è un gap.
- Il mittente ha un singolo timer per il pacchetto base della finestra (il più vecchio non riscontrato). Quando arriva un ACK per questo pacchetto, il timer viene riavviato per il nuovo pacchetto base.

L'ACK sarà con il numero di sequenza dell'ultimo pacchetto arrivato correttamente e in ordine. Il mittente può continuare a mandare altri pacchetti, ma quelli con numero di sequenza superiore a quello atteso vengono scartati dal ricevitore, che continua a mandare ACK per l'ultimo pacchetto ricevuto correttamente in ordine.



La finestra contiene N pacchetti inviati di cui ancora non è arrivato un riscontro e scorre in avanti quando vengono ricevuti gli ACK. `nextseqnum`: prossimo pacchetto da inviare. Il protocollo Go-Back-N (GBN) permette al mittente di inviare fino a N pacchetti senza attendere un ACK per ognuno. Il ricevente invia solo ACK cumulativi, confermando la ricezione di tutti i pacchetti fino a un certo numero di sequenza. Se il mittente non riceve un ACK entro un certo tempo, ritrasmette tutti i pacchetti non ancora confermati. Come si vede nell'immagine, il mittente mantiene una finestra di N pacchetti inviati ma non

ancora confermati, e il puntatore *nextseqnum* indica il prossimo pacchetto da inviare.

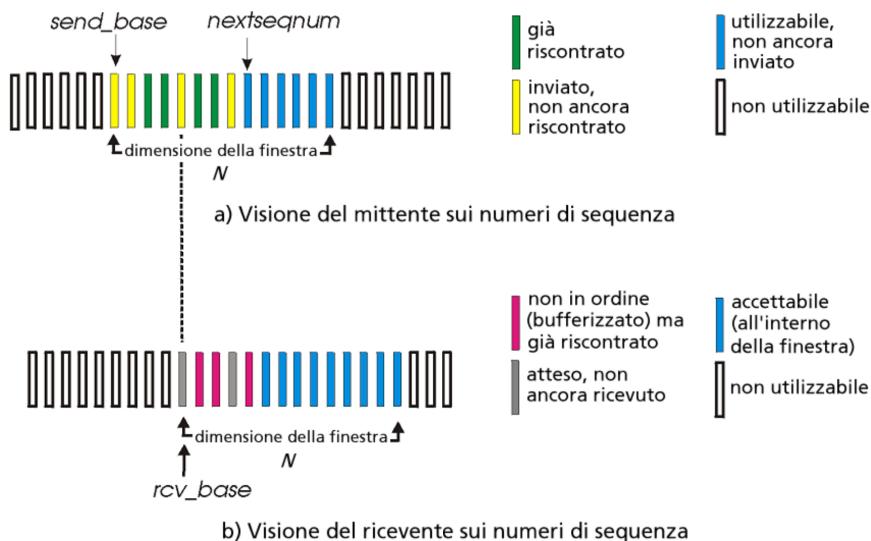
Go-Back-N: Funzionamento passo per passo

1. **Invio Pacchetti:** Il mittente invia pacchetti con numeri di sequenza crescenti (0, 1, 2, ...), fino a raggiungere la dimensione massima della finestra N .
2. **Ricezione ACK Cumulativi:** Il ricevente invia ACK cumulativi, confermando la ricezione di tutti i pacchetti fino a un certo numero di sequenza. Ad esempio, un ACK con numero di sequenza 3 indica che i pacchetti 0, 1, 2 e 3 sono stati ricevuti correttamente.
3. **Gestione Buffer Ricevitore:** Il ricevitore mantiene un singolo numero di sequenza atteso. Tutti i pacchetti con numero di sequenza superiore vengono immediatamente scartati.
4. **Timeout e Ritrasmissione:** Il mittente mantiene un timer per il pacchetto con il numero di sequenza più basso non ancora confermato. Se il timer scade, il mittente ritrasmette tutti i pacchetti non ancora confermati, a partire dal pacchetto con il numero di sequenza più basso.
5. **Esempio di funzionamento senza perdite:**
 - (a) Il mittente invia i pacchetti 0, 1 e 2.
 - (b) Il ricevente riceve i pacchetti 0, 1 e 2 e invia un ACK con numero di sequenza 2.
 - (c) La finestra del mittente scorre in avanti e il mittente invia i pacchetti 3, 4 e 5.
6. **Esempio di funzionamento con perdita di pacchetto:**
 - (a) Il mittente invia i pacchetti 0, 1 e 2.
 - (b) Il pacchetto 1 viene perso.
 - (c) Il ricevente riceve il pacchetto 0 e poi il pacchetto 2, ma scarta il pacchetto 2 perché attende il pacchetto 1. Continua a inviare ACK per il pacchetto 0.
 - (d) Il timer del mittente per il pacchetto 0 scade.
 - (e) Il mittente ritrasmette i pacchetti 0, 1 e 2.
 - (f) Il ricevente riceve i pacchetti 0, 1 e 2 e invia un ACK con numero di sequenza 2.

3.4.3 Selective Repeat

Il protocollo Selective Repeat (SR) permette al ricevente di inviare ACK per ogni pacchetto ricevuto correttamente, anche se non sono in ordine. A differenza del Go-Back-N, il ricevitore mantiene un buffer per i pacchetti fuori sequenza. La dimensione della finestra deve essere minore o uguale a $N/2$, dove N è la dimensione dello spazio dei numeri di sequenza, per evitare ambiguità nella numerazione dei pacchetti.

Il mittente mantiene un timer per ogni pacchetto inviato e ritrasmette solo i pacchetti per cui non ha ricevuto un ACK entro il timeout. Come si vede nell'immagine, il ricevente può accettare e bufferizzare pacchetti anche se non sono in ordine, e il mittente ritrasmette solo i pacchetti persi.



Selective Repeat: funzionamento passo per passo

1. **Invio Pacchetti:** Il mittente invia pacchetti con numeri di sequenza crescenti (0, 1, 2, ...), fino a raggiungere la dimensione massima della finestra ($\leq N/2$).
2. **Ricezione ACK Selettivi:** Il ricevente invia ACK selettivi, confermando la ricezione di ogni singolo pacchetto.
3. **Gestione Buffer Ricevitore:**
 - Il ricevitore mantiene un buffer per i pacchetti fuori sequenza
 - I pacchetti vengono bufferizzati fino a quando possono essere consegnati in ordine al livello superiore
 - La consegna al livello superiore avviene solo quando tutti i pacchetti precedenti sono stati ricevuti

4. Timeout e Ritrasmissione Selettiva: Il mittente mantiene un timer per ogni pacchetto inviato. Se il timer di un pacchetto scade, il mittente ritrasmette solo quel pacchetto.

5. Esempio di funzionamento senza perdite:

- (a) Il mittente invia i pacchetti 0, 1 e 2.
- (b) Il ricevente riceve i pacchetti 0, 1 e 2 e invia ACK per ognuno di essi.
- (c) I pacchetti vengono consegnati in ordine al livello superiore.
- (d) Il mittente riceve gli ACK e invia i pacchetti 3, 4 e 5.

6. Esempio di Funzionamento con Perdita di Pacchetto:

- (a) Il mittente invia i pacchetti 0, 1 e 2 (assumiamo finestra di dimensione 3).
- (b) Il pacchetto 1 viene perso.
- (c) Il ricevente riceve i pacchetti 0 e 2:
 - Invia ACK per 0 e ACK per 2
 - Bufferizza il pacchetto 2 (fuori sequenza)
- (d) Il mittente riceve ACK 0:
 - La sua finestra scorre in avanti di una posizione
 - Avendo spazio nella finestra (che ora contiene 1[non ACKed], 2, -), invia il pacchetto 3
 - Il pacchetto 1 rimane nella finestra come non riscontrato
- (e) Il mittente riceve ACK 2:
 - Marca il pacchetto 2 come riscontrato nella finestra
 - La finestra ora contiene (1[non ACKed], 2[ACKed], 3)
 - Avendo spazio nella finestra per un nuovo pacchetto, invia il pacchetto 4
- (f) Il timer del pacchetto 1 scade:
 - Il mittente ritrasmette solo il pacchetto 1
 - Continua a gestire normalmente il resto della finestra (pacchetti 2, 3, 4)
- (g) Il ricevente riceve il pacchetto 1:
 - Invia un ACK per 1
 - Ora può consegnare in ordine i pacchetti 0, 1 e 2 al livello superiore
- (h) Il mittente riceve ACK 1:
 - La finestra scorre ancora (ora contiene 3, 4, -)
 - Invia il pacchetto 5 nello spazio disponibile

THROUGHPUT: tasso di occupazione medio con cui i dati vengono trasmessi sul collegamento, rapporto tra tutti i dati trasmessi (anche più volte).

GOODPUT: tasso con cui il livello applicativo di destinazione vede arrivare i dati utili, rapporto tra i dati utili e il tempo di trasmissione.

Il *throughput* è sempre maggiore del *goodput*, solo nel caso ideale saranno uguali.

3.5 TCP: trasporto orientato alla connessione

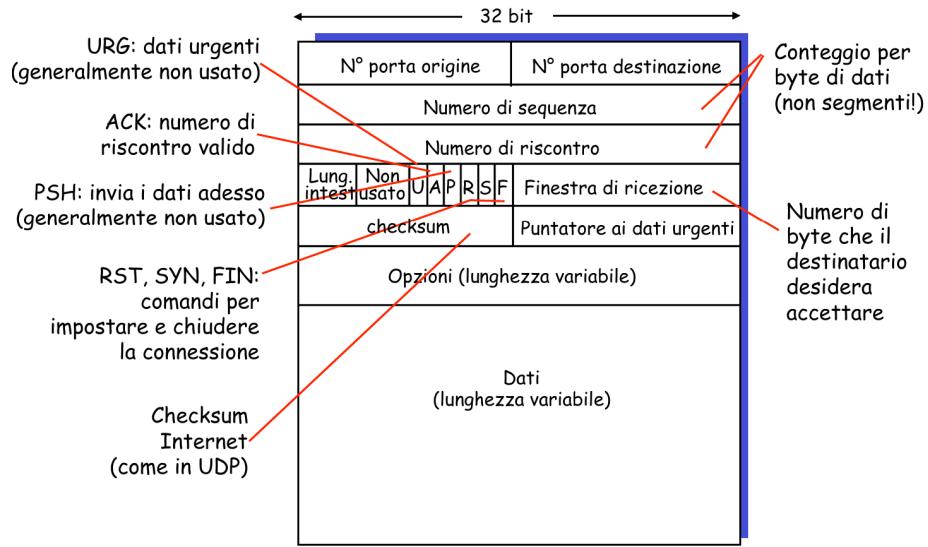
È un tipo di connessione **punto-punto**, ovvero una connessione *logica* tra mittente e destinatario, **full duplex**, abbiamo un flusso di dati bidirezionale, ovvero i dati possono fluire in *entrambe le direzioni simultaneamente*, e **orientato alla connessione**, ovvero TCP usa un *handshaking a tre vie* per stabilire la connessione, ha un flusso di byte affidabile, ovvero TCP garantisce la consegna dei dati nell'ordine corretto. I dati vengono mandati in **pipeline**, ovvero TCP permette di avere *più segmenti* in transito simultaneamente, attraverso un meccanismo *sliding window*, usato per il *controllo di flusso* e il *controllo di congestione*, abbiamo un **buffer d'invio** e un **buffer di ricezione**, usati per *memorizzare i dati* prima della trasmissione e dopo la ricezione, così che i pacchetti che arrivano fuori ordine vengono conservati e riordinati successivamente.

3.5.1 Struttura dei segmenti

L'**OVERHEAD** minimo del *TCP* è di **20 Byte**, ma l'header TCP ha una lunghezza *variabile* a causa del campo opzioni.

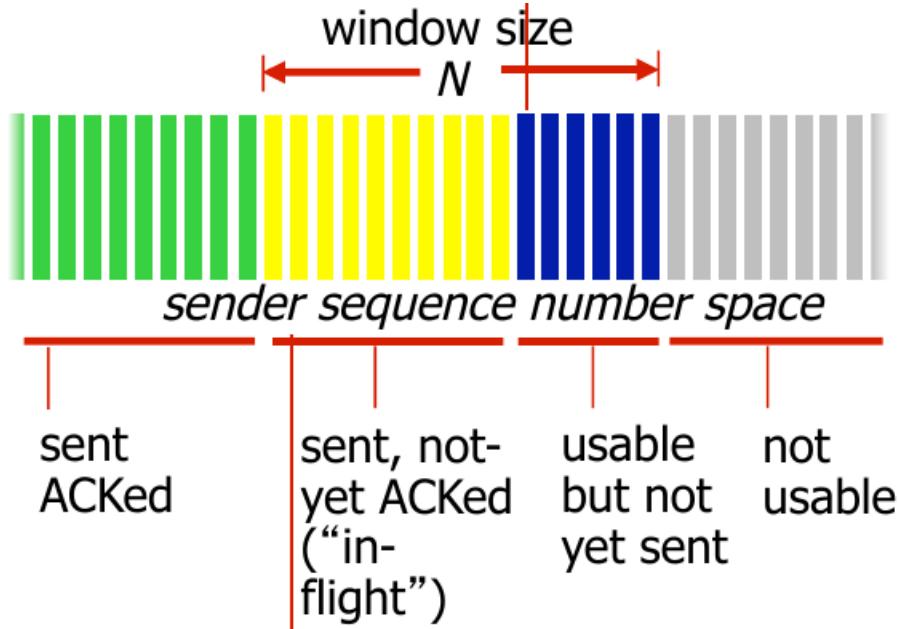
- Prima riga - uguale alla struttura del protocollo *UDP*, serve per il *multiplexing e demultiplexing*.
- Seconda riga - **Numero di sequenza**: Posizione del primo Byte all'interno del payload (= MSS, Maximum Segment Size).
- Terza riga - **Numero di riscontro**: Il numero di riscontro sarà il *prossimo numero di sequenza atteso*, ovvero il primo byte del segmento successivo a quello arrivato.
- Quarta riga - **BIT DI FLAG**: U (dati urgenti), A (**ACK**), P (PUSH), R (RST), S (SYN), F (FIN), gli ultimi tre sono comandi per impostare e chiudere la connessione, questi bit sono usati per la *gestione della connessione* e il *controllo di flusso*.
- Quarta riga - **Lunghezza intestazione**: nel protocollo UDP è sempre 8 byte, qui no, verrà specificato pacchetto per pacchetto, indicando la lunghezza dell'header in *parole di 32 bit*.
- Quarta riga - **Finestra di ricezione**: Da non confondere con la *sliding window*, dice quanto spazio si ha a disposizione per ricevere i dati, usata per il *controllo di flusso* e indica la *quantità di dati* che il ricevitore è disposto ad accettare.

- Quinta riga - **checksum**: grandezza di 16 bit, calcolata come la checksum dell'UDP, ovvero sull'intero segmento *TCP*, inclusi header e dati.
- Quinta riga - **Puntatore ai dati urgenti**: Nel caso in cui la flag U sia attiva ci sarà il puntatore alla memoria per quei dati, indicando la *fine dei dati urgenti*.
- Sesta riga - **Opzioni**: varie ed eventuali, usate per *funzionalità opzionali* come il timestamping.



3.5.2 Gestione numeri di sequenza e riscontro del TCP

Il *numero di sequenza* è il numero di byte nel flusso di dati che corrisponde al primo byte di dati nel segmento, dipende dal mittente e da come gestisce la memoria del proprio **buffer di invio**. Il *numero di riscontro* utilizza un *ACK cumulativo*, sarà il numero del prossimo byte che il destinatario si aspetta di ricevere in ordine, quindi sarà il primo byte del segmento che vorrà ricevere, quindi del successivo all'ultimo correttamente ricevuto e immagazzinato, dipende dal destinatario e da come gestisce la memoria del proprio **buffer di ricezione**.



3.5.3 Gestione del timer nel TCP

Il problema principale è stimare correttamente la durata del timer utilizzando la **media mobile esponenziale ponderata**. L'obiettivo è impostare un timeout *ragionevole*, che non sia troppo corto (causando ritrasmissioni inutili) né troppo lungo (causando ritardi). La media mobile esponenziale ponderata è una tecnica che dà *più peso ai campioni recenti* del RTT. La formula è la seguente:

$$\text{EstimatedRTT}(t) = (1 - \alpha) \cdot \text{EstimatedRTT}(t - 1) + \alpha \cdot \text{SampleRTT}(t)$$

dove solitamente $\alpha = 0.125$. In questa formula, ‘EstimatedRTT(t-1)’ è il RTT stimato *precedente* e ‘SampleRTT(t)’ è il RTT misurato *corrente*. Bisogna calcolare la *deviazione standard* del RTT:

$$\text{DevRTT}(t) = (1 - \beta) \cdot \text{DevRTT}(t - 1) + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

dove solitamente $\beta = 0.25$. In questa formula, ‘DevRTT(t-1)’ è la deviazione stimata *precedente* e ‘|SampleRTT - EstimatedRTT|’ è la *differenza assoluta* tra il RTT misurato e quello stimato.

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

Il timeout è calcolato aggiungendo un *multiplo della deviazione* al RTT stimato.

3.5.4 Trasferimento dati affidabile del TCP

Eventi del mittente

Il *livello di trasporto* riceve i dati dal *livello applicativo*, crea i pacchetti e aggiunge all'intestazione la checksum a ogni pacchetto. Il mittente mantiene un **buffer di invio** per memorizzare i dati non ancora confermati. Invia il pacchetto al destinatario che ricalcolerà il checksum e controllerà se è corretto. In caso di timeout o di *ACK* duplicati, che indicano una *potenziale perdita* di un segmento, il protocollo TCP ritrasmetterà il pacchetto, riavviando il timer. Il timeout è associato al *segmento non ancora riscontrato più vecchio*. Controlla gli *ACK* ricevuti, aggiorno ciò che è stato ricevuto e avvio il time nel caso in cui dovesse completare segmenti già inviati.

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (sempre) {
    switch (evento)

        evento: i dati ricevuti dall'applicazione superiore
            creano il segmento TCP con numero di sequenza
            NextSeqNum
            if (il timer attualmente non funziona)
                avvia il timer
            passa il segmento a IP
            NextSeqNum = NextSeqNum + lunghezza(dati)

        evento: timeout del timer
            ritrasmetti il segmento non ancora riscontrato con
            il piu' piccolo numero di sequenza

        evento: ACK ricevuto con valore del campo ACK y
            if (y > SendBase) {
                SendBase = y
                if (esistono timer non attualmente riscontrati)
                    avvia timer
            }
    } /* fine loop */
```

Quando si ritrasmette il pacchetto, il protocollo raddoppia il tempo di timeout al riavvio, nel caso di altra ritrasmissione raddoppierà il valore dell'ultimo timeout usato (quindi già raddoppiato). Questo è un meccanismo di *evitamento della congestione*.

Algoritmo della ritrasmissione rapida

Quando si arriva a 3 *ACK* duplicati si effettua una ritrasmissione rapida, prima che scada il timer. La ritrasmissione rapida è un meccanismo di *controllo della*

congestione che aiuta a evitare ritardi inutili, il timer non viene spento. Nel caso in cui nel frattempo sono arrivati i pacchetti successivi, il destinatario, al momento in cui riceverà il pacchetto che era andato perso e per cui ha mandato 3 ACK duplicati, invierà l'ACK del successivo dell'ultimo pacchetto ricevuto (sono stati bufferizzati nel mentre che aspettava quel pacchetto).

```

evento: ACK ricevuto, con valore del campo ACK pari a
        y
        if (y > SendBase) {
            SendBase = y
            if (esistono attualmente segmenti non ancora
                riscontrati)
                avvia il timer
        } else {
            incrementa il numero di ACK duplicati ricevuti per
            y
            if (numero di ACK duplicati ricevuti per y = 3) {
                rispedisci il segmento con numero di sequenza y
            }
        }
    
```

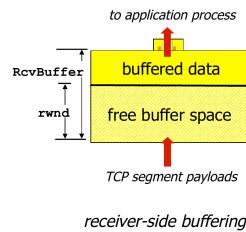
3.5.5 Controllo del flusso

Ricordiamoci che il protocollo *TCP* inizializza delle zone di memoria per il *buffer di ricezione* e *buffer di invio*, che si trovano nei *sistemi terminali*, il mittente non deve sovraccaricare il buffer del destinatario.

Il controllo di flusso serve a *prevenire che il mittente sovraccarichi il ricevitore*.

Mittente e destinatario comunicano continuamente quanto spazio hanno libero nei vari buffer.

Il valore di **RcvWindow** verrà inserito all'interno dei segmenti, il mittente usa il valore di 'RcvWindow' per *limitare la quantità di dati non riscontrati* che invia, così che non vengano inviati dati che verranno sicuramente persi. Il campo 'RcvWindow' indica la *quantità di spazio libero* nel buffer del ricevitore. **RcvBuffer** funziona per la creazione della socket, è una *system call* usata per impostare la dimensione del buffer di ricezione.



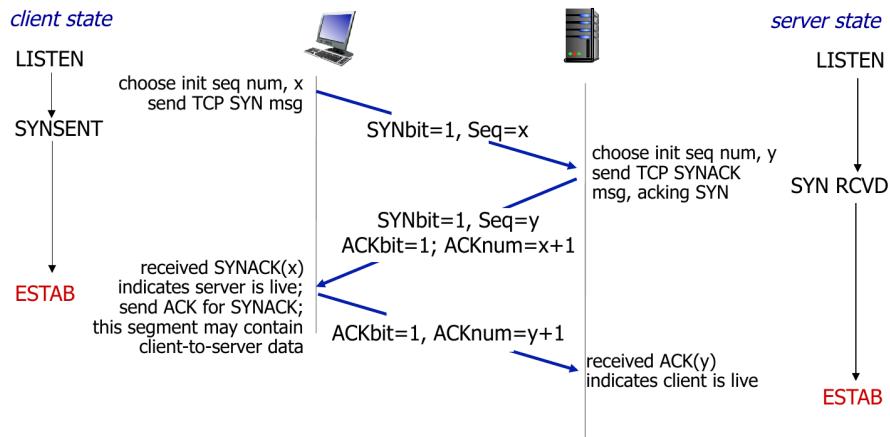
L'immagine mostra il buffer di ricezione lato destinatario. Il buffer è diviso in due parti: la parte superiore, dove sono memorizzati i dati ricevuti ma non

ancora consegnati all'applicazione, e la parte inferiore, che rappresenta lo spazio libero nel buffer. Il valore *RcvWindow* (indicato come *rwnd* nell'immagine) indica la dimensione dello spazio libero nel buffer di ricezione, e viene comunicato al mittente per regolare la velocità di trasmissione.

Gestione della connessione: Handshake a tre vie

Si stabilisce una connessione tra mittente e destinatario, si mettono d'accordo e inizializzano dei dati come *numero di sequenza* e i *buffer*. L'inizializzazione della connessione avviene tramite l'**Handshake a tre vie**:

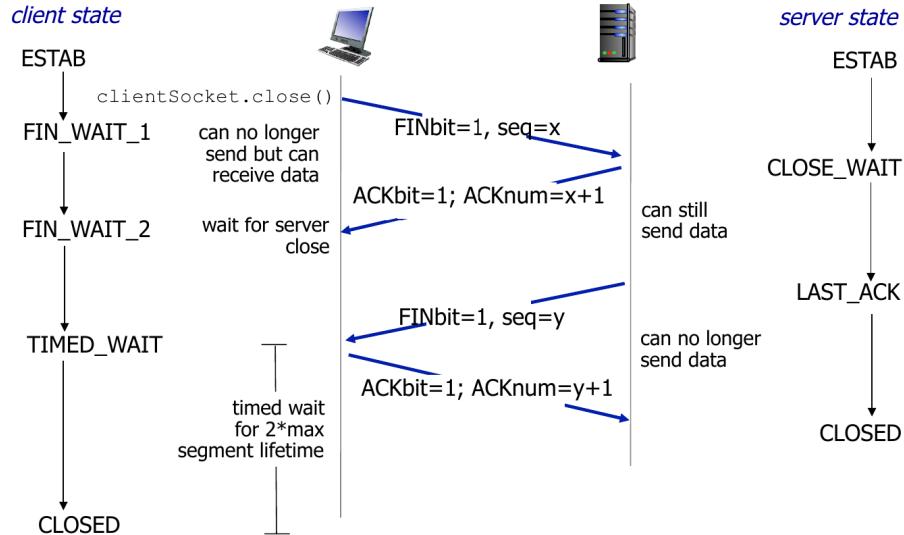
- Passo 1: il client invia un segmento SYN al server
specificando il numero di sequenza iniziale, scelto in modo casuale
nessun dato
- Passo 2: il server riceve SYN e risponde con un segmento SYNACK
il server alloca i buffer
specifica il numero di sequenza iniziale del server, scelto in modo casuale
il segmento SYNACK ha il flag SYN e ACK impostati a 1
- Passo 3: il client riceve SYNACK e risponde con un segmento ACK, che
può contenere dati
il segmento ACK ha il flag ACK impostato a 1



Per chiudere una connessione abbiamo 4 passi:

- Passo 1: il *client* invia un segmento di controllo FIN al server, con il flag FIN impostato a 1.
- Passo 2: il *server* riceve il segmento FIN e risponde con un ACK e invia un FIN.

- Passo 3: il *client* riceve FIN e risponde con un ACK. Inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve.
- Passo 4: il *server* riceve un ACK. La connessione viene chiusa.



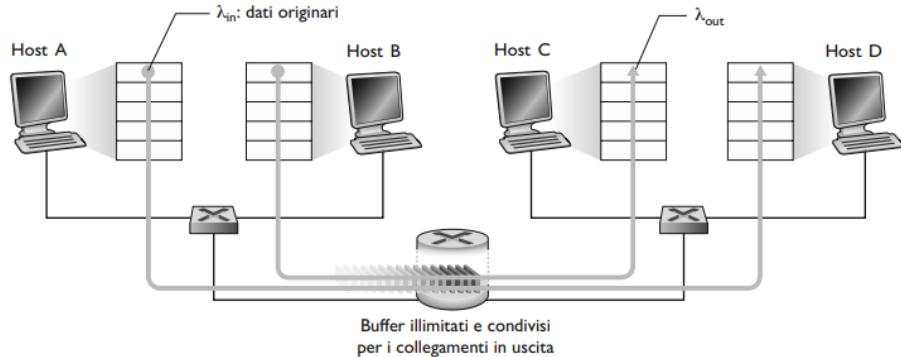
Il client entra in uno stato di attesa temporizzata dopo aver inviato l'ultimo ACK per assicurarsi che il server abbia ricevuto l'ACK e che eventuali segmenti in ritardo vengano gestiti correttamente.

3.6 Principi del controllo di congestione

La **congestione** è il blocco della rete per via di un numero di dati elevato mandati a una velocità elevata che la *rete* non riesce a gestirli, ovvero quando le *risorse di rete sono sovraccaricate*. Ciò causa pacchetti smarriti (overflow nel buffer) e lunghi ritardi (di accodamento nei buffer), che possono *degradare le prestazioni della rete*. La congestione è causata da *più mittenti* che trasmettono dati ad alta velocità.

Scenario di Congestione 1: Due Mittenti e Buffer Illimitati

In questo scenario, due mittenti inviano dati a due destinatari attraverso un singolo router con buffer illimitati e un collegamento condiviso di capacità R. Non ci sono ritrasmissioni.

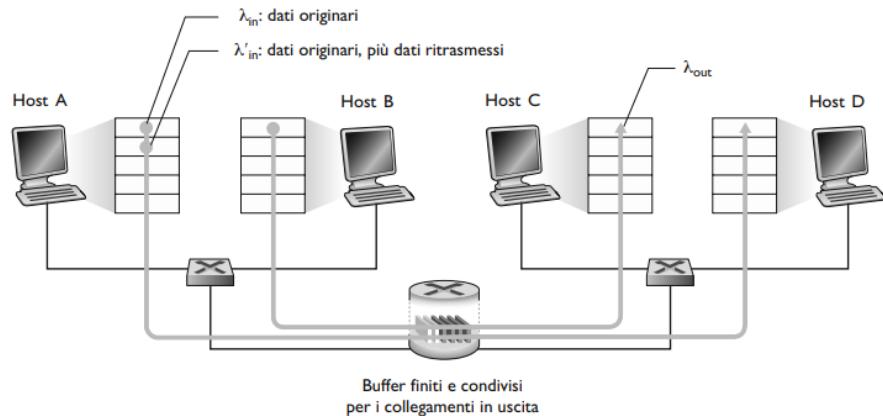


- **Throughput:** Quando il tasso di invio combinato (λ_{in}) è inferiore a $R/2$, il throughput per connessione è uguale al tasso di invio. Quando il tasso di invio combinato supera $R/2$, il throughput per connessione è limitato a $R/2$.
- **Ritardo:** Quando il tasso di invio si avvicina a $R/2$, il ritardo medio aumenta. Quando si arriva ad esser molto vicini a $R/2$ il ritardo medio tende all'infinito a causa dell'accodamento illimitato.

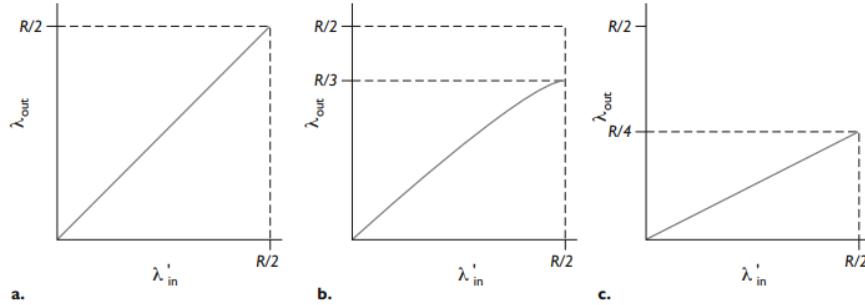
Conclusione: Anche in uno scenario idealizzato con buffer illimitati, la congestione causa ritardi significativi quando il tasso di invio si avvicina alla capacità del collegamento, nonostante si raggiunga il throughput massimo.

Scenario di Congestione 2: Buffer Finiti e Ritrasmissioni

In questo scenario, si considera un router con buffer di dimensione limitata. I pacchetti che arrivano in un buffer pieno vengono scartati, e i mittenti ritrasmettono i pacchetti persi.



- **Caso a) Ritrasmissione "Magica":** Il mittente trasmette solo quando il buffer ha spazio. In questo caso, non ci sono perdite, $\lambda'_{in} = \lambda_{in}$, e il throughput è λ_{in} fino a $R/2$.
- **Caso b) Ritrasmissione Perfetta:** Il mittente ritrasmette solo quando è certo che un pacchetto sia andato perso. In questo caso, $\lambda'_{in} > \lambda_{out}$, e il throughput è $R/3$ quando il carico offerto (λ'_{in}) è $R/2$.
- **Caso c) Ritrasmissione Prematura:** Il mittente può andare in timeout prematuramente e ritrasmettere un pacchetto che non è stato perso. In questo caso, il throughput è $R/4$ quando il carico offerto (λ'_{in}) è $R/2$, a causa di ritrasmissioni non necessarie.

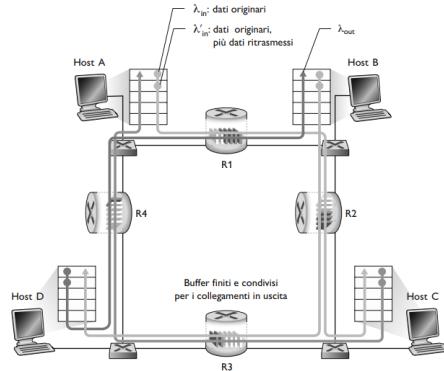


Costi della Congestione:

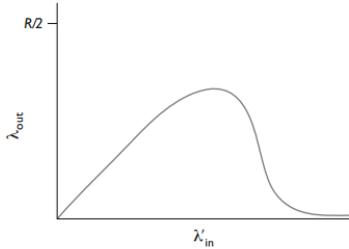
- Più lavoro (ritrasmissioni) per un dato *goodput*.
- Ritrasmissioni non necessarie: il collegamento trasporta più copie dello stesso pacchetto.

Scenario di Congestione 3: Quattro Mittenti e Percorsi Multi-Hop

In questo scenario, quattro mittenti trasmettono dati su percorsi con due collegamenti sovrapposti, con timeout e ritrasmissione. Ogni collegamento ha una capacità di R byte/s.



- **Basso Carico:** Per valori bassi di λ_{in} , gli overflow dei buffer sono rari e il throughput (λ_{out}) è approssimativamente uguale al traffico inviato. Un aumento di λ_{in} porta a un aumento di λ_{out} .
- **Alto Carico:** Per valori elevati di λ'_{in} , il traffico da B a D compete con il traffico da A a C per le risorse del router R2. Il throughput di A-C diminuisce all'aumentare del traffico B-D, tendendo a 0 quando il traffico B-D tende all'infinito.



Costo della Congestione:

- Quando un pacchetto viene scartato, la capacità trasmissiva utilizzata sui collegamenti di upstream per instradare il pacchetto risulta sprecata.

3.7 Controllo di congestione

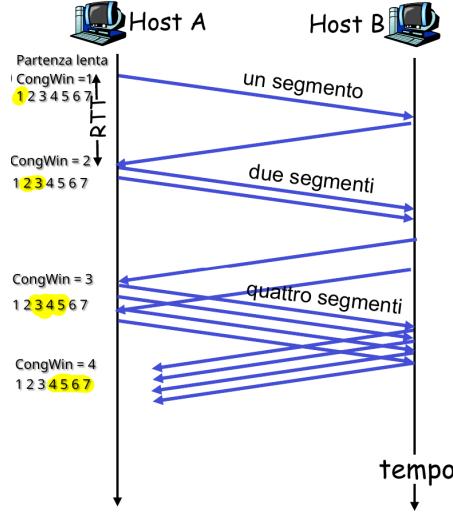
Bisogna limitare la trasmissione, si effettua mediante una "finestra di congestione" (**CongWin**), cioè una funzione dinamica della congestione.

$$\text{Frequenza d'invio} = \frac{\text{CongWin}}{\text{RTT}} \text{ byte/sec}$$

Definiamola come una misura a spanne, non precisa.

Il mittente si accorge della congestione tramite il *timeout* o il *triplice ACK duplicato*. Il mittente riduce la *frequenza d'invio* dopo essersi accorto, con tre meccanismi:

1. **Partenza lenta:** Stabilita la connessione si manda un solo pacchetto. All'inizio la velocità di trasmissione è molto lenta, poi crescerà a livello esponenziale finché non si verifica un evento di perdita, raddoppiamo la *finestra di congestione* dopo ogni *ACK ricevuto*. La *partenza lenta* progredisce fino a un valore di soglia deciso dai progettisti.

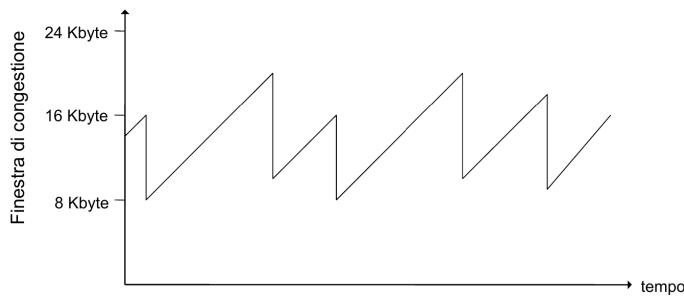


Nel caso di *triplice ACK duplicato* si passa all'algoritmo successivo per una crescita più lenta, impostando però:

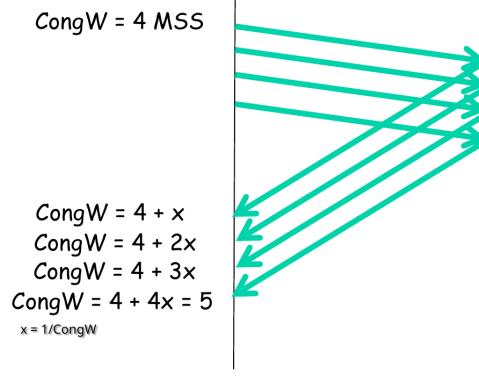
$$\text{valore di soglia} = \frac{\text{CongWin}}{2}$$

CongWin = valore di soglia

2. **AIMD**: Incremento additivo, decremento moltiplicativo in italiano. **Fase a regime**, dopo la partenza lenta. La crescita della *finestra* continua in maniera lineare di **1 MSS** dopo ogni *RTT*, nel caso di perdite la *finestra* viene **dimezzata**. Questo è il suo andamento:



Il suo funzionamento è così diagrammato:



negli esercizi, in caso di perdita di *ACK*, arrotondiamo a +1 la frazione, solo per comodità. Nella realtà, essendo in byte, si fa il calcolo e si mantiene quel valore, non ci sono problemi in caso di perdita di *ACK* poiché l'*ACK* cumulativo conferma pure il pacchetto perso.

3. **Reazione agli eventi di perdita:** Dividiamo i casi. Nel caso in cui si verifica il *timeout* resettiamo la *finestra* a 1, tornando così alla *partenza lenta*, il valore di soglia viene impostato a

$$\text{Valore di soglia} = \frac{\text{CongWin}}{2}$$

$$\text{CongWin} = 1$$

Nel caso in cui abbiamo una perdita (**Fast recovery**), *triple ACK duplucato*, imposto

$$\text{CongWin} = \frac{\text{CongWin}}{2}$$

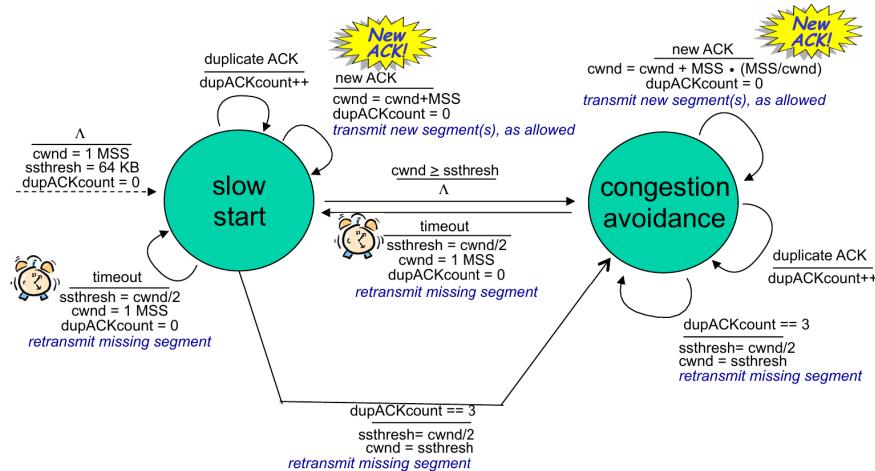
$$\text{Valore di soglia} = \text{CongWin}$$

Diagramma riassuntivo

Il diagramma riassuntivo mostra i due stati principali del controllo di congestione TCP: *slow start* e *congestion avoidance*.

- **Slow Start:** Inizia con una finestra di congestione (cwnd) di 1 MSS (Maximum Segment Size) e una soglia di slow start (ssthresh) di 64 KB. Il contatore di ACK duplicati (dupACKcount) è inizializzato a 0.
 - *Nuovo ACK:* Quando arriva un nuovo ACK, la finestra di congestione (cwnd) viene incrementata di 1 MSS, il contatore di ACK duplicati viene resettato a 0 e vengono inviati nuovi segmenti, se consentito.
 - *Timeout:* Se si verifica un timeout, la soglia di slow start (ssthresh) viene impostata a metà della finestra di congestione corrente, la finestra di congestione (cwnd) viene impostata a 1 MSS, il contatore di ACK duplicati viene resettato a 0 e viene ritrasmesso il segmento perso.

- *Triple ACK Duplicato*: Se si ricevono tre ACK duplicati, la soglia di slow start (ssthresh) viene impostata a metà della finestra di congestione corrente, la finestra di congestione (cwnd) viene impostata al valore della soglia e viene ritrasmesso il segmento perso.
- **Congestion Avoidance**: Si entra in questo stato quando la finestra di congestione (cwnd) raggiunge o supera la soglia di slow start (ssthresh).
 - *Nuovo ACK*: Quando arriva un nuovo ACK, la finestra di congestione (cwnd) viene incrementata di $1 \text{ MSS} * (\text{MSS}/\text{cwnd})$, il contatore di ACK duplicati viene resettato a 0 e vengono inviati nuovi segmenti, se consentito.
 - *Timeout*: Se si verifica un timeout, la soglia di slow start (ssthresh) viene impostata a metà della finestra di congestione corrente, la finestra di congestione (cwnd) viene impostata a 1 MSS, il contatore di ACK duplicati viene resettato a 0 e viene ritrasmesso il segmento perso.
 - *Triple ACK Duplicato*: Se si ricevono tre ACK duplicati, la soglia di slow start (ssthresh) viene impostata a metà della finestra di congestione corrente, la finestra di congestione (cwnd) viene impostata al valore della soglia e viene ritrasmesso il segmento perso.



3.8 Programmazione delle socket

Le **socket** mettono in comunicazione *livello applicativo* con il *livello di trasporto*. Le socket sono *endpoint* per la comunicazione. Esistono due tipi di *socket*: **UDP** e **TCP**.

3.8.1 Programmazione socket TCP

Il *server* crea una socket col comando `socket()`, inizialmente crea la *socket di benvenuto* per l'ascolto, quindi si crea una *socket* senza parametri, bisogna comunicarglieli successivamente. Il *server* crea una *socket di benvenuto* per l'ascolto e una *socket di connessione* per ogni client. Il *server* aggancia i parametri alla *socket* precedentemente creata tramite il comando `bind()`. Mettiamo il *server* in stato di ascolto tramite il comando `listen()`.

Il *client* crea una socket con i parametri del *server*, tramite il comando `socket()`. Il *client* aggancia i 4 parametri alla *socket* precedentemente creata con il comando `bind()`. Il *client* si connette al *server* mediante il comando `connect()`, la *socket client* è creata *implicitamente* con il comando ‘`connect()`’. Il *server* dall'altra parte accetterà la connessione sulla *socket specifica* col comando `accept()`, e manda il messaggio **SYNACK**. Il *client* manda il messaggio al *server* mediante il comando `send()` che verrà ricevuto dal comando `recv()` dal *server*, mandando in risposta tramite `send()` l'**ACK**, che verrà ricevuto dal *client* tramite `recv()`.

Si usa il comando `close()` per chiudere la connessione, possono mandarlo sia *server* che *client*, mandando il messaggio di **FIN**, ricevendo in risposta un **FINACK**.

socket()

Creazione della *socket*: `int s_listen = socket(family, type, protocol);`
`family`: AF_INET specifica IPV4
`type`: SOCK_STREAM per TCP, SOCK_DGRAM per UDP
`protocol`: 0 (pseudo, IP).

Avremo un identificativo della *socket* come ritorno della funzione.

bind()

Aggancio dei parametri alla *socket* precedentemente creata vuota:

`bind(s_listen = localAdd, AddLength);`

Si specifica la porta su cui mettersi in ascolto. ‘`bind()`’ associa la *socket* a un *indirizzo locale e una porta*.

`s_listen`: identificatore della *socket*

`localAdd`: di tipo **sockaddr_in**, una struttura già definita.

`AddLength`: lunghezza della variabile `localAdd`

```

struct sockaddr_in {
    u_char sin_len; // length of address
    u_char sin_family; // family of
                       address
    u_short sin_port; // protocol port num
    struct in_addr sin_addr; // IP Addr
    char sin_zero[8]; // set to zero, used
                      for padding
};

```

L'immagine mostra la struttura `sockaddr_in`, usata per specificare l'indirizzo e la porta a cui legare la socket.

Definisco `struct sockaddr_in sockAdd;` Imposto la famiglia: `sockAdd.sin_family = AF_INET;` Per impostare l'indirizzo IPV4 abbiamo due modi:

1. Specifichiamo l'indirizzo da ascoltare: `inet_pton(AF_INET, "\127.0.0.1", &sockAdd.sin_addr.s_addr);`
2. Ascolta da tutti gli indirizzi locali (in questo caso): `sockAdd.sin_addr.s_addr = htonl(INADDR_ANY);`

Imposto la porta: `sockAdd.sin_port = htons(9999);` `inet_pton()` converte un indirizzo IP in formato testuale in un indirizzo IP in formato binario. `htonl()` e `htons()` convertono un intero in un formato a 32 bit e 16 bit, rispettivamente, indipendente dall'architettura dell'host (big endian o little endian).

```

listen()

int status = listen(s_listen, queuelength);
Risultato: -1 errore, 0 ok
s_listen: riferimento alla socket
queuelength: numero di client che possono stare in attesa.
È una funzione non bloccante, ritorna immediatamente un valore. 'listen()' mette la socket in uno stato di ascolto passivo.

```

```

accept()

int s_new = accept(s_listen, &clientAddress, &AddLength);
s_new: nuova socket per la comunicazione con il client, fino ad adesso abbiamo usato una socket di benvenuto.

```

s_listen: riferimento alla vecchia socket di benvenuto
 clientAddress: riferimento alla struttura sockAddr_in con l'indirizzo del client.
 AddLength: dimensione della variabile clientAddress.
 È una funzione **bloccante**, si *blocca* fino a quando arriva una richiesta di connessione e quindi un **SYN**.

send()

```
int send(int s_new, const void *buf, int len, int flags);
```

s_new: descrittore della socket
 buf: puntatore al buffer
 len: dimensione del buffer
 flags: da impostare a 0. Questa funzione è usata per *inviare dati* attraverso una socket.

recv()

```
int recv(int s_new, void *buf, int len, unsigned int flags);
```

Simile alla send. Questa funzione è usata per *ricevere dati* attraverso una socket.
 buf: conterrà i dati da ricevere.

fork()

Funzione della libreria di C, **biforca** il *processo*, creando **processo padre** e **processo figlio**, processi identici con stesse variabili. Il *processo figlio* è una *copia* del processo padre e gestirà le connessioni con il *client* mentre col *processo padre* gestiamo il *listening*, rimanendo in **accept()**, ogni volta che gli arriverà una nuova richiesta per una socket faremo il **fork()** del processo padre, affidando al figlio la socket appena creata.

3.8.2 Programmazione socket UDP

Non esiste il comando **listen()** nella programmazione socket UDP, poiché non è un protocollo orientato alla connessione.

Per visualizzare i processi in corso, si può usare il comando **ps -ax**. Per visualizzare le socket aperte, si può usare il comando **netstat** (o **netstat -nap** TCP su macOS). Per terminare un processo, si può usare il comando **kill -9 <PID>**.

4 Livello rete

Mette in comunicazione *logica* gli **host**. È implementato in ogni livello, sia nei *router* che nei *sistemi terminali*. È colui che trasporta i vari *segmenti* (mandati dal livello di trasporto) e li *incapsula* in **datagrammi** lato mittente, mentre il destinatario consegna i **datagrammi** al livello trasporto, quindi *segmenti*.

Ci sono due funzioni principali a livello di rete

- **Inoltro o forwarding:** è un'azione *locale*, definisce qual è il percorso per il destinatario designato, dipende dall'**instradamento**, l'*instradamento* *informa* le decisioni di inoltro e aggiorna i percorsi e li comunica all'*inoltro*.
- **Intradamento o routing:** Consiste nel trovare (non come) la strada migliore per andare da una sorgente a una destinazione. Gli algoritmi di instradamento determinano i *percorsi* che i pacchetti seguono.

4.1 Architettura del router

Nel router abbiamo due componenti, una si occupa dell'intradamento, calcolando i *percorsi*, e una si occupa dell'inoltro, muovendo i *pacchetti* lungo quei percorsi. In alto abbiamo la componente che gestisce l'intradamento con un **algoritmo di instradamento**, sotto abbiamo la componente che gestisce l'inoltro, che avrà una tabella con tutte le informazioni mandate dalla componente di instradamento.

Il primo piano si chiama **piano di controllo**, responsabile della gestione delle *tabelle di instradamento*.

Il secondo piano si chiama **piano dei dati**, responsabile dell'elaborazione dei pacchetti ad *alta velocità*, diviso in:

- **Porte di ingresso:** porte logiche, non sono le porte fisiche, è uno stream di dati *socket*. Hanno il livello fisico (ricezione di dati), livello di collegamento (ethernet) e livello di rete con la commutazione decentralizzata: determina la porta d'uscita dei pacchetti tramite la tabella d'inoltro, nel caso in cui arrivano tanti datagrammi che il router non riesce a manipolare man mano crea un *buffer di accodamento* dove memorizza i pacchetti. Le porte di ingresso eseguono la *ricezione a livello fisico*, l'*elaborazione a livello di collegamento* e la *ricerca a livello di rete*.
- **Struttura di commutazione:** inizialmente era un computer che manipolava i dati e aveva le porte come periferiche (*Commutazione in memoria*), un metodo *più vecchio e più lento*, ora usiamo la **commutazione tramite bus**: le porte d'ingresso gestiscono l'indirizzamento del pacchetto, manipolano loro la circuiteria per l'intradamento, un metodo *più efficiente*. La soluzione ideale è **crossbar switch**, sono percorsi in parallelo con $2n$ bus che collegano n porte d'ingresso a n porte d'uscita, una struttura di commutazione *non bloccante*.

- **Porte di uscita:** porte logiche, non sono le porte fisiche, è uno stream di dati *socket*. Hanno gli stessi livelli della *porta d'ingresso* ma in modo speculare. Le porte di uscita eseguono l'*elaborazione a livello di rete, l'elaborazione a livello di collegamento e la trasmissione a livello fisico*. Le funzionalità di *livello rete* sono: **funzionalità di accodamento** che riframmenta i pacchetti nel caso in cui il *livello di collegamento* ha un *MTU* (Maximum Transmission Unit) inferiore al precedente. Le porte di uscita eseguono l'*accodamento* e la *frammentazione* quando necessario.

4.1.1 Tabelle di inoltro

Le tabelle di inoltro, utilizzate per determinare la porta di uscita di un pacchetto, sono elaborate e aggiornate dal processore di instradamento. Una copia di queste tabelle è memorizzata su ciascuna porta di ingresso per velocizzare il processo di inoltro.

Ricerca del prefisso più lungo:

- La ricerca nella tabella di inoltro si basa sul confronto tra un prefisso dell'indirizzo di destinazione del pacchetto e le righe della tabella.
- Se un indirizzo di destinazione corrisponde a più righe, il router adotta la regola di corrispondenza a prefisso più lungo, inoltrando il pacchetto all'interfaccia di collegamento associata alla corrispondenza più lunga.

Elaborazione alle porte di ingresso:

- La ricerca nella tabella di inoltro è effettuata in hardware.
- Oltre alla ricerca, le porte di ingresso eseguono anche:
 - Elaborazione a livello fisico e di collegamento.
 - Controllo e riscrittura del numero di versione del pacchetto, del checksum e del tempo di vita.
 - Aggiornamento dei contatori per la gestione di rete.

Astrazione Match-Action: L'azione di cercare la corrispondenza tra l'indirizzo IP di destinazione e inviare il pacchetto alla porta di uscita specificata è un caso specifico di un'astrazione più generale "match-action", che viene eseguita in molti dispositivi di rete, non solo nei router.

4.2 Protocollo internet: IP

Il protocollo **IP** (sia in versione 4 che in versione 6) stabilisce:

- Convenzioni di indirizzamento, ovvero come gli indirizzi IP sono usati per *identificare univocamente* gli host sulla rete.
- Formato dei datagrammi, che include *header fields* per il routing e il controllo.

- La manipolazione dei pacchetti, ovvero la *frammentazione e il riassemblaggio* dei datagrammi.

Il protocollo IP fornisce un trasferimento di datagrammi *non affidabile e connectionless*. Il protocollo **ICMP**, dipendente dal protocollo **IP**, ovvero i messaggi ICMP sono *trasportati all'interno di datagrammi IP*:

- Notifica gli errori, ovvero ICMP è usato per riportare *errori di rete* come host irraggiungibili o timeout.
- Segnalazione del router, ovvero ICMP è usato per la *scoperta dei router* e la *scoperta del MTU del percorso*.

4.3 IPv4, Protocollo IP versione 4

Ogni *interfaccia di host e router* hanno un **indirizzo IP univoco da 32 bit**. L'**interfaccia** è il confine tra host e collegamento fisico, i *router* devono avere almeno due collegamenti fisici e per ogni *interfaccia* è associato un **indirizzo IP**.

4.3.1 Formato dei datagrammi

Lungo 32 bit. Abbiamo i seguenti campi:

1. (a) **Versione** del protocollo (4 o 6)
 (b) **Lunghezza intestazione**, variabile
 (c) **Tipo di servizio** quanto è importante il datagramma (non molto utilizzato)
 (d) **Lunghezza del datagramma**, che include sia l'header che i dati.
2. (a) **Identificatore a 16 bit**: usato per il *riassembaggio* dei datagrammi frammentati.
 (b) **flag**: usati per il *controllo della frammentazione*.
 (c) **Spiazzamento di frammentazione a 13bit**: indica la *posizione del frammento* nel datagramma originale.
3. (a) **Tempo di vita residuo**: *TTL - Time to live*, quanti router può attraversare il datagramma, il datagramma nasce con un tempo di vita previsto, a ogni passaggio viene decrementato di 1, quando il *TTL* arriva a 0 il router elimina il pacchetto. Il TTL è usato per *prevenire i loop di instradamento*.
 (b) **Protocollo di livello superiore**: specifica quale *protocollo* stiamo utilizzando a livello di *trasporto*, per sapere a quale servizio del *sistema operativo* mandare i pacchetti. Questo campo indica il *protocollo usato dal payload* (es. TCP o UDP).

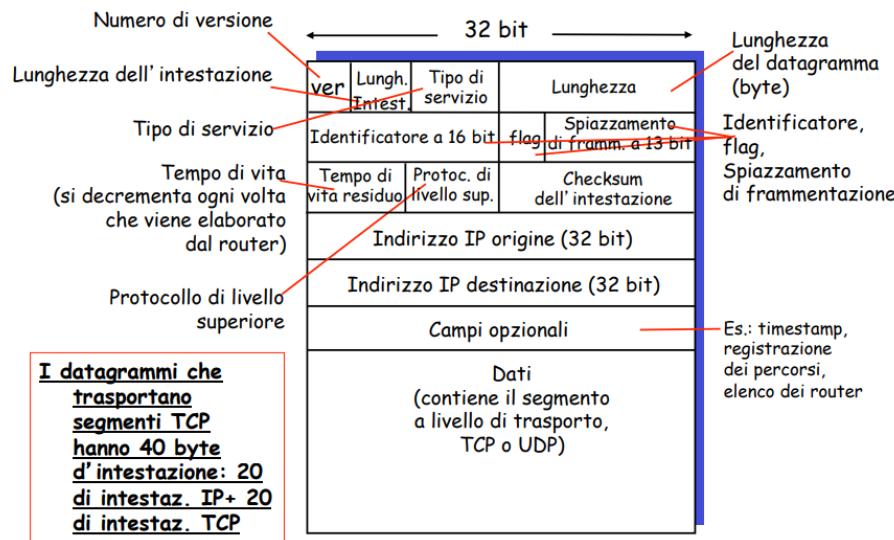
- (c) **Checksum** della sola intestazione, stesso algoritmo del protocollo *UDP*, chiamato **checksum internet**, non comprende il campo *Dati*. Viene calcolata da ogni *router* in cui passa. Il checksum è calcolato solo sull'*header* e non sui dati.

4. Indirizzo IP origine

5. Indirizzo IP destinazione

6. **Campi opzionali:** usati per *funzionalità opzionali* come la sicurezza o il timestamping.

7. Dati



4.3.2 Frammentazione dei datagrammi IP

L'unità massima di trasmissione (**MTU**) è la quantità massima di dati che possono passare in quel determinato *livello di collegamento*. MTU è un concetto del *livello di collegamento*. I **datagrammi IP** vengono frammentati in *datagrammi più piccoli* quando un datagramma è *più grande dell'MTU* di un collegamento.

4.3.3 Sottorete

L'*indirizzo IP* è diviso in due parti, per l'*indirizzamento gerarchico*:

- **Parte di sottorete:** bit di alto ordine, che identificano la *rete* o la *sottorete*.
- **Parte dell'host:** bit di basso ordine, che identificano l'*host specifico* all'interno della sottorete.

Una sottorete è definita anche come *reti IP*, ovvero una *rete logicamente separata* all'interno di una rete più grande.

4.3.4 Assegnazione indirizzi internet CIDR

CIDR: Classes InterDomain Routing, un *metodo per allocare indirizzi IP e instradare pacchetti IP*. L'*indirizzo IP* viene diviso in $a.b.c.d/x$ dove x è un numero in bit che definisce la **maschera di rete**, ovvero il *numero di bit nel prefisso di rete*. Facendo $32 - x$ avremo il numero di bit "liberi" per la *sottorete*, ovvero il *numero di bit disponibili per gli indirizzi host* nella sottorete.

Esempio

200.23.16.0/23 diventa 11001000.00010111.00010000.00000000 dove la parte in grassetto è la **parte di host**, mentre il resto è il **prefisso di rete**. Avendo $x = 23$ faremo $32 - 23 = 9$, quindi avremo correttamente 9 bit di **parte di host**.

4.3.5 Indirizzamento - convenzioni

- **Broadcast:** tutti i bit della **parte di host** posti a 1. Il datagramma viene consegnato a *tutti gli host* della sottorete.
- **Identificativo della rete:** Tutti i bit della **parte di host** è posta a 0. Indirizzo della rete, usato per *identificare la sottorete stessa*.
- **Identificatore del router:** solitamente il primo indirizzo disponibile (dopo quelli *riservati*, ovvero alcuni indirizzi sono *riservati* e non possono essere usati per gli host) è del *router*, usato come *default gateway* per gli host nella sottorete.

4.3.6 Netmask

Una **netmask** è una sequenza di 32 bit associata a un indirizzo IP per l'individuazione del prefisso di rete e parte di host. La netmask è usata *insieme a un indirizzo IP* per determinare la parte di rete e la parte host. I bit che valgono 1 identificano la *parte di rete* e quelli che valgono 0 identificano la *parte di host*. La sequenza *contigua* di 1 identifica la parte di rete e la sequenza *contigua* di 0 identifica la parte host.

11111111 . 11111111 . 11111111 . 11000000	Netmask
255 . 255 . 255 . 192	
11000000 . 10101000 . 00001010 . 01000101	Indirizzo
192 . 168 . 10 . 69	
Prefisso di rete	Host

L'immagine mostra come la netmask separa la parte di rete e la parte host di un indirizzo IP.

4.3.7 Inoltro dei pacchetti: Host

Operazioni:

1. Verifica se il destinatario è nella stessa sottorete: si effettua l'*AND* tra il proprio indirizzo e la propria *netmask*, e l'*AND* tra l'indirizzo di destinazione e la propria *netmask*, per *estrarre il prefisso di rete* da entrambi gli indirizzi, e si verifica se sono uguali.
2. Se è sulla stessa *sottorete* invia direttamente all'host, usando l'*indirizzo di livello collegamento* del destinatario. Se non è sulla stessa *sottorete* invia al router, che è il *default gateway* per la sottorete.

4.3.8 Inoltro dei pacchetti: Router

Il router esegue le seguenti operazioni per inoltrare un pacchetto:

- Verifica se il destinatario è in una delle sottoreti direttamente connesse al router.
- Altrimenti, per ogni riga della tabella di routing:
 - Esegue un AND logico tra l'indirizzo di destinazione e la netmask della riga.
 - Verifica se il risultato è uguale alla rete di destinazione.
- Tra tutte le righe che hanno avuto successo, sceglie quella con la netmask più lunga.
- Se non c'è corrispondenza, invia sulla default route.

4.3.9 Come ottenere un blocco di indirizzi IP

Bisogna contattare il proprio **ISP** e ottenere la divisione in otto blocchi uguali di indirizzi contigui. Gli ISP sono responsabili di *allocare gli indirizzi IP* ai loro clienti. La divisione in otto blocchi è una *pratica comune* ma non un requisito rigido. Gli indirizzi contigui sono importanti per un *instradamento efficiente*.

4.3.10 Come ottenere un singolo indirizzo

Due approcci:

- **configurazione manuale:** si imposta manualmente l'indirizzo IP alla macchina, richiedendo l'*intervento di un amministratore*.
- **DHCP:** Dynamic Host Configuration Protocol, un *protocollo client-server*, assegna automaticamente un indirizzo IP, una subnet mask, un default gateway e altri parametri di rete una volta connesso in rete.

4.3.11 DHCP

È una funzionalità di livello rete ma gestita da un processo a livello applicativo che utilizza delle socket UDP, porta con numero 67 (il server usa la porta 67 come *porta di destinazione*), mentre i client aprono la porta 68 (il client usa la porta 68 come *porta di destinazione*) con indirizzo IP 0.0.0.0. DHCP è un *protocollo a livello applicativo* che usa UDP.

Il client appena inserito sulla rete non ha, giustamente, indirizzo IP e non conosce l'indirizzo IP del server DHCP, quindi il client invia un messaggio *broadcast* (usando un *indirizzo broadcast limitato* come 255.255.255.255) sulla porta 67, cercando un server DHCP, anche il server DHCP risponderà in *broadcast* (usando un *indirizzo broadcast limitato*), poiché il destinatario non ha ancora un *indirizzo IP*.

Consente di ottenere **dinamicamente** gli indirizzi IP degli *host*, assegnando indirizzi IP da un *pool di indirizzi disponibili*. Non assegna obbligatoriamente lo stesso IP all'*host*, varia in base a quelli che ha disponibile. Il **transaction ID** serve al *server* per sapere con chi sta parlando e a chi assegnare l'*indirizzo IP*, usato per *abbinare richieste e risposte*.

1. **DHCP discover**: da parte degli host, un messaggio broadcast a tutta la rete alla ricerca di un **server DHCP**.
2. **DHCP offer**: il *server DHCP* offre un indirizzo IP all'*host*.
3. **DHCP request**: l'*host* accetta l'indirizzo IP proposto dal *server DHCP*.
4. **DHCP ack**: il *server DHCP* invia l'ack di conferma.

4.3.12 NAT

Consente di separare una rete specifica dalle altre, crea la **rete privata**. L'acronimo è: **Network address translation**, usato per *conservare gli indirizzi IP pubblici*.

Il **NAT** si trova all'interno del *router*, che si trova al *confine* tra la rete privata e quella pubblica. Le reti esterne vedono la rete privata come un **unico indirizzo IP**, che sarà quello assegnato al *router*. L'indirizzo IP pubblico è l'**unico indirizzo IP** visibile all'esterno. Le macchine all'interno della *rete privata* avranno degli **indirizzi IP privati**, che saranno visibili solo all'interno della *rete privata*. Le reti private usano *range di indirizzi IP privati*.

Il vantaggio è di avere un unico indirizzo IP fornito dall'*ISP* per la **rete pubblica**, andando a mascherare gli *indirizzi IP privati* alle *rete esterne*.

Implementazione

Il router NAT riceve un datagramma, genera un numero di porta d'origine per quella macchina che ha mandato il datagramma, sostituisce l'indirizzo IP d'origine con il proprio per la rete esterna e sostituisce il numero di porta iniziale con quello generato precedentemente. Il router NAT *mantiene una tabella di*

traduzione per mappare indirizzi IP privati e porte a indirizzi IP pubblici e porte.

Il router accede al datagramma, modificando nell'*header IP* e nell'*header del livello di trasporto* la porta d'origine e ricalcola la checksum a livello trasporto, poi cambia l'indirizzo IP e ricalcola la checksum a livello di rete.

Il protocollo NAT può gestire al massimo tante connessioni quante sono le porte disponibili, quindi NAT ha *limitazioni* nel numero di connessioni concorrenti.

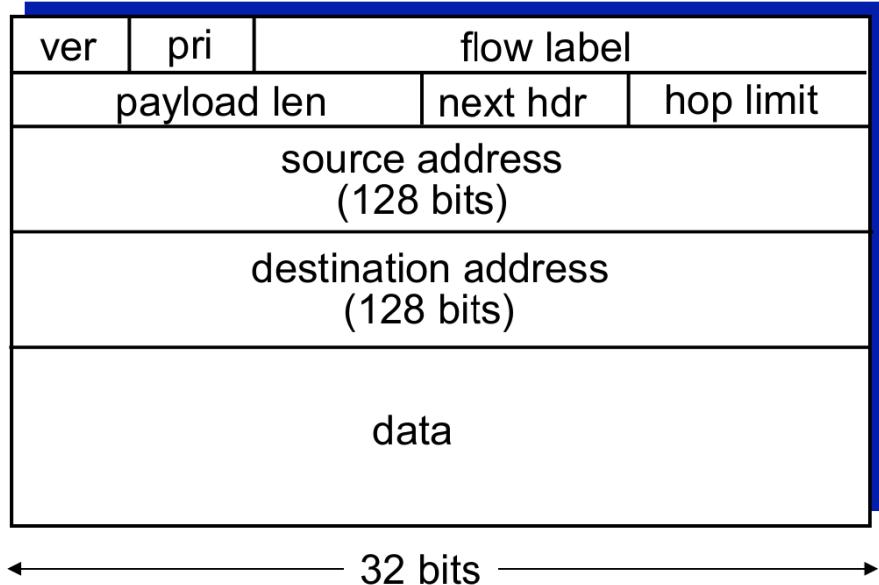
4.4 IPv6

Il motivo per cui è stato progettato il protocollo **IP versione 6** è che si stanno esaurendo gli indirizzi IP a 32 bit. Gli indirizzi IPv6 sono *128 bit*, rispetto ai 32 bit di IPv4. Inoltre è stato progettato per migliorare l'infrastruttura generale e velocizzarla.

4.4.1 Formato dei datagrammi

Il formato dei datagrammi **IPv6** ha un'intestazione a 40 byte ed è di lunghezza *fissa* e *semplificata* al contrario della versione 4. Inoltre non è consentita la frammentazione, il router non potrà frammentare il datagramma, la frammentazione è gestita dagli *end system* in IPv6.

1. (a) **ver**: versione del protocollo IP in uso.
(b) **pri**: priorità di flusso, indica la priorità del datagramma.
(c) **flow level**: identifica i datagrammi appartenenti allo stesso flusso.
2. (a) **payload len**: lunghezza del payload del datagramma.
(b) **next hdr**: indica il tipo di header successivo (es. TCP, UDP, extension header).
(c) **hop limit**: ttl, indica il numero massimo di hop che il datagramma può attraversare.
3. **source address**: indirizzo di origine a *128 bit*.
4. **destination address**: indirizzo di destinazione a *128 bit*.
5. **data**



Le varie novità di **IPv6** sono:

- Elimina i campi di frammentazione.
- Checksum: eliminata dall'header IP e gestita dai livelli superiori.
- Opzioni: il campo non è scomparso ma è nelle *extension headers* puntate da "next hdr".
- ICMPv6: nuova versione di ICMP, usata per la *scoperta dei vicini* e la *scoperta dei router*, oltre che per la segnalazione degli errori.

Il protocollo riesce a mascherare i datagrammi IPv6 in datagrammi IPv4 quando si passa su un hop che parla solo IPv4, tramite il tunneling che permette di *incapsulare i datagrammi IPv6 in datagrammi IPv4*.

5 Livello di rete (Piano di controllo)

Devo farlo da solo a casa che sono stanco