

# Reti e sistemi operativi - Reti

Jacopo De Angelis

6 luglio 2019

## Indice

<b>1 Reti di calcolatori e internet</b>	<b>4</b>
1.1 Che cos'è internet? . . . . .	4
1.1.1 Una descrizione pratica . . . . .	4
1.1.2 Descrizione del servizio . . . . .	4
1.1.3 Che cos'è un protocollo? . . . . .	4
1.2 La sezione di accesso alla rete . . . . .	4
1.2.1 Terminali, client e server . . . . .	4
1.2.2 Servizio senza connessione e servizio orientato alla connessione . . . . .	5
1.3 La sezione interna della rete . . . . .	7
1.3.1 Comutazione di circuito e commutazione di pacchetto . . . . .	7
1.3.2 Comutazione di circuito . . . . .	7
1.4 Accesso alla rete e mezzi trasmissivi . . . . .	10
1.4.1 Accesso alla rete . . . . .	10
1.4.2 Mezzi fisici . . . . .	12
1.5 Gli ISP e la rete dorsale di internet . . . . .	12
1.6 Ritardi e perdite nelle reti a commutazione di pacchetto . . . . .	13
1.6.1 Tipi di ritardo . . . . .	13
1.6.2 Ritardo di coda e perdita di pacchetti . . . . .	14
1.7 Strati protocollari . . . . .	14
1.7.1 Architettura stratificata . . . . .	14
1.7.2 La pila protocollare di internet . . . . .	15
<b>2 Strato di trasporto</b>	<b>17</b>
2.1 Introduzione e servizio dello strato di trasporto . . . . .	17
2.1.1 Relazione fra gli strati di trasporto e di rete . . . . .	17
2.1.2 Panoramica dello strato di trasporto in internet . . . . .	17
2.2 Multiplexing e demultiplexing . . . . .	19
2.2.1 Multiplazione e demultiplazione orientate alla connessione	20
2.3 Trasporto senza connessione: UDP . . . . .	20
2.3.1 Struttura del segmento UDP . . . . .	22

2.3.2	Checksum di UDP . . . . .	24
2.4	Principi di un trasferimento affidabile dei dati . . . . .	24
2.4.1	Costruzione di un protocollo per il trasferimento affi- dabile dei dati . . . . .	24
2.4.2	Protocolli pipeline per il trasferimento affidabile dei dati	30
2.4.3	Go-Back-N (GBN) . . . . .	33
2.4.4	Ripetizione selettiva (SR . . . . .	36
2.5	Trasporto orientato alla connessione: TCP . . . . .	38
2.5.1	La connessione TCP . . . . .	38
2.5.2	Struttura del segmento TCP . . . . .	39
2.5.3	Trasferimento affidabile dei dati . . . . .	41
2.5.4	Controllo di flusso . . . . .	43
2.5.5	Gestione della connessione TCP . . . . .	44
2.6	Principi del controllo della congestione . . . . .	48
2.6.1	Le cause e i costi della congestione . . . . .	48
2.7	Controllo della congestione del TCP . . . . .	53
2.8	Controllo della congestione del TCP . . . . .	53
2.8.1	Fairness . . . . .	57
<b>3</b>	<b>Strato di rete e instradamento</b>	<b>59</b>
3.1	Introduzione . . . . .	59
3.1.1	Forwarding e Routing . . . . .	59
3.1.2	Modelli di servizio di network . . . . .	62
3.2	Il protocollo di internet (IP): forwarding e indirizzamento (ad- ressing) in internet . . . . .	63
3.2.1	Formato dei datagrammi IPv4 . . . . .	63

## Programma esteso

- Introduzione e cenni al livello fisico *cap. 1.1 - 1.5 (no 1.3.2)*
- Livello di trasporto *cap. 3 (no 3.6.2 e 3.7.2)*:
  - funzioni del livello di trasporto
  - trasporto UDP
  - trasporto TCP
  - controllo della congestione
- Livello di rete *cap. 4.1, 4.3 (no 4.3.5), 5.2, 5.6*:
  - funzioni del livello di rete
  - indirizzamento IP
  - algoritmi di instradamento
- LAN, Wireless LAN, Elementi di livello fisico *cap. 6.1, 6.2, 6.3 (no 6.3.4), 6.4, 7.1, 7.2 (no 7.2.1), 7.3 (no 7.3.6) :*
  - funzioni del livello di collegamento
  - CSMA/CD e LAN Ethernet

# 1 Reti di calcolatori e internet

## 1.1 Che cos'è internet?

### 1.1.1 Una descrizione pratica

Internet è una rete pubblica di calcolatori sparsi in tutto il mondo. I terminali sono collegati tra di loro attraverso link di comunicazione, diversi link possono trasmettere i dati a differenti velocità. Questa velocità è chiamata “larghezza di banda” e di solito si misura in bit/secondo.

Solitamente i terminali sono collegati indirettamente attraverso dispositivi di comunicazione detti router. Un router preleva le informazioni che arrivano tramite uno di questi link in ingresso e lo reindirizza a un link in uscita. Queste informazioni sono chiamate “pacchetti”. Il tragitto del pacchetto da è detto route o path. Internet usa una tecnica conosciuta come “commutazione di pacchetto” (packet switching) che permette a più terminali di condividere un cammino o anche solo parte di questo. I terminali accedono a internet attraverso gli ISP (Internet Service Providers). Ogni ISP è una serie di router e di link di comunicazione.

I terminali eseguono protocolli di comunicazione che controllano l'invio e la ricezione di informazioni all'interno di internet. Due dei più importanti sono il TCP (Transmission Control Protocol) e l'IP (Internet Protocol). Il protocollo IP specifica il formato dei pacchetti che sono scambiati fra router e fra terminali.

### 1.1.2 Descrizione del servizio

Internet permette la distribuzione delle applicazioni che girano sui suoi terminali per scambiare dati fra le diverse unità. Internet fornisce due servizi per le applicazioni da esso distribuite: un servizio orientato alla connessione e un servizio senza connessione. Il primo garantisce che i dati trasmessi saranno consegnati al destinatario nella loro integrità, il secondo invece risulta inaffidabile.

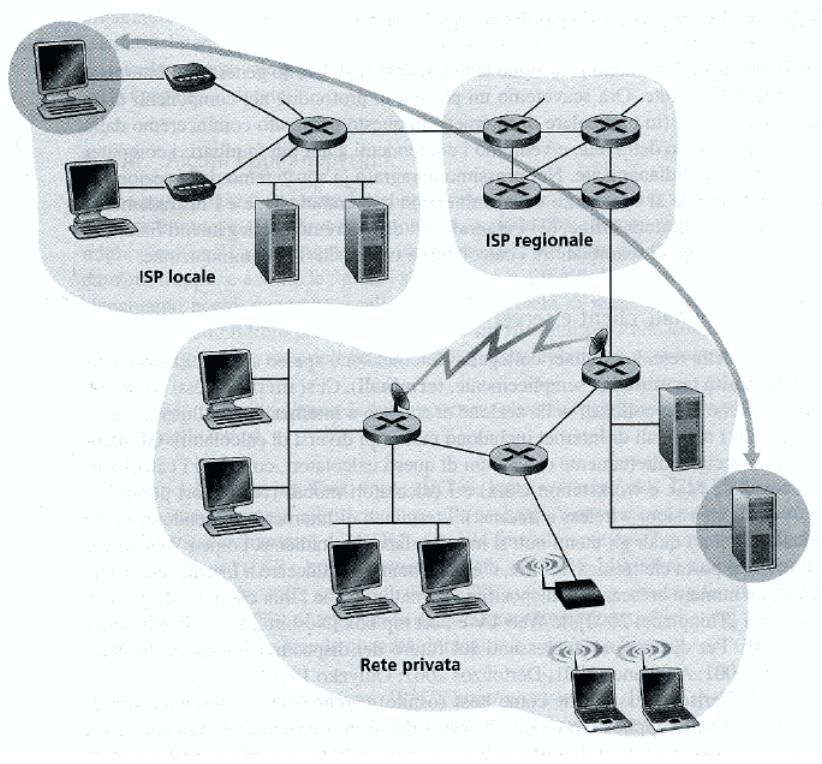
### 1.1.3 Che cos'è un protocollo?

Un protocollo definisce il formato e l'ordine dei messaggi scambiati tra due o più entità comunicanti, così come le azioni che hanno luogo a seguito della trasmissione e/o ricezione di un messaggio o di altri eventi.

## 1.2 La sezione di accesso alla rete

### 1.2.1 Terminali, client e server

I computer colegati a internet sono spesso chiamati host o end-system (“terminali”). Ci si riferisce a essi come terminali perché sono collocati nella



sezione di accesso a internet. Il termine host (ospite) deriva dal fatto che ospitano (eseguono) programmi di livello applicativo come i browser o simili. Gli host, a volte, sono suddivisi in due categorie:

- Client
- Server

Un programma client è un programma che gira su un terminale che richiedere e riceve un servizio da un programma server che gira su un altro terminale. Poiché, tipicamente, un client gira su un calcolatore mentre il server gira su un altro, le applicazioni client/server in internet sono, per definizione, applicazioni distribuite. A questo livello di astrazione i router, i link e altri “componenti” di internet servono come “scatola nera” che trasferisce messaggi tra i diversi componenti per la comunicazione in internet.

### 1.2.2 Servizio senza connessione e servizio orientato alla connessione

Le reti TCP/IP, e in particolare internet, forniscono due tipi di servizio per le sue applicazioni:

- Un servizio senza connessione

- Un servizio orientato alla connessione

Chi crea un'applicazione internet deve programmare l'applicazione per l'impiego di uno di questi due servizi.

**Servizio orientato alla connessione** Il client e il server (residenti in due diversi terminali) si scambiano pacchetti di controllo prima di spedire i pacchetti contenenti i dati reali. Queste procedure di "stretta di mano" (handshaking procedure), allertano client e server, permettendo loro di prepararsi per l'arrivo massiccio dei pacchetti. Una volta terminata questa procedura la connessione tra i due terminali è instaurata.

Perché si utilizza la terminologia "servizio ORIENTATO alla connessione" e non semplicemente "servizio di connessione"? Questo perché solo i due terminali sono coscienti della connessione, all'interno della rete i commutatori sono ignari della connessione e non mantengono informazioni sullo stato della connessione. Il servizio orientato alla connessione di internet è raggruppato con molti altri servizi:

- **Trasferimento di dati affidabile:** un'applicazione può affidarsi a una connessione per consegnare tutti i suoi dati senza errori o nell'ordine appropriato. Questa affidabilità deriva dall'impiego di segnali di riscontro e ritrasmissioni.
- **Controllo di flusso:** assicura che nessuna delle due estremità saturi l'altra con l'invio a velocità eccessiva di troppi pacchetti.
- **Controllo della congestione:** previene che internet entri nello stato di blocco incrociato (gridlock), ovvero quando un router è congestionato e rischia di perdere pacchetti. In questa circostanza, se le velocità di comunicazione continuano a riempire la rete troppo velocemente, i pacchetti saranno persi per la maggior parte. Internet evita questo problema costringendo i terminali a ridurre la velocità di invio durante i periodi di congestione, riscontrata grazie alla mancanza dei messaggi di riscontro.

**Il servizio orientato alla connessione di internet ha un nome: TCP (Transmission Control protocol).**

**Servizio senza connessione** Non esiste handshake. Quando un'estremità di un'applicazione vuole inviare pacchetti a un'altra, semplicemente, li invia. Poiché manca l'handshake l'invio sarà più veloce ma non esisterà un messaggio di riscontro dell'avvenuta ricezione. **Il servizio di internet senza connessione è l'UDP (User Datagram Protocol).**

### 1.3 La sezione interna della rete

#### 1.3.1 Comutazione di circuito e commutazione di pacchetto

Esistono due principali tipi di approccio per la costruzione della sezione interna di una rete:

- **la commutazione di circuito** (circuit switching): le risorse necessarie lungo un percorso per fornire la comunicazione fra due terminali sono riservate per la durata della sessione.
- **la commutazione di pacchetto** (packet switching): le risorse non sono riservate, i messaggi della sessione utilizzano le risorse a richiesta e, di conseguenza, devono aspettare per accedere al link di comunicazione.

Le reti telefoniche sono un esempio di rete a commutazione di circuito. Internet, invece, è un esempio di rete a commutazione di pacchetto. Le reti non sono per forza o di un tipo o dell'altro.

#### 1.3.2 Comutazione di circuito

Nell'immagine qua accanto i quattro commutatori di circuito sono collegati da due link. Ognuno di questi link è costituito da n circuiti, in modo che ciascun link possa mantenere n connessioni simultanee. I terminali sono collegati direttamente a uno dei commutatori. Alcuni degli host hanno un accesso analogico ai commutatori, mentre altri hanno un accesso numerico diretto. Per l'accesso analogico è necessario un modem. Quando due host desiderano comunicare, la rete stabilisce un circuito dedicato end-to-end fra essi. In questo caso viene prenotato un circuito su ciascuno dei due link.

#### **Multiplazione (multiplexing) nelle reti a commutazione di circuito)**

Un circuito in un link è realizzato mediante la multiplazione a divisione di frequenza (FDM) o la multiplazione a divisione di tempo (TDM). Con l'FDM, lo spettro di frequenza di un link è diviso fra le connessioni stabilite sul link, dedicando così una banda di frequenza.

Per il TDM il dominio temporale è suddiviso tra quattro circuiti con quattro time slot in ciascun frame; a ciascun circuito è assegnato lo stesso slot nei frame TDM che si succedono. La velocità di trasmissione di ciascun circuito è uguale a:

$$\text{velocità del frame} * \text{numero di bit in uno slot}$$

per esempio, se il link trasmette 8000 frame al secondo e ogni slot è costituito da 8 bit, la velocità di trasmissione è 64 Kb/s.

i fautori della commutazione di pacchetto hanno sempre sostenuto che la commutazione di circuito porta a sprechi, perché i circuiti dedicati sono inattivi durante i periodi silenti (ad esempio quando la linea telefonica non viene usata).

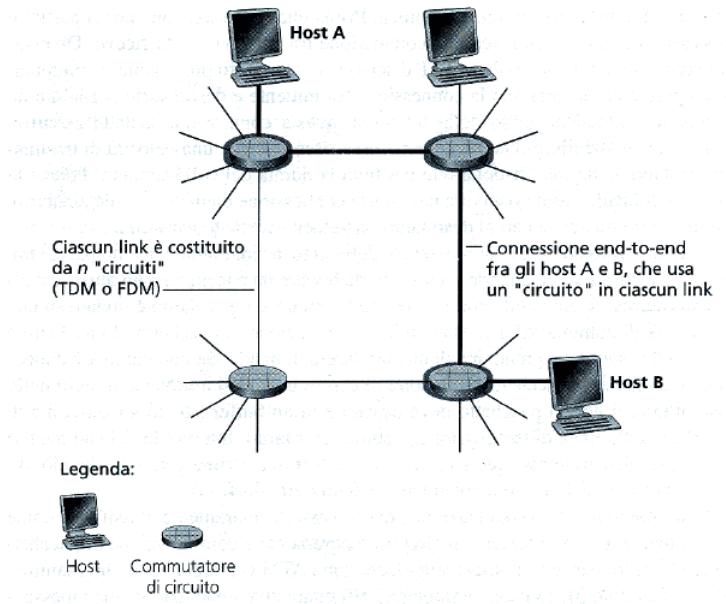


Figura 1: Una semplice rete a commutazione di circuito, che consiste di quattro commutatori di circuito collegati da quattro link

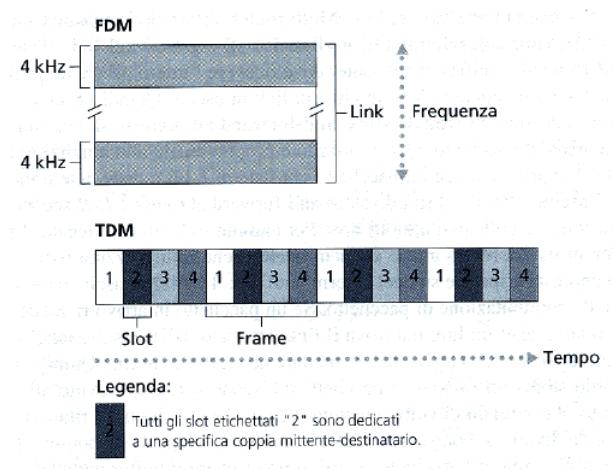


Figura 2: Con L?FDM ciascun circuito occupa continuamente una frazione della larghezza di banda. Con il TDM ciascun circuito occupa periodicamente tutta la larghezza di banda durante brevi intervalli di tempo (durante gli slot)

**Commutazione di pacchetto** Nelle moderne reti di calcolatori, la sorgente suddivide i messaggi lunghi in pezzi più piccoli di dati conosciuti come pacchetti.

Fra sorgente e destinazione ciascuno di questi pacchetti viaggia lungo link di comunicazione e commutatori di pacchetto (router). I pacchetti sono trasferiti sulla linea con velocità massima rispetto a quella del link.

Molti router utilizzano la **trasmissione “store and forward”** all’ingresso dei link, ovvero che il router deve ricevere l’intero pacchetto prima di poter cominciare a trasmettere il primo bit sul link in uscita. Quindi i router store and forward introducono un ritardo store and forward all’ingresso di ciascun link. Questo ritardo è proporzionale alla lunghezza in bit del pacchetto, in particolare un pacchetto di  $L$  bit inoltrato su di un link in uscita a  $R$  bit/s ci metterà un ritardo di  $L/R$  secondi. Ogni router è collegato a molti link. Per ciascun link cui è collegato il router ha un buffer di uscita che immagazzina pacchetti che il router si appresta a spedire su quel determinato link. Nel caso un pacchetto in uscita trovi il link occupato, dovrà attendere che il pacchetto precedente venga spedito, aggiungendo così un ulteriore ritardo chiamato “ritardo di coda”. L’entità di questo ritardo è variabile ed è dipendente dalla congestione della rete. In certi casi si può anche perdere pacchetti, sia in arrivo che in uscita.

La commutazione di pacchetto impiega la multiplazione statistica, in netto contrasto con la multiplazione a divisione di tempo. In questo caso, infatti, i pacchetti vengono spediti in ordine di arrivo. Ora proviamo a calcolare il tempo che occorre ad inviare un pacchetto di  $L$  bit da un host all’altro attraverso una rete a commutazione di pacchetto. Supponiamo che esistano  $Q$  link fra i due host, ciascuno con velocità  $R$  bit/s. assumiamo ritardi dovuti a coda e propagazione end-to-end trascurabili e che non sia stabilita alcuna connessione (niente handshake). Il pacchetto dovrà passare per  $Q-1$  link, quindi dovrà essere trasmesso  $Q-1$  volte. Essendo il tempo richiesto dallo store and forward  $L/R$ , il tempo totale sarà  **$Q(L/R)$** .

**Frammentazione del messaggio** In una moderna rete a commutazione di pacchetto, la sorgente frammenta i lunghi messaggi dello strato di applicazione in pacchetti più piccoli e invia questi ultimi nella rete. Sebbene la frammentazione complichi la vita di sorgente e destinatario, si è concluso che i vantaggi compensano grandemente gli svantaggi. Diciamo che una rete a commutazione di pacchetto effettua una commutazione di messaggio se le sorgenti non frammentano i messaggi. Quando invece un messaggio viene segmentato in pacchetti, si dice che la rete effettua in pipeline la trasmissione dei messaggi, cioè parti del messaggio vengono trasmesse in parallelo dalla sorgente e dai commutatori di pacchetto. Un vantaggio della commutazione di pacchetto con segmentazione è che il ritardo della connessione end-to-end è molto ridotto. Con le immagini accanto si può capire perché i tempi siano

più rapidi, è infatti il vantaggio della pipeline.

Per fare un esempio, consideriamo un messaggio lungo  $7,5 * 10^6$  bit. Supponiamo che tra i due host ci siano due commutatori di pacchetto e tre link, ciascun link abbia una velocità di trasmissione di 1,5 Mbit/s ( $1,5 * 10^6$  bit/s). Assumendo che la rete non sia in congestione, quanto tempo è richiesto per trasferire il messaggio con la **commutazione di messaggio**? Alla sorgente occorrono 5 secondi ( $7,5 * 10^6$  bit /  $1,5 * 10^6$  bit/s) per portare il messaggio al primo commutatore. Poiché i commutatori sono di tipo store-and-forward, il primo commutatore deve aspettare tutta la ricezione del pacchetto. Quindi questa procedura si ripete anche tra i commutatori, quindi abbiamo ( $7,5 * 10^6$  bit /  $1,5 * 10^6$  bit/s) \* 3 link.

Ora frammentiamo il messaggio in 5000 pacchetti da 1500 bit l'uno. Assumendo che non ci sia congestione, quanto ci metteremo con la **commutazione di pacchetto**? Occorre 1 ms per spostare il primo pacchetto al primo commutatore ( $1,5 * 10^3$  bit /  $1,5 * 10^6$  bit/s), poi 1 ms per ogni link, quindi il primo pacchetto arriverà dopo 3 ms a destinazione. Ora, l'ultimo pacchetto quanto ci metterà? Consideriamo che il secondo pacchetto viene spedito in contemporanea mentre il primo è sul secondo link. Secondo questa logica, l'ultimo pacchetto arriva al primo commutatore dopo  $5000 * 1\text{ms} = 5000\text{ms} = 5\text{s}$ , quindi dovrà attraversare gli altri due link. Il tempo finale sarà, quindi  $5002\text{ ms} = 5,002$  secondi contro i 15 precedenti.

Quindi, la formula è:

$$d = \frac{\text{dim\_pacchetto}}{\text{vel\_trasferimento}}$$

$$\text{tempo} = d * \text{tot\_pacchetti} + (\text{link} - 1) * d$$

Questo miglioramento deriva dal fatto che la commutazione di pacchetto agisce in parallelo. Il vantaggio della segmentazione è anche che nel caso ci sia un errore di bit su di un pacchetto, è il singolo pacchetto a poter essere scartato e non l'intero messaggio. La frammentazione non è priva di svantaggi, infatti al pacchetto devono essere aggiunte informazioni nell'**intestazione (header)** e queste possono comprendere l'identità di sorgente e destinatario. In più l'header richiede altri byte di spazio.

## 1.4 Accesso alla rete e mezzi trasmissivi

### 1.4.1 Accesso alla rete

L'accesso può essere classificato in tre categorie:

- Accesso domestico
- Accesso aziendale
- Accesso per terminali mobili

Queste categorie, però, non sono rigide e vincolanti.

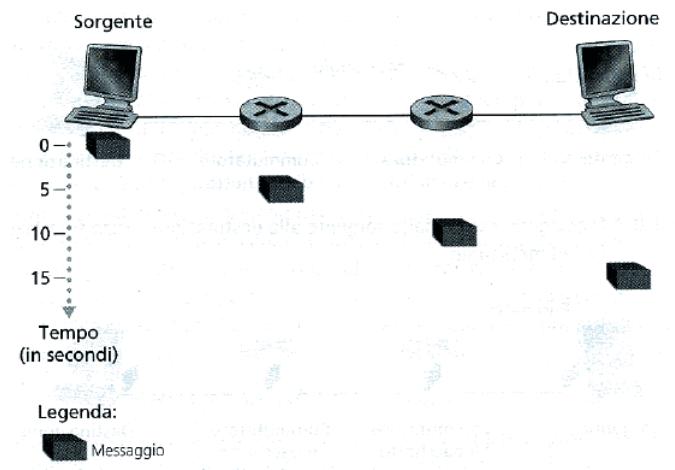


Figura 3: Tempi per il trasferimento del messaggio senza frammentazione dello stesso

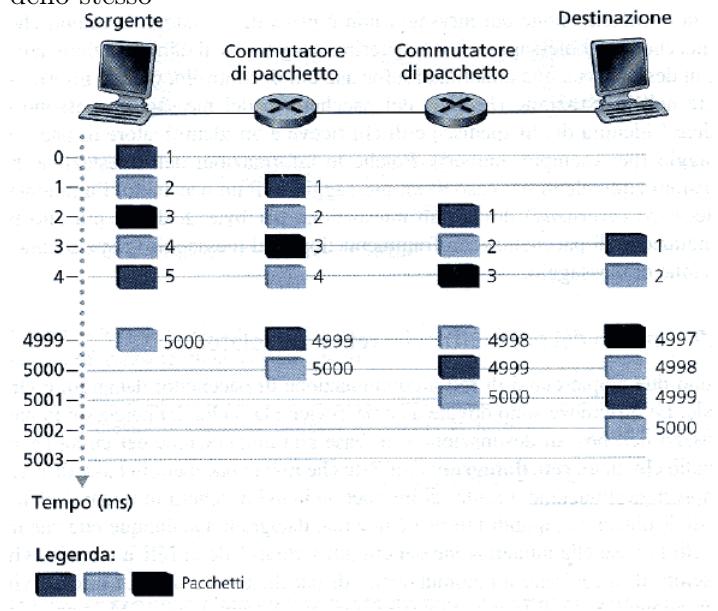


Figura 4: Tempi per il trasferimento del messaggio quando lo stesso è frammentato in 5000 pacchetti

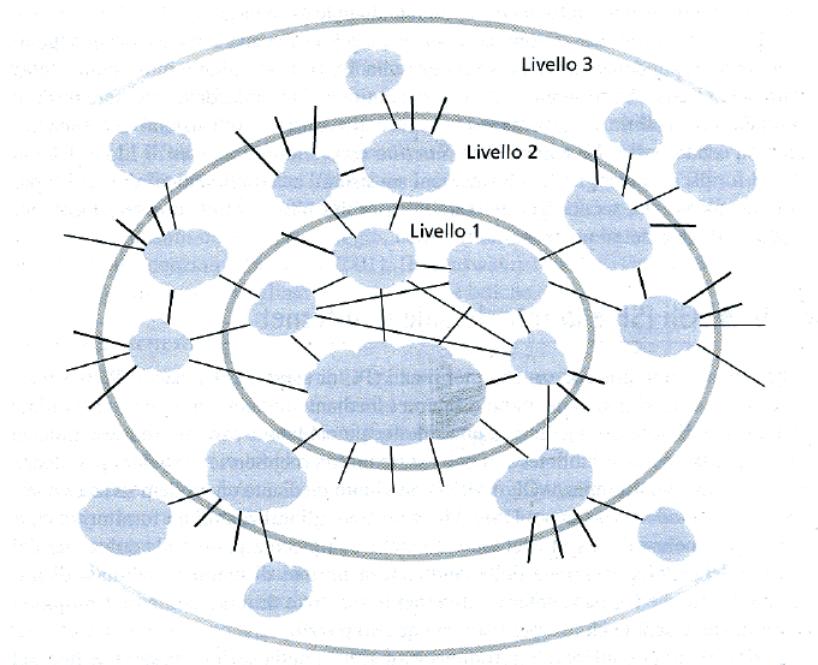


Figura 5: Interconnessione degli ISP

#### 1.4.2 Mezzi fisici

Si dividono in due categorie:

- Guidati: guidati attraverso un mezzo solido
- Non guidati: si propagano nell'atmosfera

### 1.5 Gli ISP e la rete dorsale di internet

Le reti di accesso situate nella sezione esterna di internet sono connesse al resto di internet attraverso una gerarchia a livelli di fornitori di servizi (ISP):

- Livello 1: Rete dorsale di internet
- Livello 2: Copertura tipicamente regionale o nazionale, connesso a pochi ISP di livello 1.

Gli ISP sono in rapporto di utente-fornitore. Quando due ISP sono collegati direttamente tra di loro sono detti “pari” tra loro. I punti di collegamento tra i vari ISP sono detti “punti di presenza” (POP). Gli ISP spesso si connettono in Network Access Point (NAP).

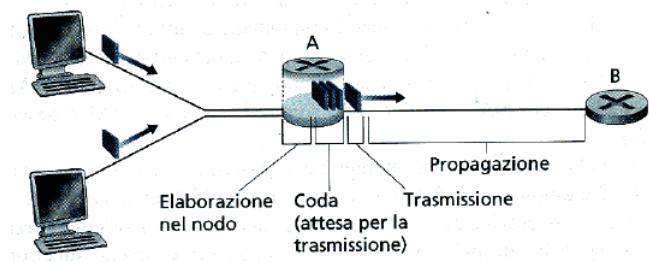


Figura 6: Ritardo al nodo e nel router A

## 1.6 Ritardi e perdite nelle reti a commutazione di pacchetto

### 1.6.1 Tipi di ritardo

**Ritardo di elaborazione** Tempo richiesto per esaminare l'intestazione del pacchetto e per determinare dove instradarlo. Può comprendere altri fattori come il tempo per il controllo degli errori. Dopo quest'elaborazione, il router invia il pacchetto alla coda che precede il link diretto al router B.

**Ritardo in coda** È il tempo di attesa prima dell'instradamento verso B. Dipende dal numero di pacchetti già in coda.

**Ritardo di trasmissione** Sia la lunghezza del pacchetto  $L$  bit e la velocità di trasmissione tra i router A e B  $R$  bit/s. Il ritardo di trasmissione (detto anche *ritardo store-and-forward*) è  $L/R$ , ovvero l'ammontare del tempo richiesto per trasmettere tutti i bit del pacchetto nel link. Il ritardo di trasmissione è tipicamente dell'ordine dei microsecondi ai millisecondi.

**Ritardo di propagazione** Il tempo richiesto per arrivare dall'inizio del link al router B è il ritardo di propagazione, dipende dalla velocità di propagazione del link. Si calcola come la distanza fra due router diviso la velocità di propagazione del mezzo, ovvero il ritardo di propagazione è  $d/s$ , dove  $d$  è la distanza fra i router A e B e  $s$  è la velocità di propagazione del link.

**Confronto tra ritardo di trasmissione e ritardo di propagazione** Il ritardo di trasmissione è il tempo richiesto dal router per spingere all'esterno il pacchetto; è funzione della lunghezza del pacchetto e della velocità di trasmissione del link, ma non ha nulla a che fare con la distanza fra due router.

Il ritardo di propagazione è il tempo che impiega un bit a propagarsi da un router al successivo; è funzione della distanza fra due router, ma non ha nulla a che vedere con la lunghezza del pacchetto o la velocità di trasmissione del link.

Se indichiamo con  $d_{elab}$ ,  $d_{coda}$ ,  $d_{trans}$ ,  $d_{prop}$  i ritardi di elaborazione, di coda, di trasmissione e di propagazione, il ritardo totale del nodo è dato da:

$$d_{elab} = d_{elab} + d_{coda} + d_{trans} + d_{prop}$$

### 1.6.2 Ritardo di coda e perdita di pacchetti

La più complicata e importante componente del ritardo totale del nodo è il ritardo di coda. A differenza degli altri ritardi, il ritardo di coda può variare da pacchetto a pacchetto. Se ci sono 10 pacchetti, il primo non avrà ritardo di coda, l'ultimo invece avrà un ritardo relativamente grande.

Quand'è consistente e quando insignificante il ritardo di coda? Dipende molto da altri fattori come velocità del link, congestione e distribuzione del traffico. Indichiamo con  $a$  la velocità media di arrivo dei pacchetti (l'unità è pacchetti/s),  $R$  è la velocità di trasmissione (bit/s) e i pacchetti sono costituiti da  $L$  bit. Perciò, la velocità media a cui i bit arrivano ad accordarsi è  $Labit/s$ . Il rapporto **La/R**, detto **intensità di traffico**, ha un ruolo per la stima del ritardo di coda:

- $>1$ : arrivano più velocemente di quanto se ne vadano, in questo caso la coda continua ad aumentare
- $\leq 1$ : la natura del traffico in arrivo influenza il ritardo di coda, ovvero se arrivano periodicamente, allora la coda non farà in tempo ad accumularsi. Se invece arrivassero a raffiche, ma periodicamente, la media del ritardo sarà significativa.

**Perdita di un pacchetto** Quando un pacchetto arriva su una coda piena, il pacchetto è scartato e perso. In questo caso può essere ritrasmesso dal nodo precedente, dal terminale o essere perso definitivamente.

## 1.7 Strati protocollari

### 1.7.1 Architettura stratificata

Per ridurre la complessità progettuale, i progettisti della rete organizzano i protocolli a strati (layer) o livelli.

Con un'architettura a strati dei protocolli, ciascun controllo appartiene a uno degli strati. Ciascun protocollo appartiene a uno degli strati, quindi il protocollo nello strato specifico è condiviso tra tutte le entità della rete che condividono quel protocollo. Queste entità comunicano tra di loro scambiandosi i messaggi dello strato n. Questi messaggi sono chiamati **n-PDU (layer-n Protocol Data Units)**. Il formato di una n-PDU è definito dal protocollo dello strato n. Quando presi nel loro insieme, i protocolli dei vari strati sono chiamati **pila protocollare**.

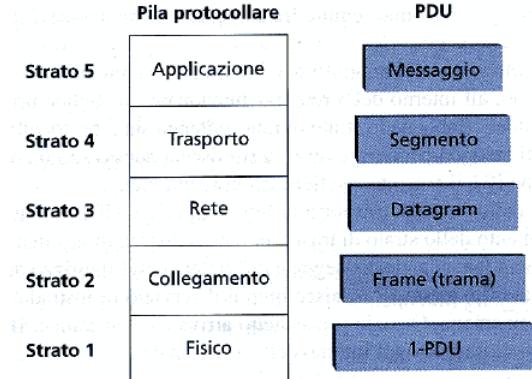


Figura 7: Protocollo a strati, livello = PDU

Un concetto chiave è quello di **modello di servizio** di uno strato: si dice che lo strato

$$n - 1$$

offre servizi allo strato n.

**Funzione degli strati** Ciascuno strato può eseguire uno o più di questi compiti:

- Controllo dell'errore
- Controllo del flusso
- Frammentazione e riassemblaggio
- Multiplexing
- Instaurazione della connessione

### 1.7.2 La pila protocolare di internet

È costituita da cinque strati

**Strato di applicazione** È responsabile del supporto delle applicazioni della rete, comprende molti protocolli tra i quali http, SMTP e FTP.

**Strato di trasporto** Vi risiedono i due protocolli, TCP e UDP:

- TCP è un servizio orientato alla connessione con garanzia di consegna e un controllo di flusso (ovvero di adattamento tra la velocità del mittente e del destinatario)
- UDP fornisce un servizio senza connessione per applicazioni che possono accettare una perdita di pacchetti (ad esempio uno stream video)

**Strato di rete** È responsabile dell’instradamento dei datagram da un host all’altro. Contiene il protocollo IP, utilizzato da tutti i componenti di internet che hanno uno strato di rete.

I protocolli dello strato di trasporto (TCP e UDP) in un host sorgente passano un segmento dello strato di trasporto e un indirizzo di destinazione allo strato IP.

**Strato di collegamento** Per muovere un pacchetto da un nodo al successivo sul percorso, lo strato di rete deve delegare il servizio allo strato di collegamento. In particolare, a ciascun nodo IP passa il datagram allo strato di collegamento, che lo invia al nodo successivo lungo il percorso.

**Strato fisico** Il suo compito è quello di muovere singoli bit all’interno della rete da un nodo all’altro.

## 2 Strato di trasporto

### 2.1 Introduzione e servizio dello strato di trasporto

Un protocollo dello strato di trasporto fornisce una commutazione logica fra i processi applicativi che funzionano su host differenti. Per comunicazione logica intendiamo che dal punto di vista dell'applicazione, è come se i terminali su cui girano i processi fossero direttamente connessi. I processi applicativi usano la comunicazione logica fornita dallo strato di trasporto per scambiarsi messaggi, senza doversi occupare dei dettagli dell'infrastruttura fisica usata per trasportarli.

**Attenzione:** i protocolli dello strato di trasporto sono implementati nei terminali ma non nei router della rete.

#### 2.1.1 Relazione fra gli strati di trasporto e di rete

Mentre un protocollo dello strato di trasporto fornisce una *comunicazione logica tra i processi* che funzionano su differenti host, un protocollo dello strato di rete fornisce la *comunicazione logica fra gli host*.

#### 2.1.2 Panoramica dello strato di trasporto in internet

Per semplificare la terminologia, nel contesto di internet, ci riferiamo alla 4-PDU come a un segmento.

**Attenzione:** nella letteratura ci si riferisce alla PDU per TCP come a un segmento, al PDU per UDP come a un datagram. In questo testo, però, si utilizzerà solo la notazione "segmento".

Il protocollo dello strato di rete ha un nome: **IP** (Internet Protocol). L'IP fornisce la comunicazione logica fra gli host. Il modello di servizio di IP è un **servizio best effort**, ovvero che "fa del suo meglio" per consegnare i segmenti fra i due host ma senza garanzie; per questo motivo IP è un **servizio inaffidabile**. Ricordiamo che **ciascun host ha un unico indirizzo IP**.

La maggior responsabilità di UDP e TCP è di estendere il servizio di spedizione di IP tra due terminali al servizio di spedizione fra due processi che funzionano sui terminali. L'estensione della spedizione da host a host alla spedizione da processo a processo è detta **multiplexing** e **demultiplexing** dello strato di trasporto. UDP e TCP forniscono anche un controllo dell'integrità inserendo campi di rilevamento di errori nelle loro intestazioni.

Questi due servizi, **spedizioni di dati da processo a processo** e **verifica degli errori** sono i due soli servizi forniti da UDP, infatti UDP è un servizio

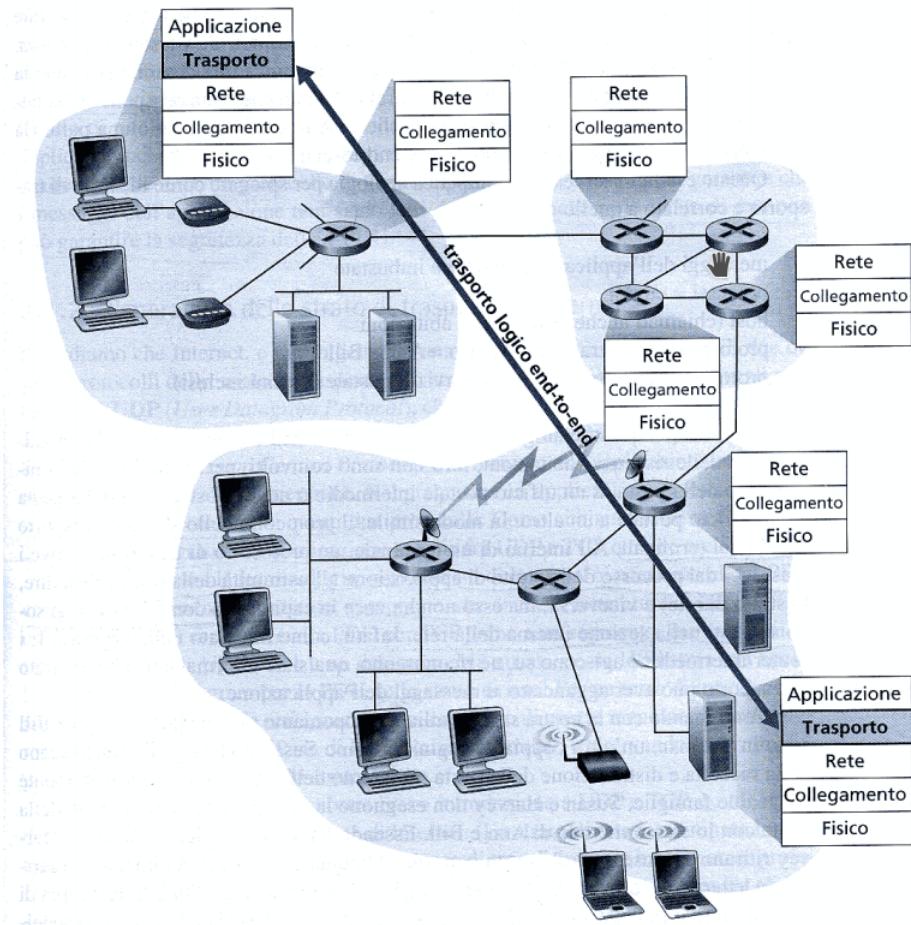


Figura 8: Lo strato di trasporto fornisce una comunicazione logica piuttosto che fisica tra le applicazioni

inaffidabile.

TCP offre molti servizi addizionali; prima di tutto un **trasferimento affidabile dei dati** usando controllo del flusso, numeri di sequenza, riscontri (acknowledgment) e timer, in questo modo TCP si assicura che i dati siano spediti da un processo mittente al destinatario correttamente e in ordine. TCP converte perciò il servizio inaffidabile IP in un servizio affidabile.

TCP usa anche il controllo di congestione, ovvero previene la saturazione da parte di qualsiasi connessione TCP della rete suddividendo equamente la banda di un link congestionato tra le connessioni TCP che lo attraversano. I campi dei numeri di porta sorgente e destinazione in un segmento dello strato di trasporto sanno. Il traffico UDP, invece, non è regolabile.

## 2.2 Multiplexing e demultiplexing

Per l'host destinatario, lo strato di trasporto riceve i segmenti (le PDU dello strato di trasporto) allo strato di rete posto subito sotto di esso. Lo strato di trasporto ha la responsabilità di inviare i dati di questi segmenti all'appropriato processo applicativo che funziona sull'host. Per comprendere come funzioni ricordiamoci prima di tutto che un processo ha un **socket**, che è una porta attraverso la quale i dati passano dalla rete al processo, e attraverso la quale i dati passano dal processo alla rete. Lo strato di trasporto nel terminale ricevente non consegna effettivamente i dati direttamente a un processo, **ma li consegna invece a un socket intermediario**. Possono esserci più socket e tutte hanno un identificatore unico. L'identificatore dipende dal fatto che il socket sia UDP o TCP.

Ogni segmento dello strato di trasporto ha un insieme di campi dedicati all'identificazione del socket; lo strato di trasporto esamina questi campi per determinare il socket ricevente e indirizzargli i segmenti. Il lavoro di recapitare i dati in un segmento allo strato di trasporto corretto socket è chiamato **demultiplexing**. Il lavoro di ottenere i dati dall'host sorgente dai diversi socket, completare i dati con le informazioni di intestazione (usate durante il demultiplexing) per creare segmenti, e di passare i segmenti allo strato di rete è detto **multiplexing**. Ogni segmento ha dei campi speciali che indicano il socket al quale il segmento deve essere consegnato. Questi campi speciali sono il **campo numero di porta sorgente** e il **campo numero di porta di destinazione**. Ciascun numero di porta è a **16 bit (0-65535)**. I numeri di porta che vanno da 0 a 1023 sono chiamati **numeri di porta ben conosciuti** e sono riservati, il che significa che sono dedicati per l'uso con protocolli applicativi noti come HTTP (80), FTP (21). L'elenco dei numeri di porta ben conosciuti è fornito nella RFC 1700.

Quando progettiamo una nuova applicazione dobbiamo assegnare un numero di porta.

Questo è il funzionamento di UDP, mentre TCP sfrutta sistemi più raffinati.

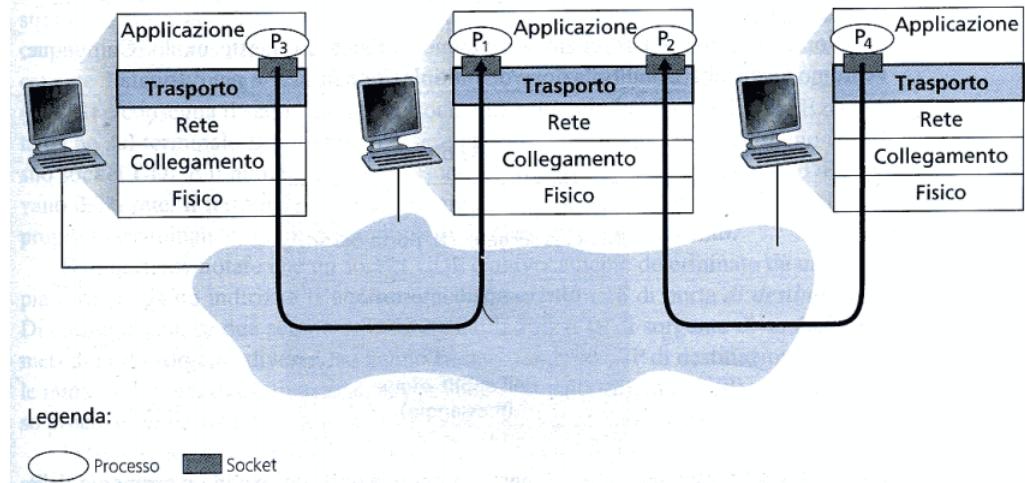


Figura 9: Multiplexing e demultiplexing dello strato di trasporto

Un socket UDP è univocamente determinato da una coppia formata da un indirizzo IP di destinazione e un numero di porta di destinazione.

### 2.2.1 Multiplazione e demultiplazione orientate alla connessione

Una sottile differenza tra un socket TCP e un socket UDP è che un socket TCP è identificato da una 4-upla:

- indirizzo IP di sorgente
- numero di porta sorgente
- indirizzo IP di destinazione
- numero di porta di destinazione

In particolare, e al contrario di UDP, due segmenti TCP entranti che recano indirizzi IP di sorgente diversi o numeri di porta sorgente diversi saranno diretti verso due diversi socket.

## 2.3 Trasporto senza connessione: UDP

L'UDP, definito nella RFC 768, esegue il minimo che un protocollo di trasporto può fare. Tranne che per la funzione di multiplexing/demultiplexing e qualche piccola verifica degli errori, esso aggiunge poco all'IP. Semplicemente aggiunge i numeri di porta di origine e destinazione, poi lo strato di rete incapsula i segmenti in un datagram IP e quindi poi li invia all'host di destinazione in modalità best effort.

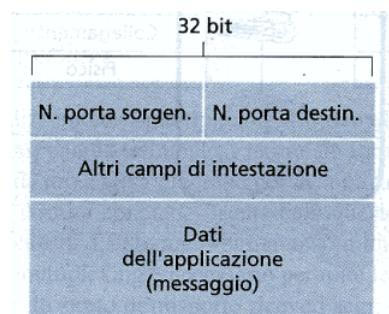


Figura 10: I campi dei numeri di porta sorgente e destinazione in un segmento dello strato di trasporto

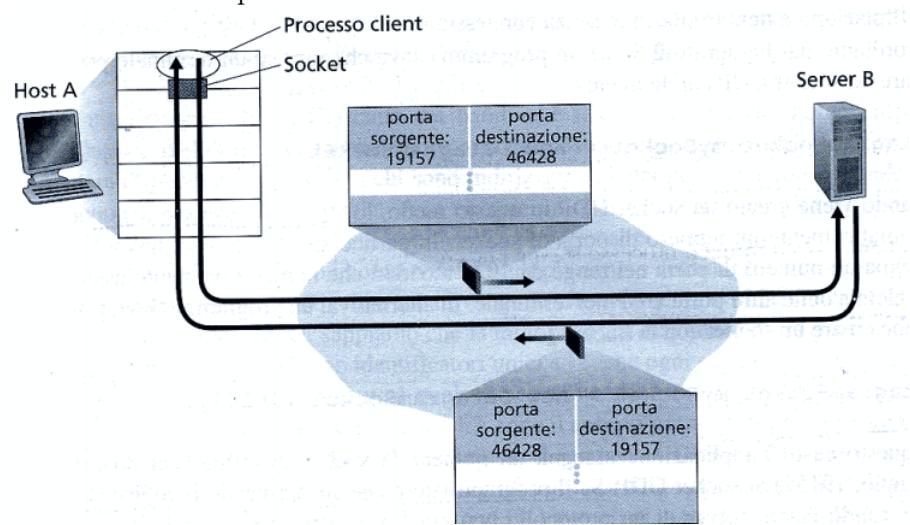


Figura 11: L'inversione dei numeri di porta sorgente e destinazione

**Attenzione:** con l'UDP, prima della spedizione del segmento, **non c'è handshake** fra le entità dello strato di trasporto che spediscono e ricevono. Per questo si dice che l'UDP è senza connessione.

Il DNS è un esempio di un protocollo dello strato di applicazione che usa l'UDP: quando l'applicazione DNS in un host vuole fare una richiesta, essa costruisce un messaggio di richiesta DNS e passa il messaggio a UDP. Infatti, se non riceve risposta, essa tenta ancora di inviare la richiesta a un altro server dei nomi, o informa l'applicazione che ha fatto la richiesta che gli è impossibile ottenere una risposta.

Perchè scegliere UDP? Per i seguenti motivi:

- **non viene creata alcuna connessione:** UDP non introduce alcun ritardo dovuto alla fase di impostazione della connessione (handshake)
- **nessuno stato della connessione:** l'UDP non mantiene lo stato della connessione e non ha traccia dei parametri di controllo della congestione e dei numeri di sequenza e riscontro, per questo un server può supportare molti più client attivi quando l'applicazione funziona su UDP invece che su TCP
- **Poco sovraccarico (*overhead*) dovuto all'intestazione del pacchetto:** il segmento TCP ha overhead di 20 byte per segmento, UDP solo 8
- **Controllo di livello applicativo più fine su quali dati vengono mandati e quando:** UDP, appena un processo applicativo manda dati a UDP, li impacchetta all'interno di un segmento e passa immediatamente il segmento allo strato di rete. TCP, invece, "strozza" il mittente quando uno o più link tra i terminali di sorgente e destinazione diventano eccessivamente congestionati. Inoltre TCP continua a rimandare un segmento fino a che non riceve l'acknowledgment, rallentando quindi la comunicazione

**Attenzione:** UDP può essere usato per generare un servizio affidabile, semplicemente i controlli di riscontro e ritrasmissione devono essere inseriti nell'applicazione stessa, cosa tediosa ma molto vantaggiosa per velocità di comunicazione e funzionalità. Molte applicazioni streaming sfruttano questo sistema.

### 2.3.1 Struttura del segmento UDP

La **checksum** (*somma di controllo*) è usata dall'host ricevente per controllare se sono stati introdotti errori nel segmento.

Applicazione	Protocollo dello strato dell'applicazione	Protocollo di trasporto adottato
Posta elettronica	SMTP	TCP
Accesso a terminale remoto	Telnet	TCP
Web	HTTP	TCP
Trasferimento di file	FTP	TCP
Server di file remoto	NFS	tipicamente UDP
Streaming multimediale	proprietario	tipicamente UDP
Telefonia Internet	proprietario	tipicamente UDP
Gestione della rete	SNMP	tipicamente UDP
Protocollo di routing	RIP	tipicamente UDP
Traduzione del nome	DNS	tipicamente UDP

Figura 12: Applicazioni diffuse in internet e protocolli che adottano

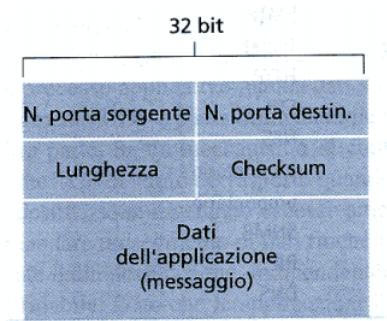


Figura 13: Struttura del segmento UDP

### 2.3.2 Checksum di UDP

La checksum di UDP effettua il rilevamento degli errori, ovvero se i bit del segmento sono stati alterati nel loro percorso.

- **Lato mittente:** si effettua il complemento a  $1^1$  della somma di tutte le parole a 16 bit del segmento, scartando ogni overflow. Il risultato è inserito nel campo checksum del segmento UDP.
- **Lato ricevente:** riceve tutte le parole a 16 bit, inclusa la checksum. Se nel pacchetto non sono stati introdotti errori, la somma al ricevente deve essere 1111111111111111.

Alcune implementazioni dell'UDP scartano semplicemente il segmento danneggiato, altri lo passano all'applicazione con l'aggiunta di un'avvertenza.

## 2.4 Principi di un trasferimento affidabile dei dati

È compito del protocollo di **trasferimento affidabile dei dati** l'implementazione di quest'astrazione del servizio. La difficoltà di questo compito deriva dal fatto che lo strato sottostante il protocollo di trasferimento affidabile dei dati può essere inaffidabile. Ad esempio, TCP è un protocollo di trasferimento affidabile che è implementato sopra uno strato di rete (IP) inaffidabile da terminale a terminale.

Per questa trattazione assumeremo che lo strato di rete sia inaffidabile. Tratteremo, inoltre, solo il caso di trasferimento unidirezionale dei dati. Il caso bidirezionale (*full duplex*) dei dati non è più complicato ma è più noioso da spiegare.

### 2.4.1 Costruzione di un protocollo per il trasferimento affidabile dei dati

Ora analizzeremo una serie di protocolli di complessità crescente fino ad un perfetto protocollo di trasferimento affidabile dei dati.

**Trasferimento affidabile dei dati su un canale completamente affidabile: rdt1.0** Consideriamo il caso in cui il canale sottostante sia completamente affidabile. Le frecce delle due FSM<sup>2</sup> indicano il passaggio del protocollo da uno stato all'altro. Gli eventi che causano la transizione sono illustrati sopra la linea orizzontale e l'azione/i intraprese sono illustrate sotto

---

<sup>1</sup>Si ottiene convertendo tutti gli 0 in 1 e tutti gli 1 in 0

<sup>2</sup>Per comprendere completamente il funzionamento di una FSM leggere gli appunti di Linguaggi e computabilità

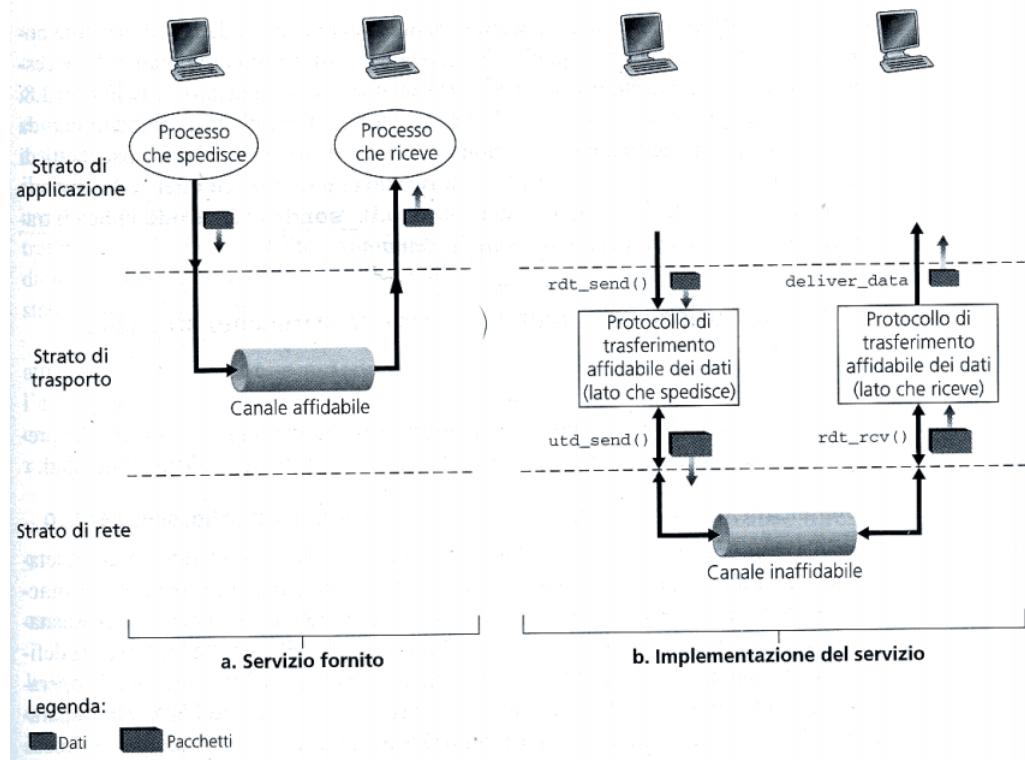


Figura 14: Astrazione di un servizio affidabile. rdt = reliable data transfer, udt = unreliable data transfer

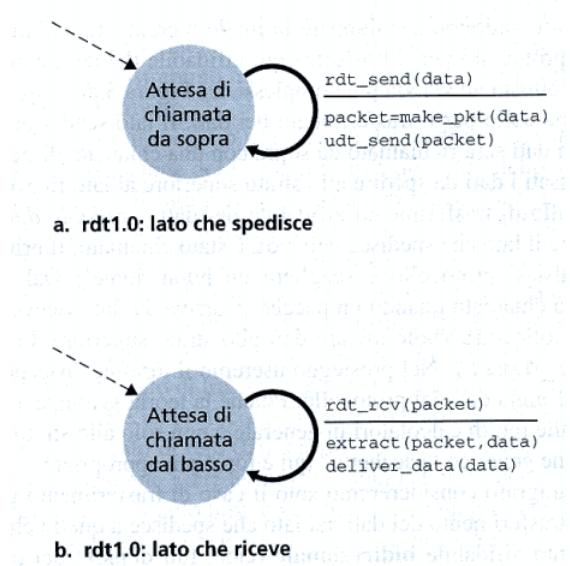


Figura 15: Macchine a stati finiti rdt1.0: un protocollo per un canale completamente affidabile (FSM - Finite State Machine)

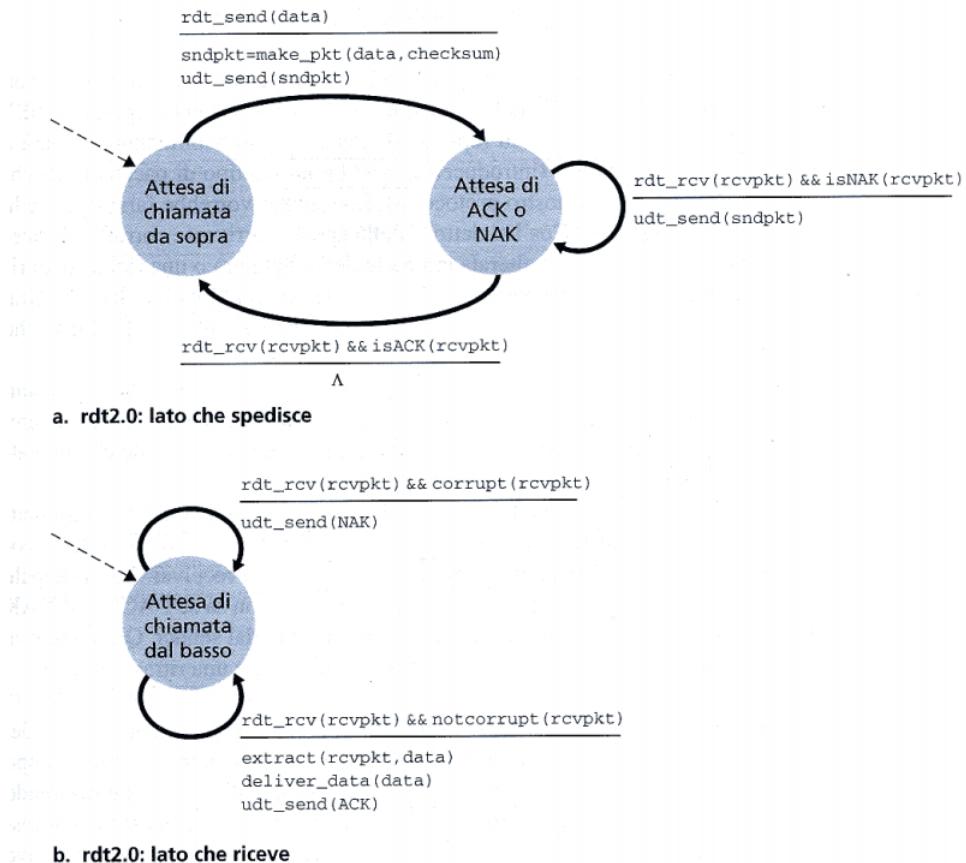


Figura 16: rdt2.0: un protocollo per un canale con errori sui bit

la linea. Quando non si verifica alcun evento o non si effettua un'azione si usa il simbolo  $\lambda$  (lambda). Lo stato iniziale è simboleggiato dalla freccia tratteggiata.

- **Lato mittente:** quando si ricevono i dati vengono inseriti in un pacchetto e viene spedito
  - **Lato ricevente:** quando viene ricevuto un pacchetto vengono estratti i dati e vengono inviati allo strato superiore

Con un canale completamente affidabile non serve alcun feedback da inviare al mittente.

**Trasferimento affidabile dei dati su un canale con errori sui bit:**  
**rdt2.0** Ora inseriamo la possibilità che i pacchetti possano essere alterati.  
Per ora assumiamo che tutti i pacchetti siano ricevuti.

I protocolli per il trasferimento affidabile dei dati che si basano sulle ri-trasmissioni sono conosciuti come **protocolli ARQ** (**A**utomatic **R**epeat **Q**uest). Fondamentalmente, in questi protocolli sono richieste tre funzionalità addizionali:

- **Rilevamento degli errori**
- **Feedback dal ricevente**: il ricevente invia un feedback esplicito al mittente, riscontri positivi (**ACK, positive acknowledgement**) e riscontri negativi (**NAK, Negative acknowledgement**)
- **Ritrasmissione**: un pacchetto che arriva con errori al ricevente sarà ritrasmesso dal mittente

Ora, vedendo la figura, notiamo che il mittente ha due stati, ovvero dopo aver spedito il pacchetto attende di ricevere l'ACK o il NAK. Se viene restituito un ACK, il mittente sa che il pacchetto più recente è stato ricevuto e allora il protocollo ritorna nello stato di attesa dei dati dallo strato superiore. Se riceve un NAK, allora ritrasmette l'ultimo pacchetto e attende un altro ACK o un NAK.

È importante notare che quando il mittente è nello stato di attesa di ACK o NAK non può accettare altri dati dallo strato superiore. Questo tipo di protocollo è conosciuto come **protocollo stop-and-wait** (*fermati e aspetta*).

Il lato ricevente ha ancora un singolo stato, semplicemente invierà un ACK o un NAK in base allo stato del pacchetto.

Ora, rimane un problema: i pacchetti ACK e NAK potrebbero essere alterati a loro volta. Come risolvere? Ci sono due possibilità:

- Aggiungere un numero di bit alla checksum sufficiente a permettere al ricevente non solo di rilevare, ma anche di correggere, eventuali errori dei bit.
- Il mittente può semplicemente reinviare i pacchetti quando riceve un pacchetto ACK o NAK difettoso. Questo modo introduce duplicati dei pacchetti. Il problema è che se il ricevente non sa se l'ACK o il NAK che ha inviato per ultimo è stato ricevuto correttamente dal mittente. Quindi non sa a priori se il pacchetto è nuovo o è una ritrasmissione.

Una semplice soluzione a questo problema è aggiungere un nuovo campo ai pacchetti dati: **un numero di sequenza** che il ricevente può semplicemente controllare per determinare se il pacchetto è nuovo o ritrasmesso.

Poichè abbiamo assunto che il canale non possa perdere pacchetti, i pacchetti non hanno bisogno di indicare a loro volta il numero di sequenza. Ora le FSM di mittente e ricevente hanno numero doppio degli stati rispetto a prima, questo perchè lo stato del protocollo deve adesso tenere in conto se il pacchetto attualmente in fase di spedizione o di ricezione dovrà avere numero di sequenza 0 o 1.

Il protocollo rdt2.1 usa ACK e NAK dal ricevente al mittente. Quando riceve un pacchetto fuori sequenza o alterato inviai un NAK. Possiamo ottenere lo stesso effetto di un NAK se viene inviato ogni volta un ACK col numero dell'ultimo pacchetto ricevuto correttamente. Un mittente che riceve due ACK con lo stesso numero di sequenza (ovvero **duplicati dell'ACK**) capisce che l'ultimo pacchetto non è andato a buon fine.

rdt2.2 introduce questa modifica, eliminando il NAK.

**Traferimento affidabile dei dati su un canale con perdite e con errori sui bit: rdt3.0** Supponiamo ora che il canale possa anche perdere i pacchetti, evento non raro. In questo caso ci sono due nuovi problemi:

- rilevare la perdita dei pacchetti
- cosa fare quando si perdono i pacchetti

L'uso di checksum, numeri di sequenza, pacchetti ACK e ritrasmissione ci permettono di risolvere l'ultima difficoltà, per la prima invece dobbiamo sviluppare nuovi sistemi.

Supponiamo che il mittente trasmetta un pacchetto di dati o che il ricevente invii un pacchetto o un riscontro e questi vengano persi. In entrambi i casi il mittente non riceve mai un riscontro. Se è disposto ad attendere abbastanza per essere sicuro che il pacchetto sia stato perso, allora può rispedirlo. La domanda ora è: per quanto deve attendere?

Il primo fattore è includere almeno un tempo equivalente al ritardo per un percorso circolare tra mittente e destinatario (che potrebbe comprendere il buffering ai router intermedi o ai gateway), più un certo ammontare di tempo richiesto dall'elaborazione di un pacchetto dal lato ricevente. La difficoltà nel definire questo ritardo porta l'eventualità della presenza di duplicati di pacchetti dati nel canale di comunicazione. rdt2.2 ha già introdotto sufficienti strumenti per la gestione dei duplicati.

Per implementare un meccanismo di ritrasmissione basato sul tempo, è necessario un **meccanismo di conto alla rovescia (*countdown timer*)** che possa interrompere il mittente dopo che è trascorso un certo tempo. Quindi, il mittente dovrà:

- avviare il timer ogni volta che un pacchetto viene spedito
- rispondere alle interruzioni del timer
- arrestare il timer

Ora, come può il mittente capire se l'ACK ricevuto riguarda l'ultimo pacchetto o un altro? Introducendo nel pacchetto di ACK un **campo di riscontro acknowledgement field**) che conterrà il numero di sequenza del pacchetto dati al quale corrisponde l'ACK. Come illustrato nella figura, poiché i numeri

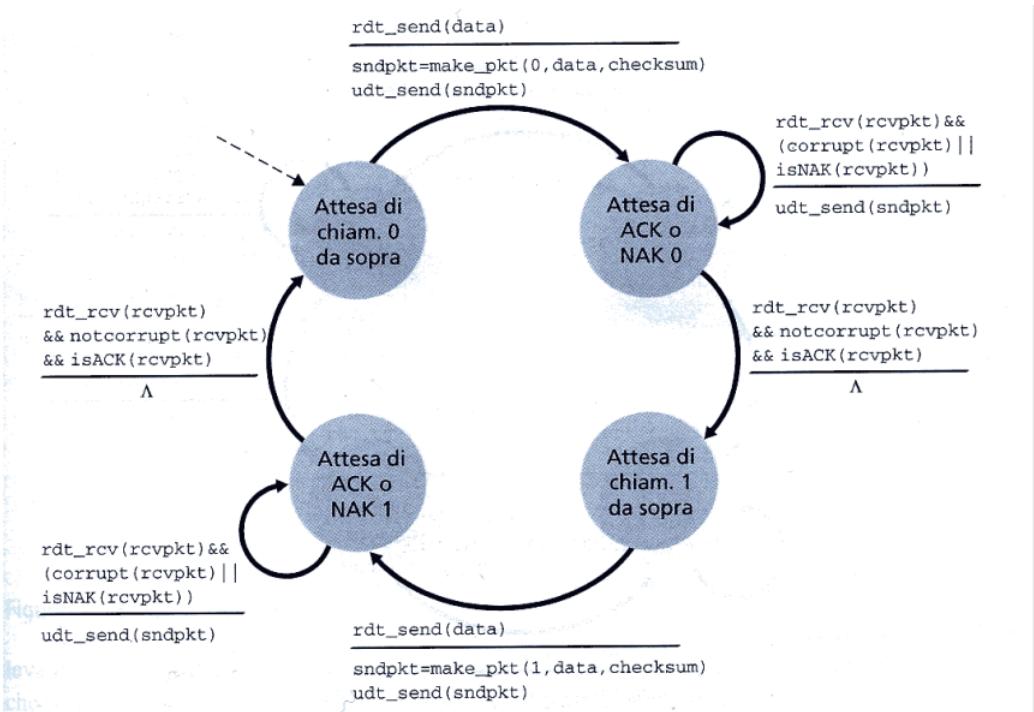


Figura 3.11 ◆ Sender rdt2.1.

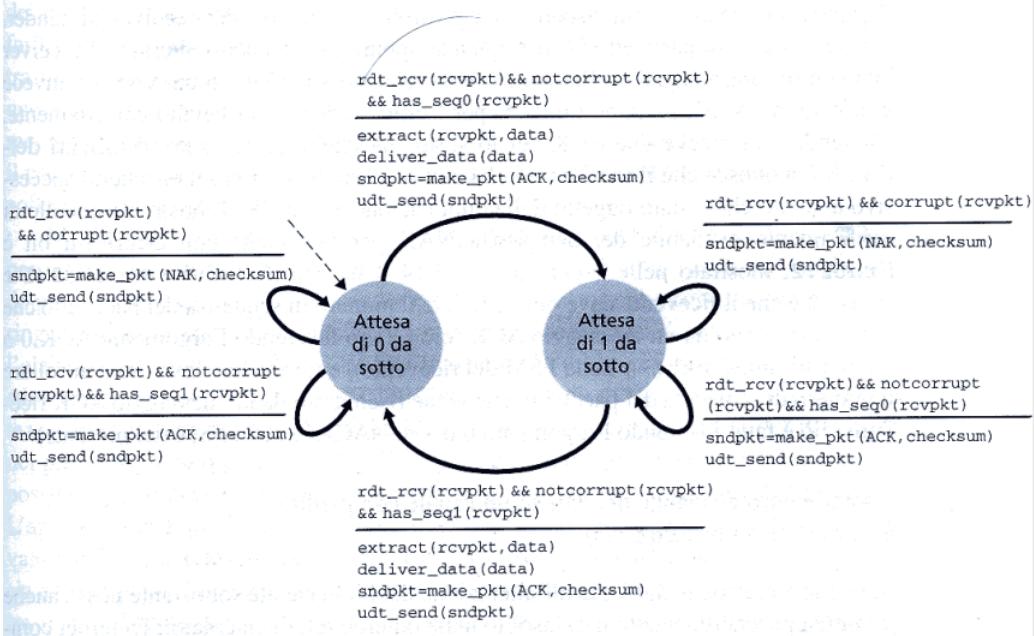


Figura 17: Sender e receiver 2.0

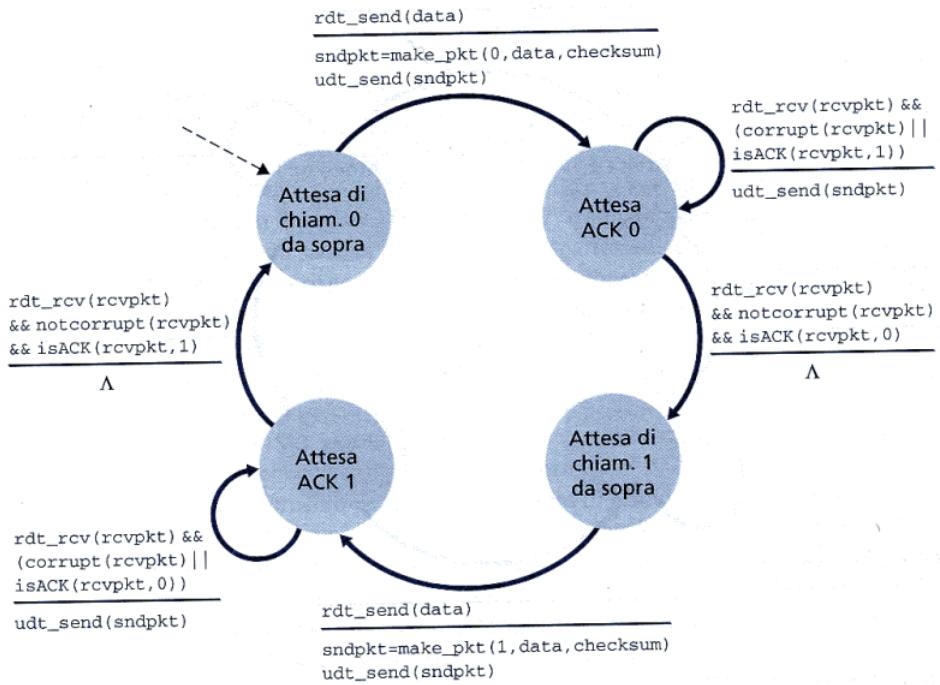


Figura 18: Sender 2.2

di sequenza si alternano tra 0 e 1, il protocollo edt3.0 qualche volta è detto **protocollo a bit alternati** (*alternating-bit protocol*).

Abbiamo assemblato un protocollo di trasferimento dei dati: checksum, numeri di sequenza, timer, ACK e NAK. Abbiamo ora un protocollo per il trasferimento affidabile dei dati che funziona!

#### 2.4.2 Protocolli pipeline per il trasferimento affidabile dei dati

Il problema principale di rdt3.0 è il fatto che è un protocollo stop-and-wait. Infatti, in un protocollo stop-and-wait il mittente è spesso in attesa (idle) poichè deve aspettare una risposta dal ricevente. Un modo per risolvere le attese troppo lunghe è permettere al mittente di inviare più pacchetti senza aspettare i riscontri. Questa tecnica è detta **pipelining**. Il pipelining ha molte conseguenze per i protocolli con trasferimento affidabile dei dati:

- La gamma dei numeri di sequenza deve essere aumentata per evitare ripetizioni
- i due host devono poter memorizzare più di un pacchetto.

la gamma di numeri di sequenza richiesti e i requisiti di buffering dipendono dal modo in cui il protocollo di trasferimento dei dati risponde alle perdite,

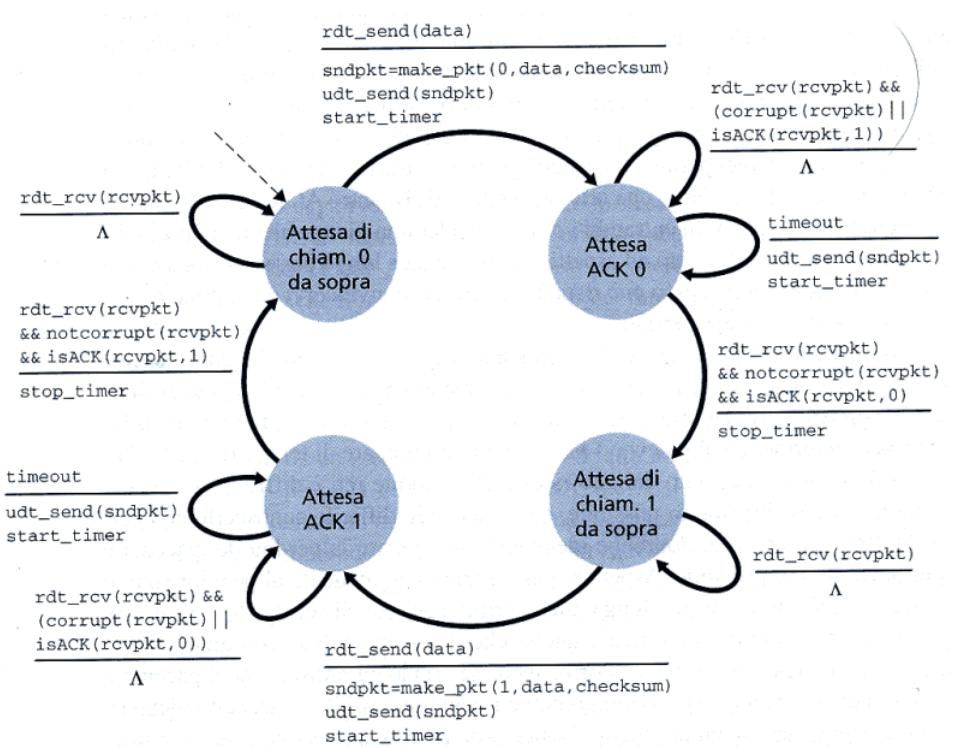


Figura 19: Sender rdt3.0

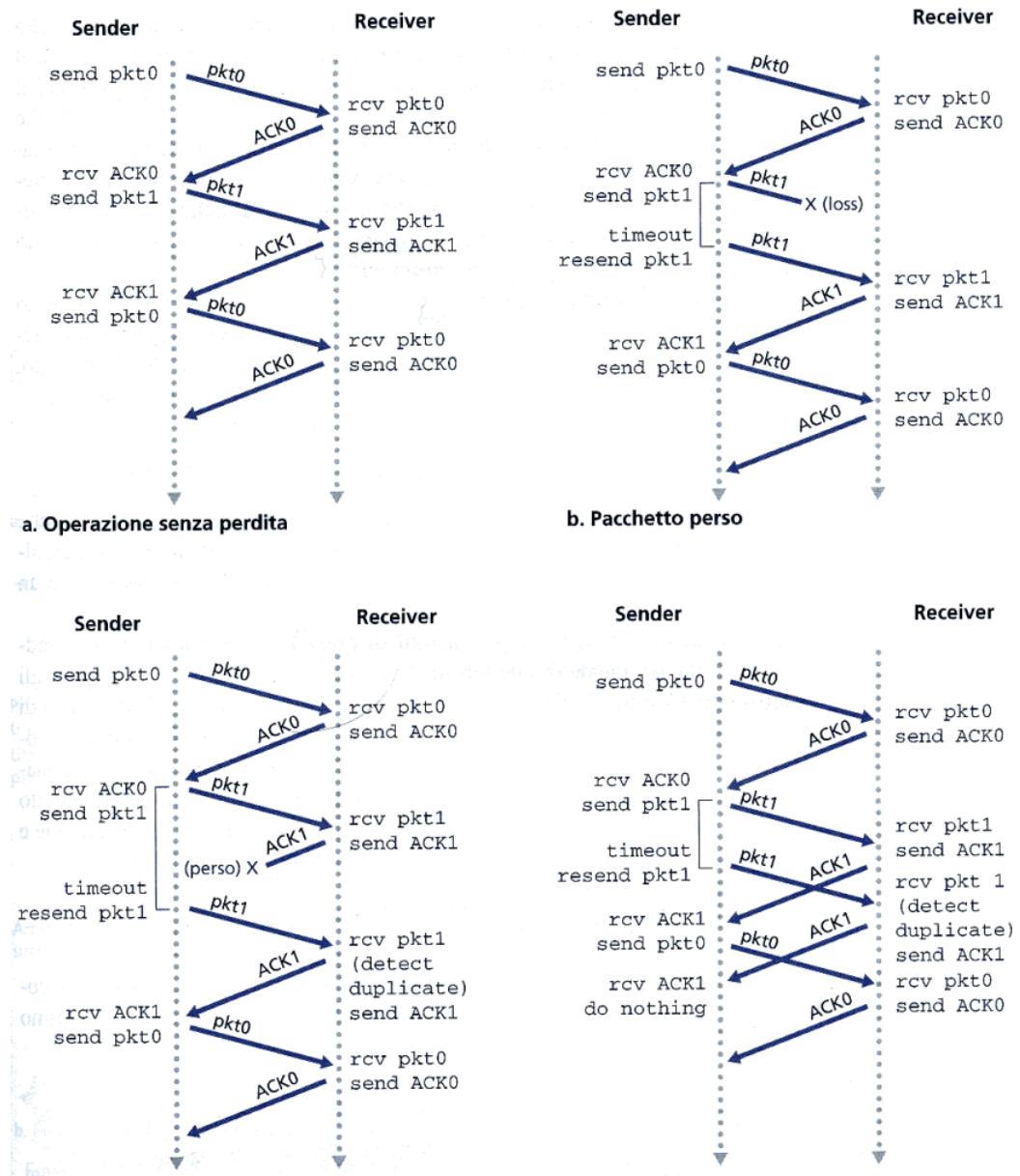


Figura 20: Operazioni dell'rdt3.0, il protocollo a bit alternati

all’alterazione e all’eccessivo ritardo dei pacchetti. Si possono identificare due approcci alla riparazione degli errori:

- Go-Back-n
- ripetizione selettiva

#### 2.4.3 Go-Back-N (GBN)

Il mittente può trasmettere pacchetti multipli senza aspettare il riscontro, ma è costretto ad avere non più di un certo numero massimo consentito di pacchetti non riscontrati,  $N$ , nella pipeline. Se definiamo come *base* il numero di sequenza del pacchetto più vecchio senza riscontro e come *nextseqnum* il più piccolo numero di sequenza inutilizzato, allora nella gamma dei numeri di sequenza si possono identificare quattro intervalli:

- $[0, \text{base}-1]$ : pacchetti già trasmessi
- $[\text{base}, \text{nextseqnum}-1]$ : pacchetti che sono stati spediti e ancora senza riscontro
- $[\text{nextseqnum}, \text{base}+N-1]$ : questi numeri di sequenza nell’intervallo possono essere usati per i pacchetti che possono essere spediti immediatamente
- $>\text{base}+N$ : non possono essere usati finché un pacchetto non riscontrato viene riscontrato

$N$  è noto come **dimensione della finestra** e il protocollo GBN come **protocollo a finestra scorrevole**. Il sender GBN deve affrontare tre tipi di eventi:

- Chiamata da sopra: quando  $\text{rdt}_s.end()$  è chiamata da sopra, il mittente prima controlla per valutare se la finestra è satura ( $N$  pacchetti in circolazione non riscontrati) e decide se spedire il nuovo pacchetto o se ritornare i dati allo strato superiore.
- ricezione di un ACK: un riscontro con numero di sequenza  $n$  sarà interpretato come un riscontro cumulativo che indica che tutti i pacchetti con un numero di sequenza fino a  $n$ ,  $n$  compreso, sono stati correttamente ricevuti
- un evento timeout: se interviene un timeout, il mittente rispedisce *tutti* i pacchetti che sono già stati spediti ma che non hanno ricevuto riscontro

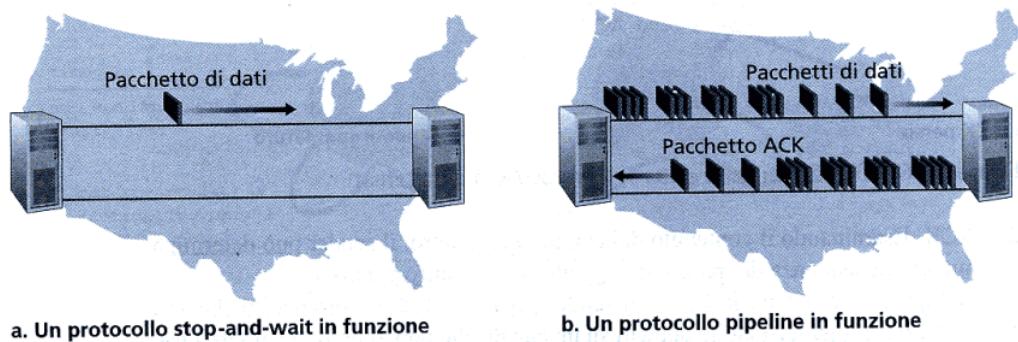


Figura 21: Confronto fra protocolli stop-and-wait e pipeline

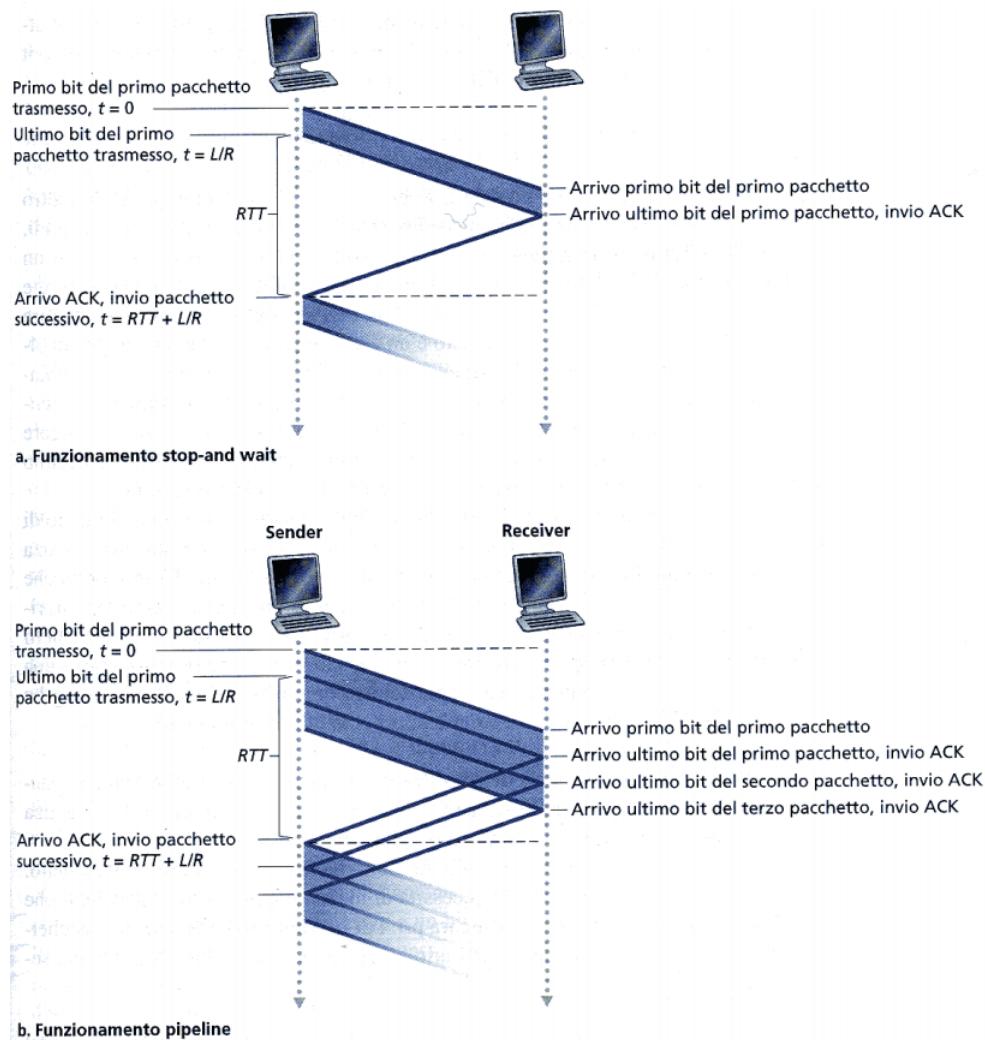


Figura 22: Spedizione stop-and-wait e pipeline

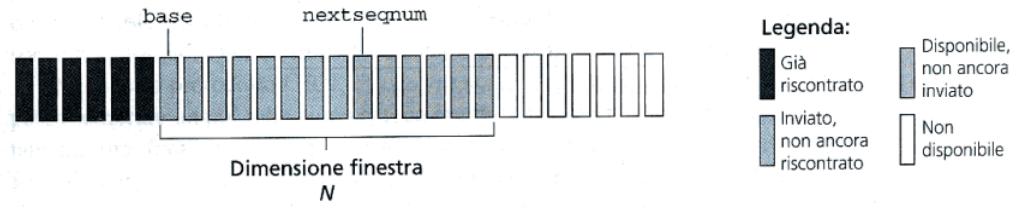


Figura 23: Il punto di vista del sender sui numeri di sequenza nel GBN

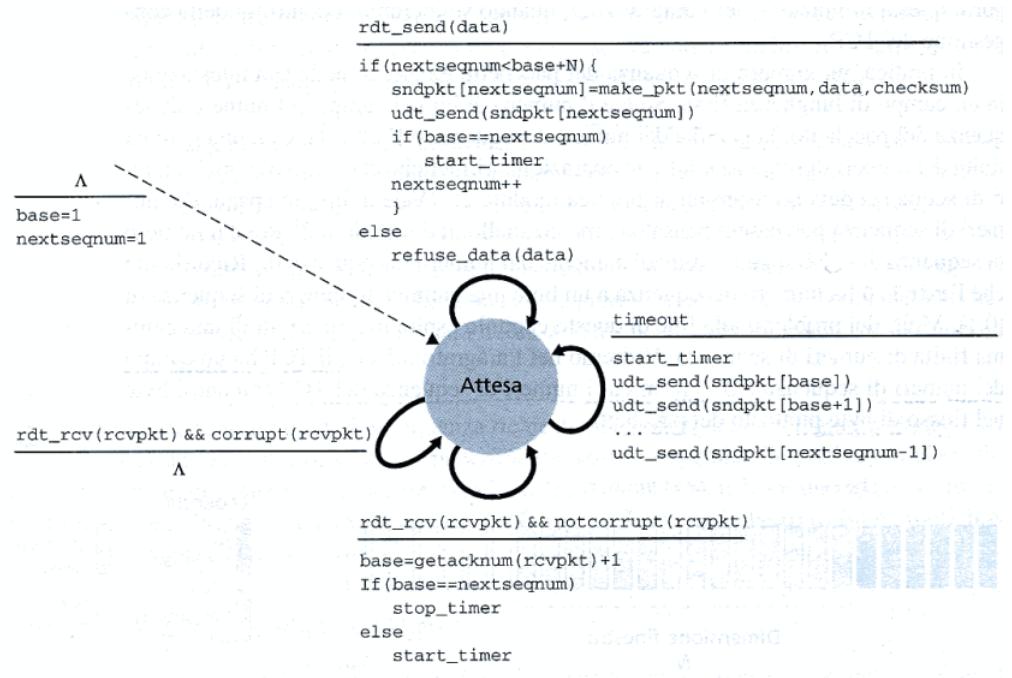


Figura 24: Descrizione dell'FSM estesa del sender GBN

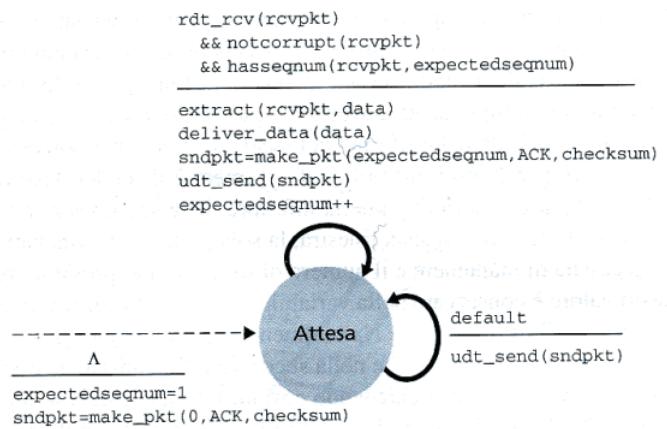


Figura 25: Descrizione dell'FSM estesa del receiver GBN

Le azioni del ricevente sono anch'esse semplici. Se un pacchetto con un numero di sequenza n è ricevuto correttamente ed è in ordine (ovvero se l'ultimo pacchetto inoltrato allo strato superiore è l' $n^*1$ ) il ricevente invia un ACK per il pacchetto n. Negli altri casi il pacchetto viene scartato e rispedisce un ACK per l'ultimo pacchetto in ordine. In questo protocollo GBN, il ricevente scarta i pacchetti non in ordine, questo permette di non dover mantenere in memoria alcun pacchetto fuori ordine. Ovviamente lo svantaggio è la richiesta di più ritrasmissioni eventuali.

#### 2.4.4 Ripetizione selettiva (SR)

Esistono casi nei quali GBN può avere problemi di prestazioni, ad esempio quando la dimensione della finestra e il prodotto ritardo-larghezza di banda sono entrambi grandi, nelle pipeline possono trovarsi molti pacchetti. Un singolo errore può costringere GBN a ritrasmettere un grande numero di pacchetti, magari non tutti necessari. In più queste ritrasmissioni possono saturare la pipeline.

I protocolli a ripetizione selettiva evitano le ritrasmissioni non necessarie grazie alla rispedizione di quei soli pacchetti che si sospetta siano giunti al ricevente con errori. Una finestra di dimensioni N dovrà ancora essere usata per limitare il numero di pacchetti da evadere, non riscontrati, nella pipeline. Il problema è identificare la dimensione necessaria della finestra affinché non ci siano errori.

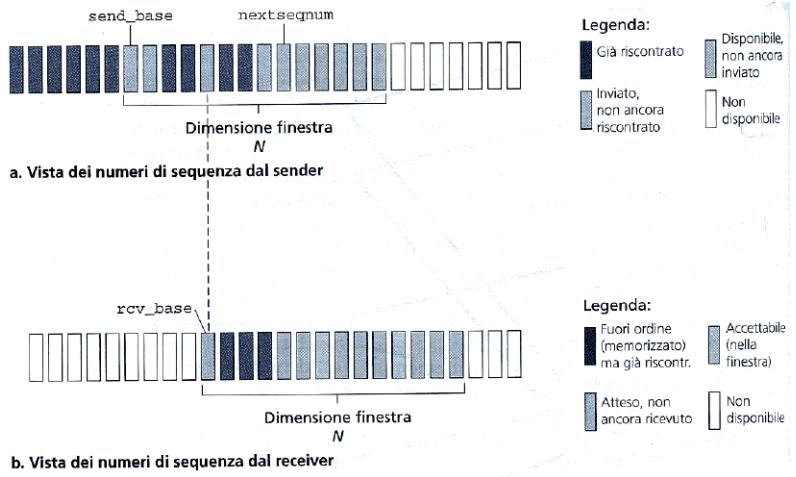


Figura 26: SR: viste degli intervalli dei numeri di sequenza dal sender e dal receiver

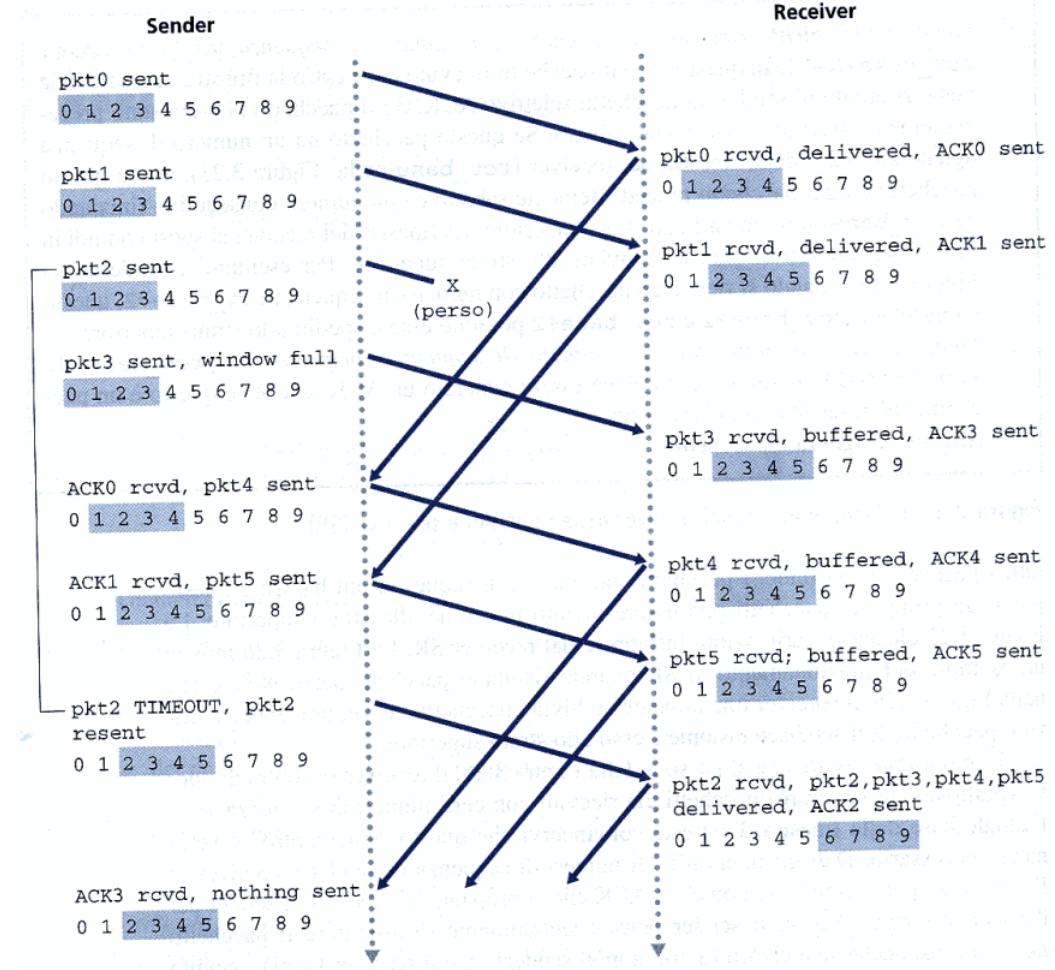


Figura 27: Operazioni del SR

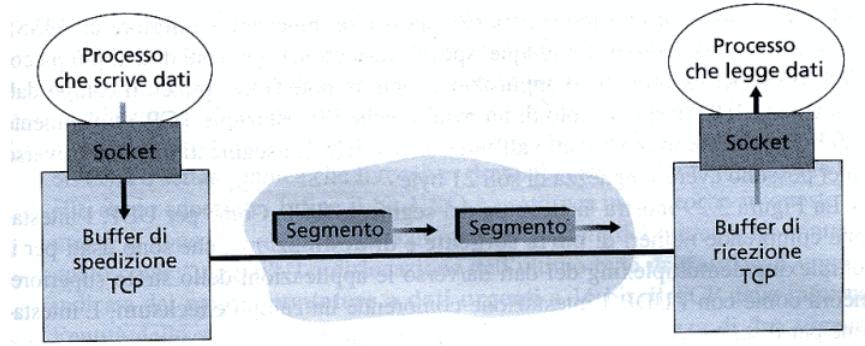


Figura 28: Buffer di spedizione e ricezione del TCP

## 2.5 Trasporto orientato alla connessione: TCP

### 2.5.1 La connessione TCP

Si dice che TCP è **orientato alla connessione** perché prima che un processo applicativo possa cominciare a spedire dati, i due processi devono scambiarsi un **handshake** (*stretta di mano*), ovvero devono prima inviarsi alcuni segmenti preliminari per stabilire i parametri del successivo trasferimento di dati. TCP funziona solo nei terminali e non negli elementi intermedi della rete, questi elementi non mantengono lo stato della connessione TCP. Infatti, i router intermedi dimenticano completamente la connessione TCP, vedono solo i datagram, non le connessioni.

Una connessione TCP fornisce un trasferimento dei dati **full duplex**, cioè consente ai dati di viaggiare contemporaneamente nelle due direzioni. Una connessione TCP è anche **point-to-point**, cioè fra un singolo sender e un singolo receiver. Con TCP il cosiddetto "*multicasting*" (il trasferimento da un mittente a più riceventi contemporaneamente) è impossibile.

**Handshake a tre vie:**

- il client invia uno speciale segmento TCP
- il server risponde con un secondo segmento speciale TCP
- il client risponde con un terzo segmento speciale

I primi due non contengono "carico utile" (payload), il terzo può trasportarne.

TCP indirizza i dati al **buffer di spedizione** (send buffer) della connessione, che è uno dei buffer che viene riservato durante l'iniziale handshake a tre vie. La quantità massima di dati che può essere prelevata e inserita in segmenti è limitata dalla **dimensione massima del segmento** (MSS, *Maximum Segment Size*). Il valore di MSS dipende dall'implementazione del TCP (determinata dal sistema operativo) e spesso può essere configurato.

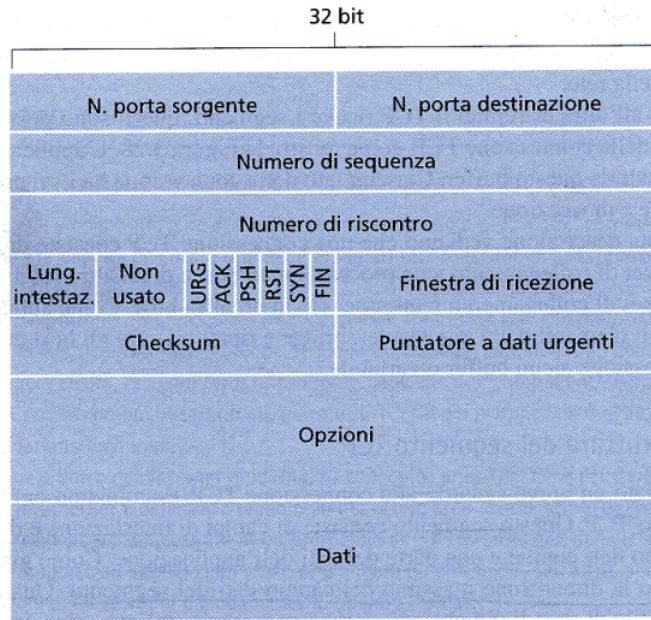


Figura 29: Struttura del segmento TCP

TCP unisce a ciascun pezzo dei dati del client un'intestazione TCP, formando così i segmenti TCP.

Quando il ricevente riceve i segmenti, questi sono posti nel buffer di ricezione della connessione TCP,

### 2.5.2 Struttura del segmento TCP

L'intestazione TCP è tipicamente di 20 byte. Come per UDP, l'intestazione comprende **numeri di porta sorgente e di destinazione** e **checksum**. Inoltre, l'intestazione TCP contiene anche:

- **numero di sequenza e numero di riscontro**: 32 bit
- **dimensione finestra**: 16 bit
- **lunghezza intestazione**: 4 bit, specifica la lunghezza dell'intestazione del TCP in parole a 32 bit
- **opzioni**: è a lunghezza variabile, è usato quando un sender e un receiver negoziano la massima dimensione del segmento MSS o come fattore di scala della finestra per l'uso nelle reti ad alta velocità
- **campo flag**: 6 bit
  - **ACK**

- **RST**
- **SYN**
- **FIN**
- **PSH**: indica che il receiver dovrebbe passare immediatamente i dati allo strato superiore
- **URG**: indica che in questo segmento ci sono dati che lo strato superiore ha definito come "urgenti". La dislocazione dell'ultimo byte di questi dati urgenti è indicata dal **campo puntatore a dati urgenti** a 16 bit

**Numeri di sequenza e numeri di riscontro** Il **numero di sequenza per un segmento** è il numero del primo byte del segmento.

Supponiamo di avere una serie di segmenti, tutti da 1000 byte. Per ogni segmento il numero di sequenza sarà il numero del primo byte, quindi 0, 1000, 2000, ecc.

Il **numero di riscontro** che l'host A inserisce nel suo segmento è il numero di sequenza del prossimo byte che l'host A aspetta dall'host B. Immaginando di aver ricevuto riscontro per i byte 0-535 e 900-1000, il numero di riscontro inviato sarà 536, ovvero il numero del primo byte non riscontrato. TCP riscontra solo i byte fino al primo mancante, si dice quindi che ha **riscontri cumulativi**.

**Impostazione e gestione dell'intervallo di timeout per le ritrasmissioni (con integrazioni per mancanza di pagine del libro)** Il **Round Trip Time (RTT)** nelle telecomunicazioni è il tempo che intercorre tra l'invio di un segnale più il tempo necessario per la ricezione della conferma di quel segnale.

Nelle reti, l'RTT è il tempo che passa da quando il segmento TCP viene inviato (ossia passa al livello di rete) a quando ritorna l'ACK del segmento stesso. Trascurando il tempo di trasmissione dell'ACK, viene calcolato come:

$$RTT = T_{tx} + 2T_p$$

dove  $T_{tx}$  è il tempo di trasmissione<sup>3</sup> e  $T_p$  è il tempo di propagazione<sup>4</sup>.

All'atto dell'invio di un pacchetto, il mittente registra il valore corrente del tempo locale, e quando riceve l'ACK registra nuovamente il valore temporale. Effettuando la sottrazione tra i due valori si ottiene una stima singola del RTT. Più stime possono essere combinate insieme per calcolare il RTT medio. Nel protocollo TCP viene stimato analizzando gli RTT dei segmenti non ritrasmessi secondo la seguente formula:

---

<sup>3</sup>Rapporto tra la dimensione del segmento e la velocità di trasmissione

<sup>4</sup>È il tempo necessario al segnale fisico per propagarsi lungo la linea di trasmissione fino al nodo successivo e da qui alla destinazione finale

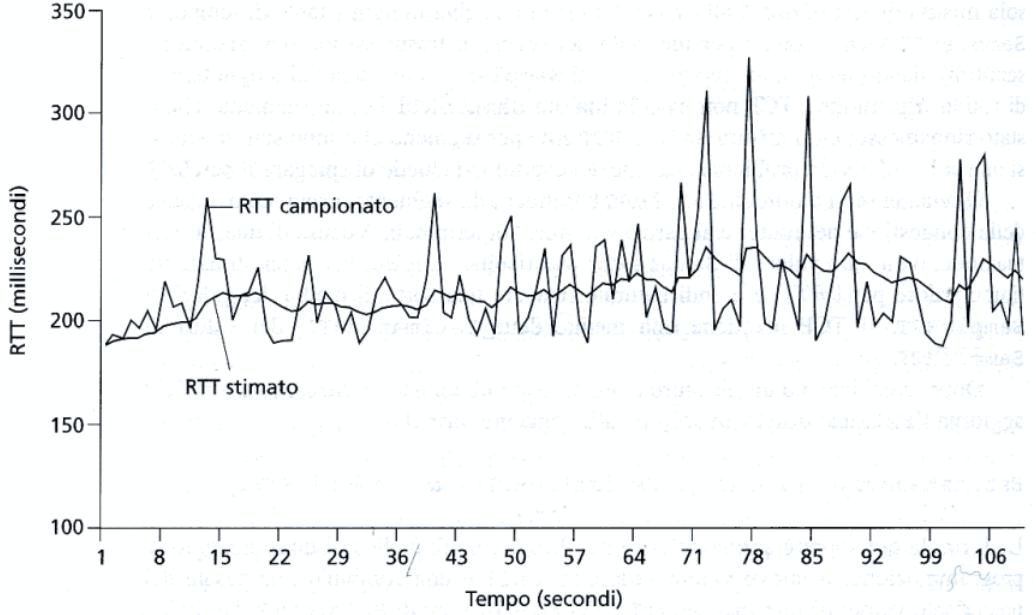


Figura 30: RTT campionati e RTT stimati

$$EstimatedRTT = (1 - \alpha)EstimatedRTT_{precedente} + \alpha SampleRTT^5$$

dove  $\alpha$  è posto a  $\frac{1}{8}$  in modo da modellare il valore degli RTT in base ai pacchetti più recenti, dando loro un peso esponenzialmente decrescente.

In realtà questo modo di calcolare l'RTT non tiene conto della varianza dei campioni di RTT. Un nuovo modo di calcolarlo è il seguente:

$$\begin{aligned} DIFF &= EstimatedRTT + SampleRTT \\ EstimatedRTT &= EstimatedRTT - (\delta * DIFF), 0 < \delta < 1 \end{aligned}$$

Dati i valori *EstimatedRTT* e *DevRTT*, che valore dovrebbe essere usato per l'intervallo di timeout di TCP? Chiaramente l'intervallo dovrebbe essere maggiore o uguale a *EstimatedRTT* ma non troppo maggiore. È consigliabile impostare il timeout pari a *EstimatedRTT* più un margine che dovrebbe essere grande quando ci sono fluttuazioni ampie nei valori di *SampleRTT*, piccolo in caso contrario. Il valore di *DevRTT* dovrebbe entrare in gioco qui.

$$\begin{aligned} DevRTT &= (1 - \beta) * DevRTT_{precedente} + \beta * |SampleRTT - EstimatedRTT| \\ TimeoutInterval &= EstimatedRTT + 4 * DevRTT \end{aligned}$$

### 2.5.3 Trasferimento affidabile dei dati

Ricordiamo che il servizio IP è inaffidabile.

Il TCP crea un **servizio di trasferimento affidabile dei dati** sopra al ser-

---

<sup>5</sup>EstimatedRTT: RTT stimato, SampleRTT: RTT campionato

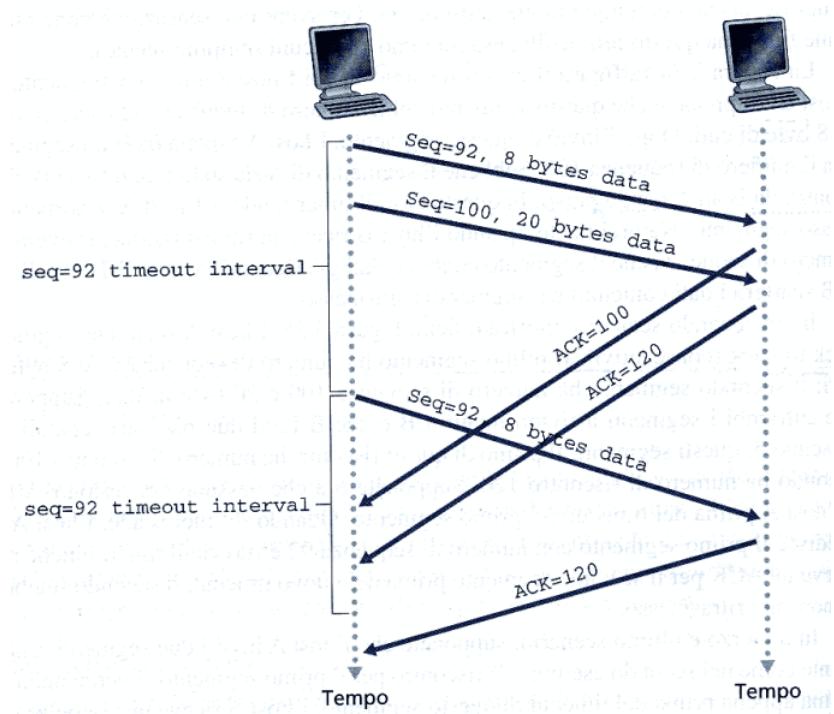


Figura 31: Il segmento 100 non è ritrasmesso

vizio inaffidabile best effort fornito da IP. Vediamo che ci sono tre principali eventi legati alla trasmissione e ritrasmissione dei dati nel mittente TCP:

- ricezione di dati dall'applicazione sovrastante
- timeout del timer: il timer viene avviato quando il segmento viene passato a IP e viene associato al più vecchio segmento non riscontrato. L'intervallo di scadenza per questo timer è il *TimeoutInterval*
- ricezione di un ACK

**Raddoppio dell'intervallo di timeout** In questa modifica, quando si verifica un timeout, il TCP ritrasmette il segmento non ancora riscontrato con il più piccolo numero di sequenza ma, ogni volta che il TCP ritrasmette, esso imposta il prossimo intervallo di timeout al doppio del valore precedente. Quindi gli intervalli crescono esponenzialmente dopo ogni ritrasmissione. Comunque, ogni volta che il timer viene riavviato dopo uno dei due altri eventi, il *TimeoutInterval* viene nuovamente derivato da *EstimatedRTT* e *DevRTT*.

Questa modifica fornisce una forma limitata di controllo della congestione.

Evento	Azione del receiver TCP
Arrivo di segmento in ordine con numero di sequenza atteso. Tutti i dati fino al numero di sequenza atteso sono già riscontrati. Nessun buco nei dati ricevuti.	ACK ritardato. Attesa fino a 500 ms dell'arrivo di un altro segmento in ordine. Se il successivo segmento in ordine non arriva in questo intervallo, invia un ACK.
Arrivo di segmento in ordine con numero di sequenza atteso. Un altro segmento in ordine in attesa della trasmmissione dell'ACK. Nessun buco nei dati ricevuti.	Invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti in ordine.
Arrivo di segmento fuori ordine con numero di sequenza più alto di quello atteso. Si rileva un buco.	Invia immediatamente un ACK duplicato, che indica il numero di sequenza del prossimo byte atteso.
Arrivo di segmento che chiude parzialmente o completamente i buchi nei dati ricevuti.	Invia immediatamente un ACK, posto che il segmento inizia all'estremità inferiore del buco.

Figura 32: Raccomandazioni per la generazione di ACK del TCP

**Ritrasmissione veloce** Quando il mittente TCP riceve tre duplicati ACK per gli stessi dati, prende questa informazione come conferma che il segmento successivo a quello riscontrato tre volte è andato perso. In questo caso, TCP esegue una **ritrasmissione veloce**, ritrasmettendo il segmento mancante prima che il timer di quel segmento scada.

**Go-Back-N o ripetizione selettiva?** Ricordiamo che i riscontri TCP sono cumulativi e che i segmenti ricevuti correttamente ma guari sequenza non sono riscontrati individualmente dal ricevente. Quindi, TCP deve ricordare solo il più piccolo numero di sequenza di un byte trasmesso ma non riscontrato. In questo senso **TCP assomiglia molto a un protocollo in stile GBN**. Ci sono però delle differenze. Infatti, una modifica proposta per il TCP, il cosiddetto **riscontro selettivo**, consente a un ricevente TCP di riscontrare selettivamente segmenti fuori sequenza piuttosto che riscontrare cumulativamente l'ultimo segmento ricevuto correttamente, in sequenza. Quindi, TCP è un ibrido tra i due sistemi.

#### 2.5.4 Controllo di flusso

Ricordiamo che gli host riservano un buffer di ricezione per la connessione. Quando la connessione TCP riceve byte che sono costretti e in sequenza, colloca i dati nel buffer di ricezione. Il processo dell'applicazione associato leggerà i dati da questo buffer. Se l'applicazione è relativamente lenta nella lettura dei dati, il mittente potrebbe inviare troppi dati e potrebbe saturare il buffer di ricezione.

Il TCP fornisce alle sue applicazioni un **servizio di controllo del flusso**,

ovvero un servizio di adattamento delle velocità. Questo controllo è detto **controllo della congestione**.

Il TCP fornisce il controllo di flusso attraverso il mantenimento nel mittente di una variabile detta **finestra di ricezione** (*receive window*). La finestra di ricezione è usata per dare al mittente un'idea di quanto spazio è disponibile nel buffer del ricevente. **La finestra di ricezione è dinamica**, ovvero varia durante la connessione.

Definiamo le seguenti variabili:

- $LastByteRead$ : numero dell'ultimo byte nel flusso di dati letto dal buffer dal processo dell'applicazione in B
- $LastByteRcvd$ : numero dell'ultimo byte nel flusso di dati che è arrivato dalla rete ed è stato collocato nel buffer di ricezione in B

Poichè al TCP non è permesso di saturare il buffer assegnato, dobbiamo avere:

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

La finestra di ricezione,  $RcvWindow$ , è posta uguale alla quantità di spazio disponibile nel buffer:

$$RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Poichè lo spazio disponibile cambia con il tempo,  $RcvWindow$  è dinamica. Come viene usata  $RcvWindow$ ? B inserisce in ogni segmento inviato ad A il valore attuale di  $RcvWindow$ , inizialmente impostando  $RcvWindow = RcvBuffer$ .

L'host A a sua volta mantiene traccia di due variabili:

- $LastByteSent$ : ultimo byte inviato
- $LastByteAcked$ : ultimo byte riscontrato

La differenza tra questi due è l'insieme dei byte non ancora riscontrati.

Poichè B invia dati ad A solo se deve inviare qualcosa o se deve dare riscontro, nel caso il buffer venisse svuotato ma B non dovesse inviare niente, allora A non saprebbe mai che il buffer si è svuotato. Per risolvere, A continua ad inviare segmenti con un byte di dati quando la finestra di ricezione di B è zero, in questo modo, quando saranno ricevuti dal buffer, B invierà un riscontro con la nuova  $RcvWindow$ .

### 2.5.5 Gestione della connessione TCP

Ora vedremo come una connessione TCP viene instaurata e chiusa.

Supponiamo che un host A (client) voglia instaurare una connessione con un host B (server). Il processo del client informa il TCP che vuole stabilire una connessione con il server. Il TCP client procede allora a stabilirla nel seguente modo:

- **Passo 1:**  $TCP_{client}$  invia uno speciale segmento a  $TCP_{server}$ . Questo segmento non contiene dati dello strato di applicazione ma un bit del campo flag nell'intestazione, il cosiddetto SYN bit, è posto a 1. Inoltre il client sceglie un numero di sequenza iniziale ( $client\_isn$ ) e inserisce questo numero nel campo numero di sequenza del segmento iniziale SYN.
- **Passo 2:** Quando il datagram IP contenente il segmento SYN del  $TCP_{client}$  arriva al server dell'host, il server estrae il segmento TCP dal datagram, determina il buffer TCP e le variabili alla connessione, e invia al client TCP un segmento che autorizza la connessione. Anche questo segmento non contiene dati dello strato di applicazione ma contiene tre informazioni:
  - SYN posto a 1
  - il campo di riscontro del segmento TCP è posto a  $client\_isn + 1$
  - il server sceglie il proprio numero iniziale di sequenza ( $server\_isn$ ) e colloca questo valore nel campo del numero di sequenza nell'intestazione TCP

A volte ci si riferisce al segmento che autorizza la connessione come a un segmento **SYNACK**.

- **Passo 3:** Dopo la ricezione del segmento SYNACK, anche il client destina buffer e variabili alla connessione. L'host del client invia allora al server un ulteriore segmento. Quest'ultimo segmento riscontra il segmento che autorizza la connessione inserendo nel campo di riscontro  $server\_isn + 1$ . Il bit SYN è posto a 0 perché la connessione è stabilita

Questa è la procedura di **handshake a tre vie**.

Ciascuno dei due processi che partecipano a una connessione TCP possono chiuderla. Quando viene chiusa, le risorse (buffer e variabili) negli host sono deallocate.

Il processo dell'applicazione impone un comando di chiusura, questo fa sì che il TCP del client invii un speciale segmento TCP al processo dell'altro host. Questo segmento speciale ha un bit nel campo flag dell'intestazione del segmento, il campo **FIN** posto a 1. Quando l'host riceve questo segmento, invia un ACK al mittente per poi a sua volta inviare un segmento di chiusura della connessione allo stesso modo. Il primo host a questo punto invia un ACK in risposta e tutti e due deallocheranno le risorse.

Durante il periodo dell'esistenza della connessione TCP, il protocollo TCP funziona in ciascun host eseguendo transizioni tra vari **stati del TCP**.

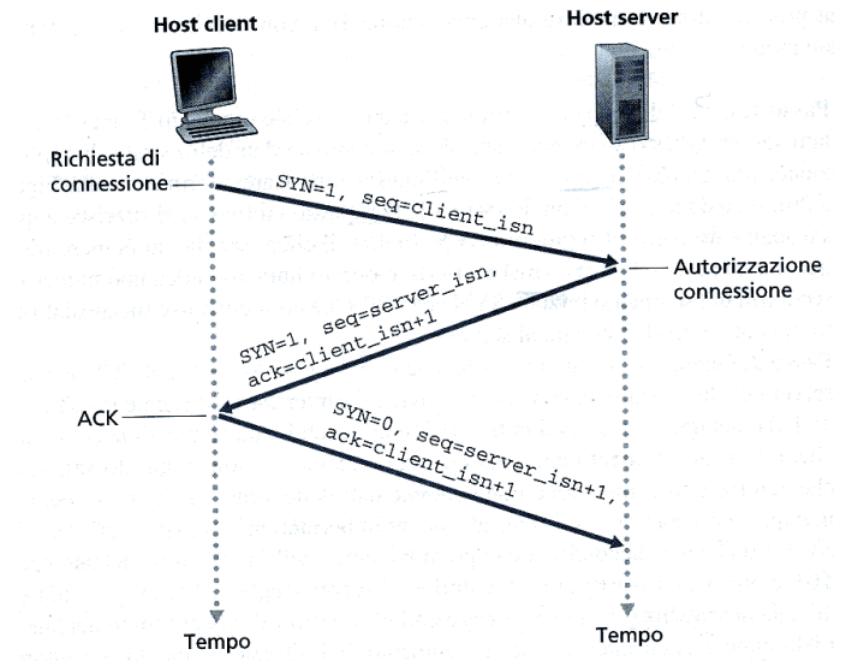


Figura 33: Handshake a tre vie

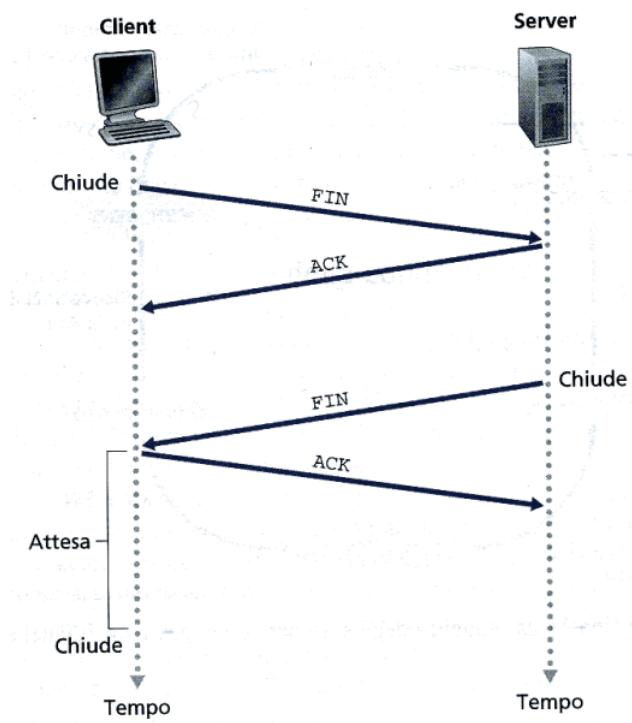


Figura 34: Chiusura di una connessione TCP

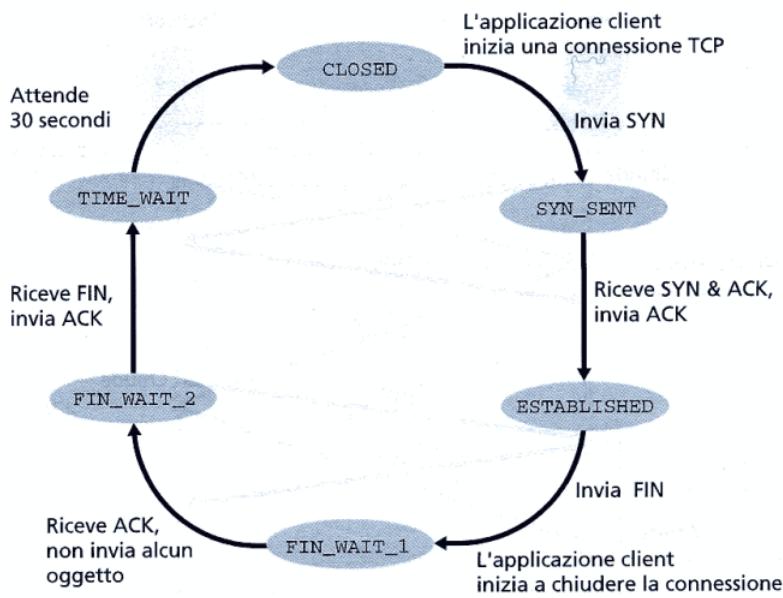


Figura 35: Tipica sequenza degli stati per i quali passa un TCP del client

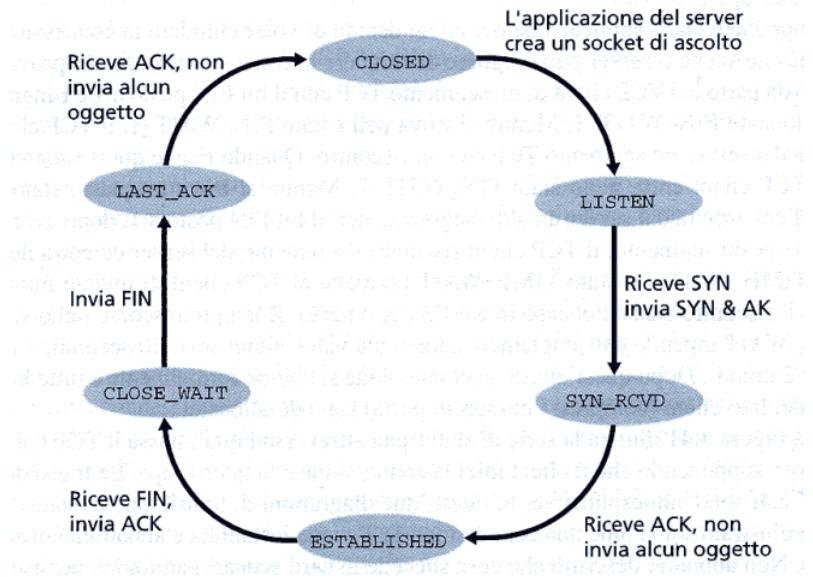


Figura 36: Tipica sequenza degli stati per i quali passa un TCP del server, supponendo che sia il client a interrompere la connessione

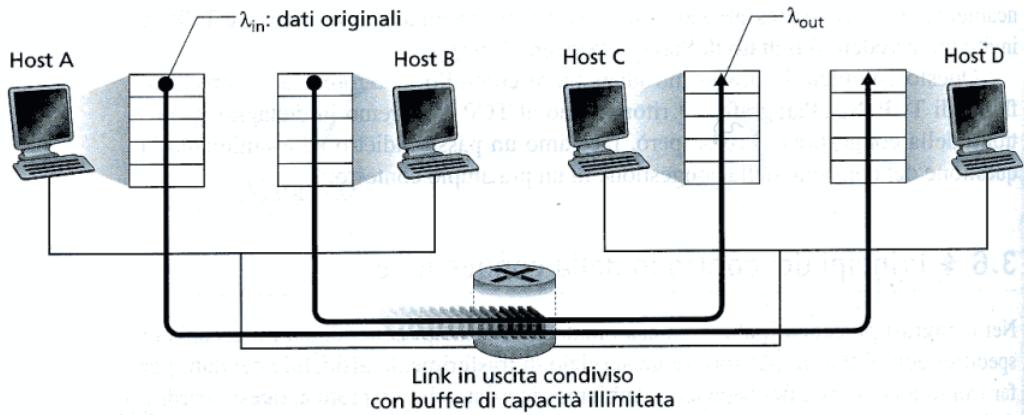


Figura 37: Primo scenario di congestione: due connessioni dividono un singolo hop con buffer infinito

## 2.6 Principi del controllo della congestione

Sappiamo che una delle cause principali della perdita di segmenti è la congestione della rete. La ritrasmissione dei segmenti cura l'effetto ma per correggere il problema alla radice servono sistemi che "strozzino" i sender nel caso di una congestione.

### 2.6.1 Le cause e i costi della congestione

Considereremo tre scenari di complessità crescente in cui si verifica la congestione. In tutti i casi valuteremo le cause della congestione e i suoi costi.

**Scenario 1: due sender, un router con buffer infinito** Due host (A e B), ciuscino con una connessione che condivide un singolo hop (salto, router) fra sorgente e destinazione.

Assumiamo che A stia inviando dati a una velocità media di  $\lambda_{in}$  byte/s. Questi dati sono "originali", nel senso che ciascuna unità di dati è inviata nel socket una sola volta. I dati sono incapsulati e spediti. non viene eseguito alcun recupero degli errori (ad esempio ritrasmissione), controllo di flusso o controllo della congestione. Ignorando il carico addizionale dovuto all'aggiunt delle informazioni di intestazione, la velocità a cui l'host A offre il traffico al router in questo primo scenario è  $\lambda_{in}$  byte/s. L'host B opera in maniera simile, quindi supponiamo che stia inviando anch'esso dati alla velocità di  $\lambda_{in}$  byte/s.

I pacchetti degli host A e B passano attraverso un router e su un link di uscita condiviso di capacità R. Il router ha buffer che permettono di incamerare i pacchetti in arrivo quando la velocità di arrivo supera la capacità di uscita.

Il grafico riporta il **throughput per la connessione** (numero di byte al

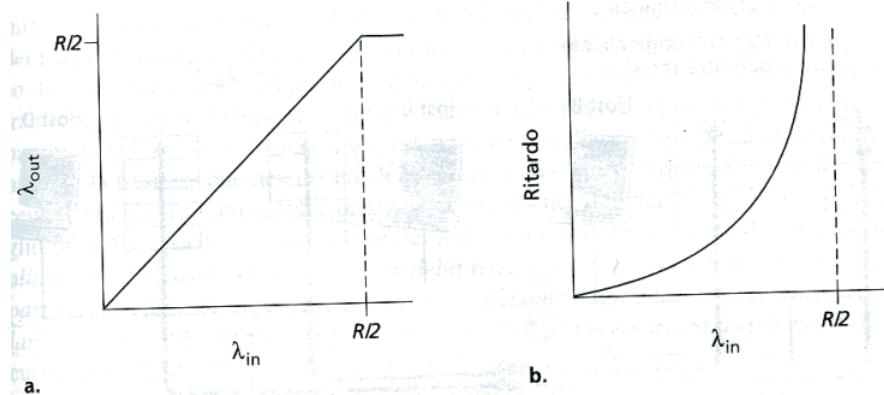


Figura 38: Primo scenario di congestione:throughput e ritardo in funzione della velocità di spedizione dell'host

secondo al receiver) in funzione della velocità di spedizione dei dati: per una velocità fra 0 e  $R/2$ , il throughput al receiver uguaglia la velocità di spedizione del sender.

**Attenzione:** il limite superiore  $R/2$  è una conseguenza della condivisione della capacità del link fra le due connessioni. Infatti, quando la velocità di spedizione supera  $R/2$ , il throughput è solo  $R/2$ .

Quando la velocità di spedizione supera  $R/2$ , il numero medio di pacchetti in coda nel router diventa illimitato, e il ritardo emdio tra sorgente e destinazione diventa infinito (assumendo che la connessione vada avanti all'infinito). Quindi, operare vicino a  $R$  come velocità è certo ottimo per il throughput ma pessimo per il ritardo.

Abbiamo trovato il costo della congestione di rete in questo scenario: ci si devono aspettare grandi ritardi di coda quando la velocità di arrivo dei pacchetti è prossima alla capacità del link.

**Scenario 2: due sender, un router con buffer finito** Assumiamo ora che il buffer del router sia finito. Una conseguenza di ciò è che i pacchetti in eccesso saranno scartati quando raggiungono un buffer pieno. Consideriamo anche che ciascuna connessione sia affidabile, ovvero che se un pacchetto viene perso dal router, sarà ritrasmesso dal mittente. Proprio per il fatto che i pacchetti possono essere rispediti dobbiamo stare attenti al termine "velocità di spedizione":

- lo strato dell'applicazione invia dati a  $\lambda_{in}$  byte/s
- lo strato di trasporto invia dati (originali e ritrasmessi) a  $\lambda'_{in}$  byte/s.  
Ci si riferisce a questa velocità come **carico offerto alla rete**

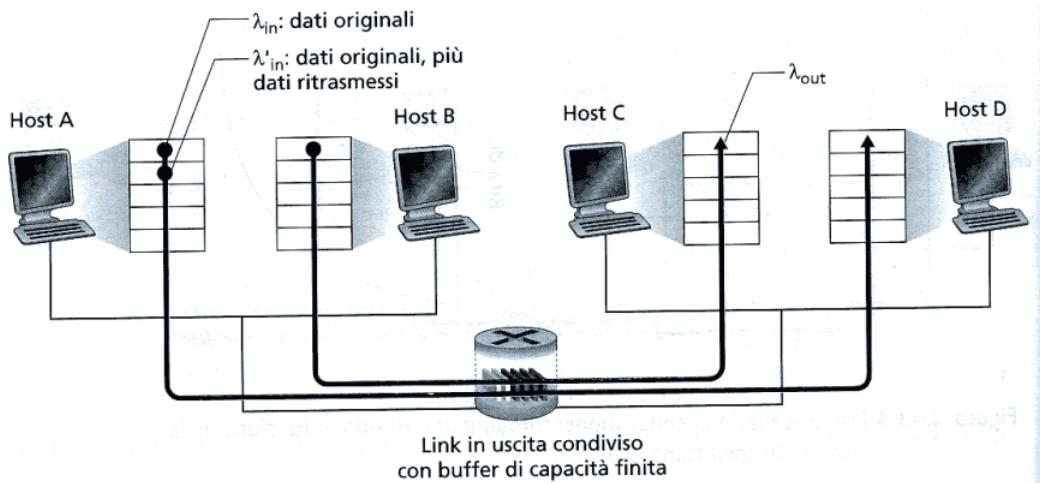


Figura 39: Secondo scenario: due host (con ritrasmissioni) e un router con buffer finito

Ora, se il mittente potesse sapere se il buffer è pieno o no allora non ci sarebbero ritrasmissioni, portando a  $\lambda'_{in} = \lambda_{in}$ . Questo caso è illustrato dalla retta superiore del grafico a sinistra.

Ora invece consideriamo il caso nel quale il sender rispedisca un pacchetto solo quando è sicuro che sia andato perso. In questo caso, le prestazioni potrebbero assomigliare a quelle del grafico a destra.

**Scenario 2: due sender, un router con buffer finito** Per capire, consideriamo il caso in cui il carico offerto  $\lambda'_{in}$  sia uguale a  $0,5R$ . In accordo con la figura a destra, a questo valore di carico offerto la velocità dei dati spediti spediti all'applicazione del receiver è  $R/3$ . Allora, delle  $0,5R$  unità di dati trasmessi,  $0,333R$  byte/s (in media) sono dati originali e  $0,166R$  byte/s (in media) sono dati ritrasmessi.

Vediamo qui un altro costo della congestione della rete: il sender deve eseguire ritrasmissioni per compensare i pacchetti scartati (dropped) a causa del sovraccarico del buffer.

Infine, consideriamo il caso nel quale il timer del sender scada prematuramente e il sender ritrasmetta il pacchetto che in realtà è stato ritardato dalla coda ma non è stato perso. In questo caso il lavoro fatto dal router nell'inoltrare la trasmissione della copia del pacchetto originale è sprecato. Il router, invece, avrebbe meglio usato la capacità di trasmissione del link per inviare un pacchetto diverso.

Ecco un altro costo della congestione di rete: le ritrasmissioni non necessarie a fronte di grandi ritardi portano il router a usare la larghezza di banda del suo link per inoltrare copie non necessarie.

La curva inferiore del grafico a sinistra mostra il throughput in funzione del

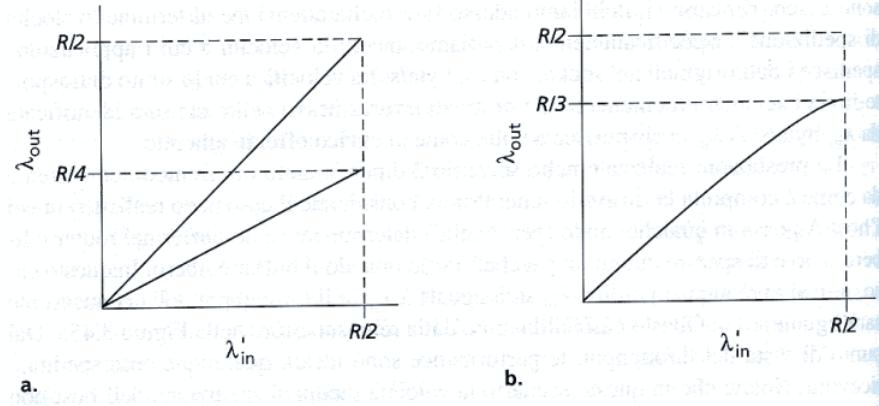


Figura 40: Secondo scenario: prestazioni

carico offerto quando si assume che ciascun pacchetto debba essere inoltrato (in media) due volte dal router, riducendo il valore asintotico a  $R/4$ .

**Scenario 3: quattro sender, router con buffer finito e percorsi a salti multipli** Quattro host trasmettono pacchetti, ciascuno su percorsi di due salti sovrapposti. Assumiamo ancora che ciascun host usi un meccanismo di timeout/ritrasmissione per implementare un servizio di trasferimento affidabile dei dati, che tutti gli host abbiano lo stesso valore di  $\lambda_{in}$  e che tutti i link dei router abbiano capacità  $R$  byte/s.

Consideriamo la connessione A-C, passando attraverso i router R1 e R2. La connessione A-C condivide R1 con D-B e R2 con B-D. Per valori estremamente piccoli di  $\lambda_{in}$ , il sovraccarico del buffer è raro e il throughput è circa uguale al carico offerto. Per valori leggermente superiori di  $\lambda_{in}$ , il throughput corrispondente è anch'esso maggiore, e una maggior quantità di dati originali è trasmessa nella rete e raggiunge la destinazione, il sovraccarico è anche raro. Quindi, per piccoli valori di  $\lambda_{in}$ , un aumento di  $\lambda_{in}$  si traduce in un aumento di  $\lambda_{out}$ .

Consideriamo ora il caso in cui  $\lambda_{in}$  (e quindi  $\lambda'_{in}$ ) sia estremamente grande. Consideriamo il router R2. Il traffico A-C in arrivo al router R2 può avere una velocità di arrivo che è al massimo  $R$ , la capacità del link da R1 a R2, indipendentemente dal valore di  $\lambda_{in}$ . Se  $\lambda'_{in}$  è estremamente grande per tutte le connessioni, allora la velocità di arrivo del traffico di B-D a R2 può essere più grande di quella del traffico A-C. Poiché il traffico di A-C e quello di B-D devono competere al router R2 a causa del limitato spazio di buffer, la quantità del traffico di A-C che con successo attraversa R2 diminuisce sempre più quando il carico offerto da B-D continua ad aumentare. Al limite, quando il carico offerto si avvicina all'infinito, un buffer vuoto in R2 è immediatamente riempito da un pacchetto di B-D, e il throughput della connessione A-C in R2 va a zero.

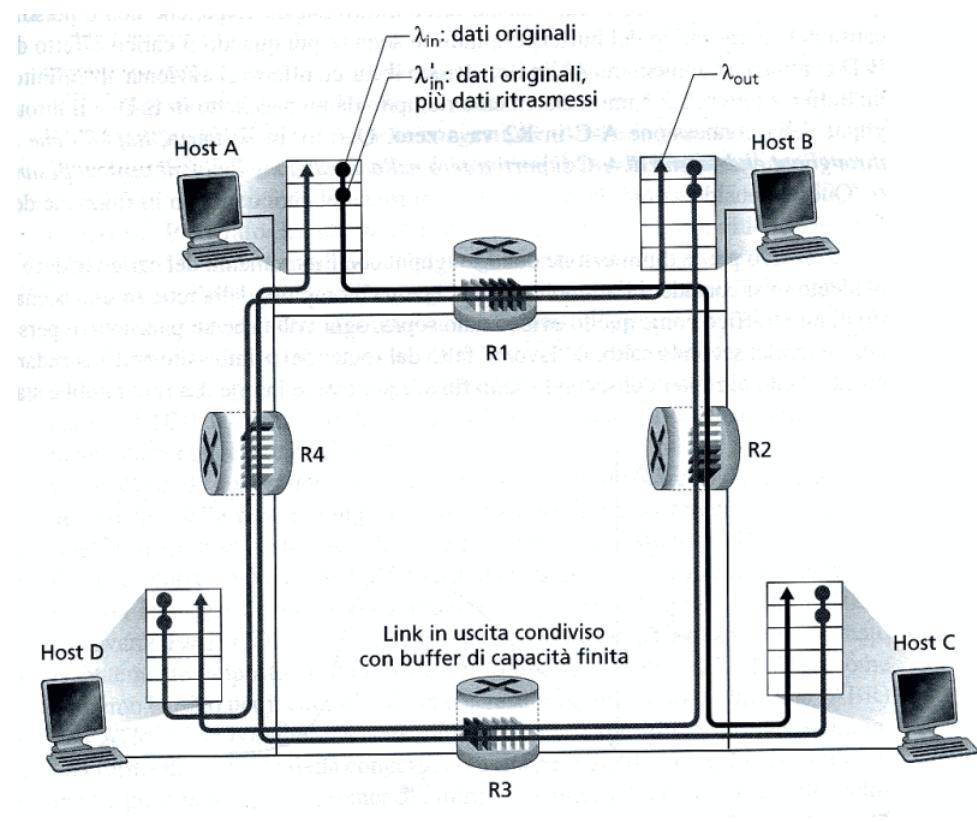


Figura 41: Quattro sender, router con buffer finito e percorsi a salti multipli

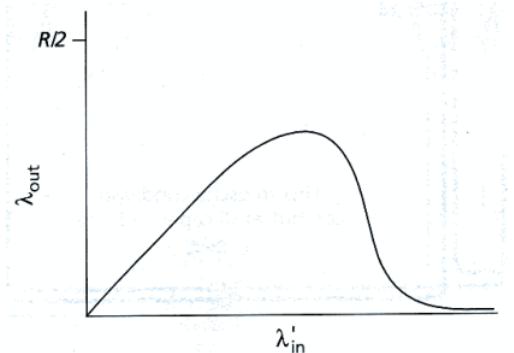


Figura 42: Terzo scenario: prestazioni con buffer finito e percorsi a salti multipli

Questo, in sostanza, implica che il throughput end-to-end di A-C si porti a zero nella condizione limite di traffico pesante.

Il motivo per la diminuzione del throughput con l'incremento del carico offerto è evidente se si considera l'ammontare del lavoro compiuto dalla rete. Se  $R_2$  si trova a scartare un pacchetto, allora il lavoro di  $R_1$  è stato sprecato. Qui possiamo vedere ancora un altro costo della perdita di un pacchetto dovuta alla congestione: quando un pacchetto è perso lungo un percorso, la capacità di trasmissione che è stata usata in ciascuno dei router a monte per inoltrare quel pacchetto al punto in cui si è perso è stata sprecata.

## 2.7 Controllo della congestione del TCP

Le due componenti più importanti di TCP sono:

- fornisce un servizio di trasporto affidabile
- ha un meccanismo di controllo della congestione

## 2.8 Controllo della congestione del TCP

Le due componenti più importanti di TCP sono:

- fornisce un servizio di trasporto affidabile
- ha un meccanismo di controllo della congestione

L'approccio seguito da TCP è di fare sì che ogni mittente limiti il ritmo a cui immette traffico nella sua connessione in funzione della congestione in rete percepita. Se un mittente TCP percepisce che c'è poca congestione nel percorso tra sé e la destinazione, allora il mittente TCP aumenta il suo ritmo di trasmissione; se il mittente percepisce che c'è congestione lungo il percorso, allora il mittente riduce il suo ritmo di invio. Ma questo approccio solleva tre problemi:

- in che modo il mittente TCP limita il ritmo a cui manda traffico nelle sue connessioni?
- come un mittente TCP percepisce che c'è congestione nel percorso tra sé e la destinazione?
- quale algoritmo dovrebbe utilizzare il mittente per cambiare il suo ritmo di invio in funzione della congestione end-to-end percepita?

Esamineremo prima in che modo un mittente limita il ritmo al quale invia traffico nella sua connessione. Il meccanismo di controllo della congestione del TCP da entrambi i lati della connessione deve tener traccia di un'altra variabile: la **finestra di congestione** (*congestion window*). La finestra di congestione impone una limitazione addizionale alla quantità di traffico che un host può inviare in una connessione. Specificatamente, l'ammontare dei dati non riscontrati che un host può avere all'interno di una connessione TCP non deve superare il minimo tra CongWin e RcvWin, che è:

$$LastByteSent - LastByteAcked \leq \min(CongWin, RcvWindow)$$

Per porre l'attenzione sul controllo della congestione assumiamo che il buffer di ricezione del TCP sia abbastanza grande da poter ignorare il vincolo della finestra di ricezione. In questo caso, la quantità di dati non riscontrati che un host può avere all'interno di una connessione TCP è limitata unicamente attraverso *CongWin*.

Consideriamo una connessione per cui i perdite e i ritardi di trasmissione dei pacchetti siano trascurabili. Quindi, approssimativamente, all'inizio di ogni tempo di round-trip<sup>6</sup> (RTT), il limite sopra esposto permette al mittente di inviare *CongWin* byte di dati nella connessione, e alla fine del RTT il mittente riceve i riscontri per i dati.

Quindi il ritmo di invio del mittente è circa *CongWin/RTT* byte/s.

Definiamo un "*evento di perdita*" a un mittente TCP come il verificarsi o di un timeout o della ricezione di tre ACK duplicati dal ricevente<sup>7</sup>. Quando c'è troppa congestione vi è perdita di datagrammi. Il datagram perso, a sua volta, dà luogo a un evento di perdita al mittente, o a un timeout o la ricezione di tre ACK duplicati, che è considerato dal mittente come un'indicazione della congestione del percorso.

Ora siamo in grado di considerare l'algoritmo che un mittente TCP usa per regolare il suo ritmo di invio in funzione della congestione percepita, ovvero l'**algoritmo di controllo della congestione di TCP**. L'algoritmo ha tre componenti principali:

---

<sup>6</sup>Il Round Trip Time (acronimo RTT) nelle telecomunicazioni è il tempo che intercorre tra l'invio di un segnale più il tempo necessario per la ricezione della conferma di quel segnale.

<sup>7</sup>Che, come abbiamo detto precedentemente, porta alla ritrasmissione veloce dei pacchetti ancora senza ACK

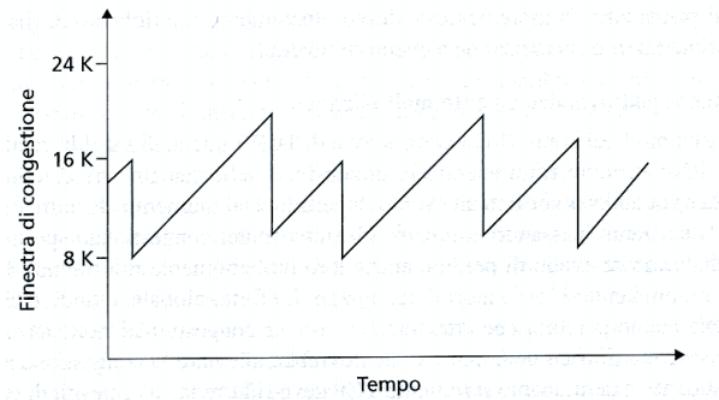


Figura 43: Algoritmo di controllo della congestione a incremento additivo-decremento moltiplicativo

- incremento adattivo, decremento moltiplicativo
- partenza lenta (*slow start*)
- reazione a eventi di timeout

**Incremento adattivo, decremento moltiplicativo** L'idea alla base del controllo di congestione di TCP è quella di far ridurre al mittente il suo ritmo di invio (diminuendo la dimensione della sua finestra di congestione, *CongWin*) quando si verifica un evento di perdita. Ma di quanto il mittente TCP deve ridurre la sua finestra di congestione quando si verifica un evento di perdita? Il TCP usa un approccio detto "**decremento moltiplicativo**", che dimezza il valore attuale di *CongWin* dopo un evento di perdita. Quindi, se il valore di *CongWin* è attualmente di 20 kbyte e si verifica una perdita, *CongWin* viene dimezzato a 10 kbyte. Il valore di *CongWin* può continuare a scendere, ma non può scendere sotto a un MSS. Questa è una spiegazione MOLTO semplificata come vedremo dopo.

Consideriamo ora come TCP debba aumentare il ritmo di invio se non percepisce congestione, ovvero quando non ci sono perdite. In questo caso TCP aumenta lentamente la sua finestra di congestione. Il mittente fa questo ogni volta che riceve un ACK, approssimativamente aumentandola di un MSS per ogni RTT fin quando non si verificano eventi di perdita.

Quindi TCP aumenta additivamente ( $CongWin = CongWin_{prec} + MSS$ ) e riduce moltiplicativamente ( $CongWin = CongWin_{prec}/2$ ). Per questo motivo, il controllo di congestione di TCP è spesso definito come un **algoritmo a incremento adattivo, decremento moltiplicativo (AIMD)**. La fase di incremento lineare del protocollo di controllo della congestione è nota come **prevenzione della congestione (congestion avoidance)**

**Partenza lenta (slow start)** Quando si inizia una connessione TCP il valore di CongWin è inizializzato a un MSS, dando luogo a un ritmo iniziale di invio pari approssimativamente a  $MSS/RTT$ . Per esempio, se  $MSS = 500$  byte e  $RTT = 200$  ms, allora il ritmo iniziale è solo circa 20 kbit/s. Dato che la banda potrebbe essere molto maggiore di  $MSS/RTT$ , sarebbe uno spreco aumentare il ritmo linearmente. Quindi, invece di incrementare il suo ritmo linearmente durante questa fase iniziale, un mittente TCP **aumenta il suo ritmo a velocità esponenziale, raddoppiando il valore di CongWin ogni RTT**. Al primo segnale di congestione si entra nel regime normale AIMD. Quindi, durante questa fase iniziale, chiamata **partenza lenta**, il mittente TCP inizia trasmettendo a un ritmo lento ma aumentando a velocità esponenziale.

**Reazioni a eventi di timeout** Il quadro presentato fino ad ora è incompleto, poiché TCP si comporta in maniera differente se la congestione è rilevata attraverso un evento di timeout e non attraverso tre ACK consecutivi. **Dopo un evento di timeout, il mittente TCP entra in una fase di partenza lenta.**

Il TCP gestisce queste dinamiche più complesse mettendo una variabile chiamata **Threshold (soglia)**, che determina la dimensione della finestra alla quale deve terminare la partenza lenta, e deve cominciare la prevenzione della congestione. La variabile *Threshold* è inizialmente posta a un valore grande (65 kbyte) in modo che non abbia alcun effetto iniziale. Quando si verifica un evento di perdita, *Threshold* è posto pari alla metà del valore attuale di *CongWin*. Per esempio, se *CongWin* è 20 kbyte, allora *Threshold* viene posto a 10 kbyte e conserverà questo valore fino al successivo evento di perdita.

Ora descriviamo come si comporta *CongWin* dopo un evento di timeout:

- **Fase iniziale:** partenza lenta,  $CongWin = CongWin_{prec} * 2$
- **Raggiungimento  $CongWin = Threshold$ :** AIMD,  $CongWin = CongWin_{prec} + MSS$
- **Evento di perdita, tipo ACK duplicato tre volte:**  $CongWin = CongWin_{prec}/2$ ,  $Threshold = CongWin_{prec}/2$
- **Evento di timeout:**  $CongWin = MSS$ ,  $Threshold = CongWin_{prec}/2$ , ricomincia la partenza lenta

Questo meccanismo è proprio della nuova versione di TCP, **TCP Reno**. Nella versione precedente, **TCP Tahoe**, per ogni evento di perdita si impostava  $CongWin = MSS$ .

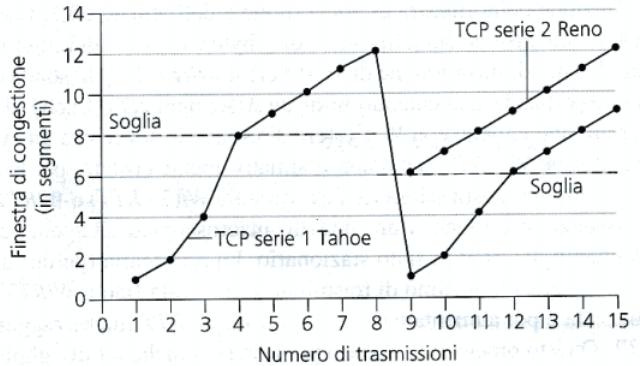


Figura 44: Evoluzione della finestra di congestione del TCP

**Descrizione macroscopica del throughput di TCP** Dato l’andamento di TCP, è naturale considerare quale possa essere il throughput medio di una connessione di lunga durata. Per questa analisi ignoriamo le fasi di partenza lenta e le perdite.

Quando la dimensione della finestra è di  $w$  byte e il tempo di round-trip è uguale a RTT secondi, il ritmo di trasmissione di TCP è circa  $w/RTT$ . Fino ad un evento di perdita,  $w$  viene aumentato di un MSS per ogni RTT. Indichiamo con  $W$  il valore di  $w$  quando si verifica una perdita. Assumendo che  $W$  e RTT siano approssimativamente costanti, il ritmo di trasmissione di TCP varia tra  $W/(2 * RTT)$  e  $W/RTT$ . Poichè il throughput aumenta linearmente fra due valori estremi, abbiamo:

$$\text{Throughput medio di una connessione} = \frac{0,75 * W}{RTT}$$

### 2.8.1 Fairness

Consideriamo  $K$  connessioni TCP, ciascuna con un diverso percorso da estremo a estremo, ma tutte passanti attraverso un link collo di bottiglia<sup>8</sup> (*bottleneck*) con ritmo di trasmissione pari a  $R$  bit/s. Supponiamo che ogni connessione stia trasferendo un grande file e non ci sia traffico UDP che passa attraverso il link collo di bottiglia. Un meccanismo di controllo della congestione è detto **fairness**, ovvero essere fair (equo) se il ritmo di trasmissione medio di ogni connessione è approssimativamente pari a  $R/K$ , cioè, ogni connessione ottiene un’uguale porzione della banda del link.

**AIMD è fair?** Sì. Consideriamo due connessioni TCP che condividono un singolo link con velocità di trasmissione  $R$ . Supponiamo che le due connessioni abbiano gli stessi MSS e RTT, che entrambe abbiano una grande quantità di dati da spedire e che nessun’altra connessione TCP o datagram

---

<sup>8</sup>Ovvero è l’unico congestionato e gli altri hanno capacità trasmissiva in abbondanza rispetto a questo

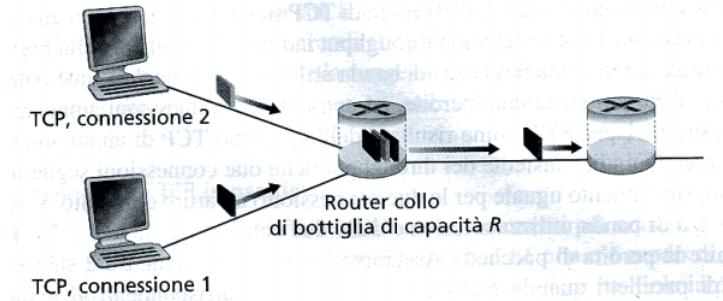


Figura 45: Due connessioni TCP condividono la banda di un singolo link collo di bottiglia

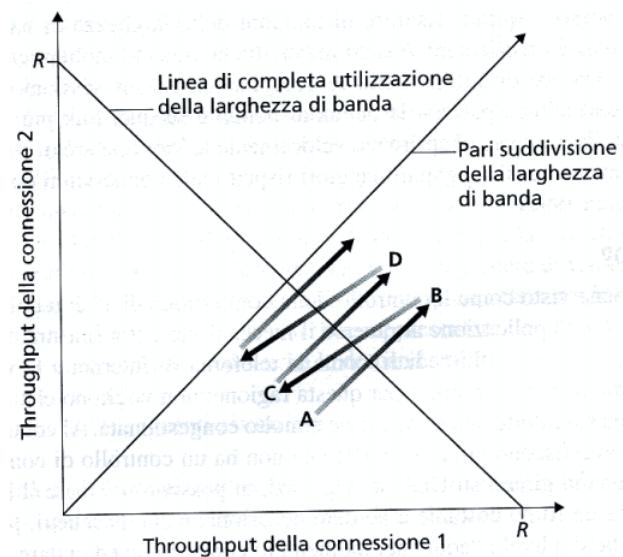


Figura 46: Throughput realizzati dalle connessioni TCP 1 e 2

UDP attraversino il link condiviso. Ignoriamo anche la partenza lenta e assumiamo che le connessioni TCP operino in modalità AIMD tutto il tempo. Se le due connessioni TCP condividono equamente la larghezza di banda del link, allora il throughput ottenuto cadrà sulla freccia a 45 gradi (pari suddivisione della larghezza di banda).

Supponiamo che le dimensioni della finestra di TCP siano quelle per cui a un certo tempo, le connessioni 1 e 2 realizzino i throughput indicati dal punto A. Poiché l'ammontare della larghezza di banda utilizzata insieme dalle due connessioni è inferiore a R, non ci saranno perdite, ed entrambe continueranno ad aumentare *CongWin* di un MSS per ogni RTT, procedendo su di una linea crescente a 45 gradi. Alla fine la somma dei due throughput sarà superiore a R come evidenziato nel punto B, allora entrambe le connessioni

ridurranno a metà la loro *CongWin*, portandosi a C e così via, mantenendo un equilibrio.

Questo è un caso molto irrealistico, infatti, ad esempio, la connessione con RTT inferiore aumenterà più velocemente, ottenendo la maggior parte della rete.

**Fairness e UDP** Molte applicazioni multimediali, come la telefonia su internet e le video conferenze, non girano su TCP proprio per questa ragione: non vogliono che il loro ritmo di trasmissione sia ridotto, anche se la rete è molto congestionata. Per questo girano su UDP, in questo modo possono aumentare costantemente la loro velocità senza problemi, sopportando alcune perdite di pacchetti. Dal punto di vista di TCP, le connessioni UDP non sono eque.

**Fairness e connessioni TCP in parallelo** TCP può non essere fair se l'applicazione sfrutta più connessioni TCP in parallelo. Per esempio i browser usano più connessioni per caricare le pagine. Per fare un esempio, immaginiamo che ci siano 9 connessioni TCP singole, tutte avranno  $R/9$  di velocità di trasmissione. Se si aggiungesse una nuova applicazione diventerebbero 10, portando la velocità a  $R/10$  per tutti. Se questa nuova applicazione, però, usasse 11 connessioni parallele, riuscirebbe a prendersi  $R/2$  della banda possibile.

## 3 Strato di rete e instradamento

### 3.1 Introduzione

Nell'immagine abbiamo due host, H1 e H2, e svariati router sul loro percorso. Supponiamo che H1 sia il mittente e H2 il destinatario e consideriamo il ruolo dello strato di rete.

Il strato di rete di H1 prende il segmento dallo strato di trasporto, incapsula ogni segmento in un datagram (che è il tipo di pacchetto dello strato di rete, e lo invia al primo router, R1. H2, quando riceve il datagram, estraе il segmento e lo invia allo strato di trasporto. Il ruolo primario dei router e di spedire (forward) i datagrammi da link di input a quello di output.

#### 3.1.1 Forwarding e Routing

Il ruolo dello strato di rete è semplice: spostare i pacchetti tra i vari host. Per fare ciò si identificano due funzioni importanti:

- **Forwarding:** quando un pacchetto arriva al link di input, il router semplicemente sposta il link all'appropriato link di output

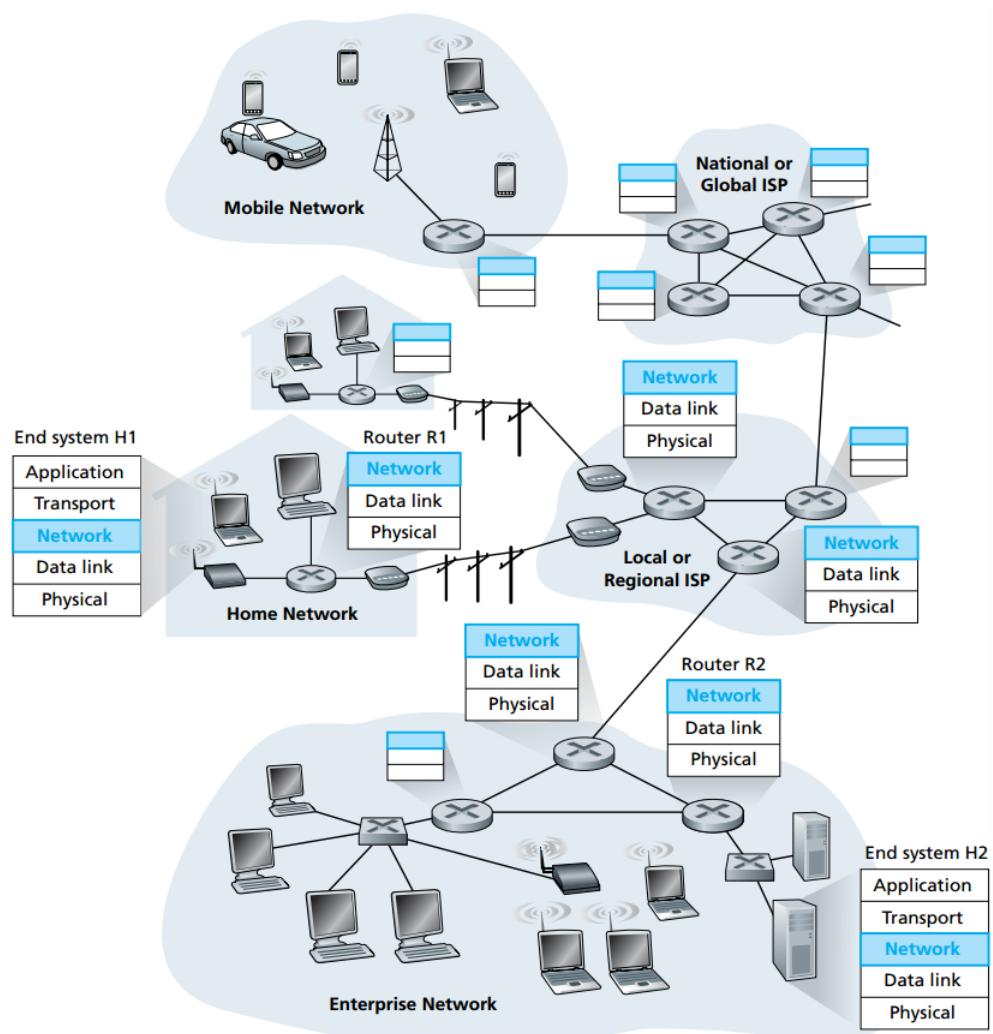


Figura 47: Lo strato di rete

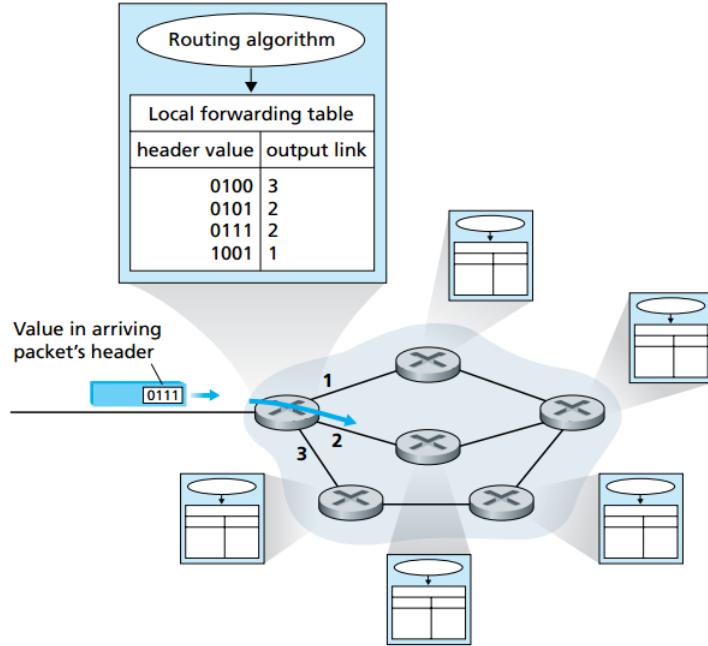


Figura 48: L'algoritmo di routing determina il valore nella tabella di forwarding

- **Routing:** Lo strato di network deve determinare il percorso che i pacchetti devono prendere affinché arrivino dal mittente al ricevente. L'algoritmo che calcola questi percorsi si chiama "**algoritmo di routing**".

Il termine **forwarding** si riferisce all'azione locale di trasferire i pacchetti mentre **routing** si riferisce al processo che coinvolge tutto il network.

Ogni router ha una **tabella di forwarding**. Un router inoltra il pacchetto esaminando il valore di un campo dell'header del pacchetto e poi usando questo valore per indicizzarlo nella tabella di forwarding. Il valore immagazzinato nella tabella di forwarding indica l'interfaccia del link di output sul quale deve essere instradato il pacchetto. Nella figura viene mostrato il funzionamento della tabella di forwarding: il pacchetto arriva con un'intestazione, questa viene cercata all'interno della tabella e determina il link da prendere. Come vengono determinati i valori della tabella di routing? L'algoritmo determina i valori che sono inseriti nella tabella. L'algoritmo può essere centralizzato (risiede su di un router che poi invia le informazioni ai router periferici) o decentralizzato (in ogni router).

Per impostare la terminologia, da qui useremo **packet switch** per indicare un dispositivo generico di packet-switching che trasferisce i pacchetti da un

link all’altro. Alcuni packet switch, chiamati **link-layer switches**, basano le loro decisioni di forwarding sul valore nel campo del frame dello strato di collegamento, altri packet-switch, i **router** basano la loro decisione sul valore del campo dello strato di rete.

**Impostazione della connessione** In alcuni network di computer lo strato di network ha una terza funzione: **l’impostazione della connessione** (*Connection setup*). La funzione è simile a quella dell’handshake, semplicemente viene eseguita tra tutti i router del percorso in modo che i pacchetti possano scorrere tra di essi.

### 3.1.2 Modelli di servizio di network

Quando il livello di trasporto trasmette un pacchetto al livello di rete, può fare affidamento sul livello di rete per inviare il pacchetto alla sua destinazione? Quando più pacchetti sono spediti, arriveranno tutti in ordine? L’intervallo di tempo tra l’invio sequenziale di due pacchetti sarà uguale all’intervallo di ricezione? La rete offrirà un feedback sulla congestione?

La risposta a queste domande e altre sono determinate dal modello di servizio offerto dal livello di rete. Il **modello di servizio di rete** definisce le caratteristiche del trasporto end-to-end.

Consideriamo ora alcuni servizi che il livello di rete può offrire. Il livello di trasporto del mittente, quando passa il segmento al livello di rete, specifica quali servizi includere:

- **Spedizione garantita**: il servizio garantisce che il pacchetto arriverà
- **Spedizione garantita con un ritardo massimo**: il servizio non solo garantisce la spedizione del pacchetto, ma consegnerà entro entro un specifico limite di ritardo host-to-host

Per di più, i seguenti servizi possono essere offerti ad un flusso dati:

- **Arrivo in ordine**: il servizio garantisce che i pacchetti arriveranno nell’ordine di invio
- **Banda minima garantita**: emula il comportamento del link di trasmissione, garantendo che sotto quella velocità di invio non ci sarà perdita di pacchetti e ogni pacchetto arriverà entro un certo ritardo (ad esempio, 40 millisecondi)
- **Massimo jitter garantito**: il servizio garantisce che l’ammontare di tempo tra due pacchetti successivi è uguale all’ammontare di tempo tra i loro arrivi a destinazione
- **Servizio di sicurezza**: usando una chiave segreta per la sessione che solo i due host conoscono, il livello di rete del mittente può criptare i pacchetti che poi saranno decriptati dal ricevente.

Network Architecture	Service Model	Bandwidth Guarantee	No-Loss Guarantee	Ordering	Timing	Congestion Indication
Internet	Best Effort	None	None	Any order possible	Not maintained	None
ATM	CBR	Guaranteed constant rate	Yes	In order	Maintained	Congestion will not occur
ATM	ABR	Guaranteed minimum	None	In order	Not maintained	Congestion indication provided

Figura 49: I modelli di servizio di internet, ATM CBR e ATM ABR. ATM = Asynchronous Transfer Mode, CBR = constant bit rate, ABR = Available bit rate

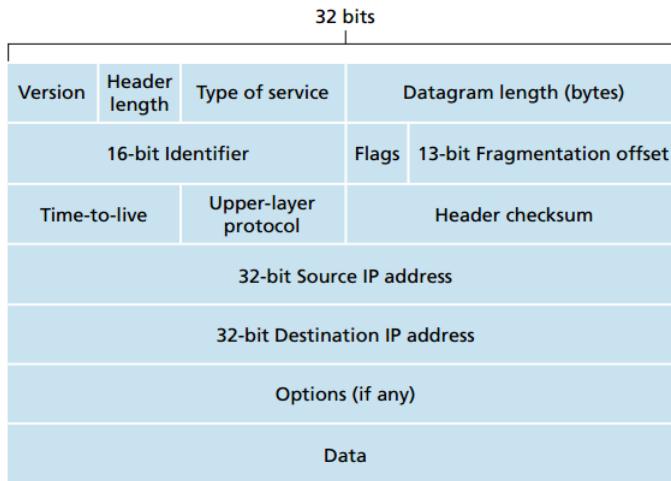


Figura 50: Datagram IPv4

Il livello di rete di internet offre un solo servizio, conosciuto come **best-effort** (al meglio delle possibilità). Un servizio Best-effort è uguale a dire che è un servizio senza garanzie, infatti anche se non arrivassero mai i pacchetti allora il servizio sarebbe comunque best-effort.

### 3.2 Il protocollo di internet (IP): forwarding e indirizzamento (addressing) in internet

Ci sono due versioni di IP oggi, IPv4 e IPv6. Noi studieremo solo IPv4.

#### 3.2.1 Formato dei datagrammi IPv4

Ricordiamo che un pacchetto del livello di rete è detto "datagram". I campi chiave di un datagram IPv4 sono:

- Numero di versione: questi 4 bit specificano la versione del protocollo IP per sapere come identificare i seguenti campi
- **lunghezza dell'header:** questi 4 bit determinano da che punto il datagram inizia veramente
- **Tipo di servizio (TOS):** servono a distinguere i vari tipi di datagram
- **Lunghezza del datagram:** mostra la lunghezza del datagram, dati e header compresi
- Identifieri, flag, offset di frammentazione
- **Time-to-live (TTL):** imposta un massimo di vita al datagram, in modo che non rimanga in rete per sempre. Questo valore viene decrementato di 1 ogni volta che il datagram è processato da un router. Se il TTL arriva a 0, il datagram viene scartato
- **Protocollo:** questo campo viene usato solo quando il datagram arriva a destinazione. il valore di questo campo indica il protocollo specifico del livello di trasporto da usare (TCP o UDP)
- **Checksum:** come per i segmenti di UDP e TCP
- **Indirizzi IP di arrivo e destinazione**
- **Opzionali:** permettono l'estensione dell'header
- **Dati** (payload)

**Frammentazione dei datagrammi IPv4** Il massimo ammontare di dati che un frame del livello di collegamento può trasportare è chiamato "**unità massima di trasmissione**" (*MTU*). Poichè ogni datagram IP è incapsulato in un frame del livello di collegamento per il trasporto da un router all'altro, allora l'MTU del protocollo del livello di collegamento pone un limite alla lunghezza del datagram IP. Il vero problema, però, non è questo limite ma il fatto che i vari router sul percorso possano utilizzare differenti protocolli con differenti MTU. Supponiamo che ci arrivi un pacchetto di data dimensione e che questo debba essere indirizzato verso un altro link ma, problema, questo ha un MTU inferiore rispetto a quello in entrata. Che fare? La soluzione è la frammentazione dei dati nel datagram IP in due più piccoli datagram IP, ognuno incapsulato in un datagram più piccolo in un frame separato. Ognuno di questi datagram è chiamato **frammento**.

Per non aggiungere lavoro ai router, IPv4 non fa riassemblare i frame a questi ma è compito dell'host destinatario. Per riassemlarli, però, il sistema necessita di sapere se:

- i frame fanno parte di un frame più grande

Fragment	Bytes	ID	Offset	Flag
1st fragment	1,480 bytes in the data field of the IP datagram	identification = 777	offset = 0 (meaning the data should be inserted beginning at byte 0)	flag = 1 (meaning there is more)
2nd fragment	1,480 bytes of data	identification = 777	offset = 185 (meaning the data should be inserted beginning at byte 1,480. Note that $185 \cdot 8 = 1,480$ )	flag = 1 (meaning there is more)
3rd fragment	1,020 bytes (= 3,980–1,480–1,480) of data	identification = 777	offset = 370 (meaning the data should be inserted beginning at byte 2,960. Note that $370 \cdot 8 = 2,960$ )	flag = 0 (meaning this is the last fragment)

Figura 51: Frammenti IP

- se ha ricevuto tutti i frame
- come questi debbano essere riassemblati

Per risolvere questo problema, IPv4 setta i campi di identificazione, flag e offset di frammentazione nell'intestazione. Quando un datagram viene creato, il mittente mette un numero di identificazione, l'IP di arrivo e di destinazione all'interno dell'intestazione. Tipicamente, ogni volta che viene creato un datagram il numero di identificazione viene incrementato di 1. Quando il datagram viene frammentato, vengono inseriti tutti questi campi e il numero di identificazione è uguale per tutti i frammenti, in questo modo il ricevente sa quando più frame fanno parte dello stesso frame originale.

Poichè IP non è affidabile, quindi affinchè il ricevente sappia di aver effettivamente ricevuto l'ultimo frammento, il flag di questo è impostato a 0, mentre tutti gli altri hanno flag uguale a 1. In più l'offset permette di capire se il frammento è all'interno del datagram IP originale.

Per fare un esempio, immaginiamo che un datagram di 4000 byte (20 byte di intestazione più 3980 byte di dati) arrivi ad un router e debba essere inoltrato su di un link con MTU di 1500 byte. Dobbiamo quindi frammentare il datagram in 3 frammenti. Supponiamo che il numero di identificazione sia 777. Le caratteristiche dei tre frammenti sono descritte dall'immagine. Alla destinazione, i dati vengono passati al livello di trasporto