# Reti di calcolatori e internet

Appunti

Alessandro Ferro

# Introduzione

Questi appunti, presi dal libro "Reti di Calcolatori e Internet - un approccio Top Down" (Kurose, Ross, 7th edition) sono parziali in quanto una parte sono presi a penna. La numerazione dei capitoli, sezioni e paragrafi segue quella del libro.

# Indice

# Capitolo 1

# Reti di calcolatori e Internet

[Le sezioni 1.1 e 1.2 sono scritte a penna sul quaderno]

## 1.3 Il nucleo della rete

### 1.3.2 Commutazione di circuito

Per spostare i dati in una rete esistono due approcci: commutazione di pacchetto e commutazione di circuito.

Nelle reti a commutazione di circuito, le risorse necessarie sono allocate per l'intera durata della comunicazione tra i sistemi periferici. Nelle reti a commutazione di pacchetto, invece, tali risorse non sono riservate e sono allocate in funzione della necessità della rete.

Le reti telefoniche sono esempi di rete a commutazione di circuito in quanto il canale resta aperto anche in assenza di comunicazione.

Quando la rete stabilisce un circuito, viene riservata anche una certa larghezza di banda, pertanto, il mittente può trasferire i dai a una velocità costante garantita.

D'inverso, quando un host invia un pacchetto a un altro host su una rete a commutazione di pacchetto, esso viene immesso nella rete senza che vengano ad esso riservate risorse.

#### Multiplexing nelle reti a commutazione di circuito

In una rete a commutazione di circuito può essere implementato il multiplexing, ovvero l'utilizzo dello stesso mezzo trasmissivo da più utenti.

Il multiplexing viene implementato tramite:

- Suddivisione di frequenza: lo spettro di frequenza di un collegamento viene suddiviso tra le connessioni stabilite. Ciò, ad esempio, è il caso di walkie-talkie che comunicano su un circuito a una frequenza diversa da altri walkie-talkie che comunicano sullo stesso circuito.

- Suddivisione di tempo: quando la rete stabilisce una connessione a commutazione di circuito, assegna degli slot temporali per ogni sessione.

#### Confronto tra commutazione di pacchetto e commutazione di circuito

I denigratori della commutazione di pacchetto sostengono che quest'ultimo mal si adatti alle applicazioni in tempo reale a causa dei suoi ritardi variabili e non deterministici. I suoi sostenitori, invece, affermano che la commutazione di pacchetto non soltanto offra una migliore condivisione della larghezza di banda, ma è anche più semplice, efficiente ed economica.

La commutazione di pacchetto risulta essere più efficiente perché la commutazione di circuito pre-alloca l'uso del collegamento trasmissivo indipendentemente dalla richiesta necessaria, con collegamenti garantiti ma potenzialmente non necessari. D'altro canto la commutazione di pacchetto alloca le risorse su richiesta.

### 1.3.3 Una rete di reti

Gli ISP si distinguono per la copertura geografica: gli ISP di accesso, che forniscono Internet direttamente all'utente finale, sono connessi agli ISP regionali che a loro volta sono connessi agli ISP di primo livello (cosiddetti ISP globali). Tale configurazione permette a tutti i nodi della rete di essere in comunicazione.

Per ridurre i costi una coppia di ISP vicini e di pari livello gerarchico può fare uso di peering, cioè connettere direttamente le loro reti in modo tale che il traffico passi direttamente tra di loro anziché passare per un intermediario.

## 1.4 Ritardi, perdite e throughput nelle reti a commutazione di pacchetto

Idealmente, vorremmo che i servizi Internet siano in grado di spostare dati tra due sistemi periferici istantaneamente e senza nessuna perdita dati. Ciò, ovviamente, non è fattibile nella realtà: il throughput è necessariamente limitato, ci possono essere ritardi e si potrebbero perdere pacchetti.

### 1.4.1 Panoramica del ritardo nelle reti a commutazione di pacchetto

Un pacchetto parte da un host, passa una serie di router e conclude il viaggio in un altro host. In ciascun nodo lungo il percorso (inclusi sorgente e destinazione) il pacchetto subisce vari tipi di ritardo. I principali ritardi sono dovuti al ritardo di elaborazione, ritardo di accodamento, ritardo di trasmissione e ritardo di propagazione che complessivamente formano il ritardo totale di nodo.

La figura 1.16 mostra un ottimo schema che riassume i tipi di ritardo.

#### Ritardo di elaborazione

Il tempo richiesto per esaminare l'intestazione del pacchetto e per determinare dove dirigerlo, nonché il tempo richiesto per controllare eventuali errori al suo interno fa parte del ritardo di elaborazione.

#### Ritardo di accodamento

Una volta in coda, il pacchetto subisce un ritardo di accodamento mentre attende la trasmissione sul collegamento. Ne abbiamo già parlato in 1.3.1 "ritardi di accodamento e perdita di pacchetti"

#### Ritardo di trasmissione

Sia $L$ la lunghezza del pacchetto in bit e $R$ la velocità di trasmissione in bps dal router A al router B, il ritardo di trasmissione è $\frac{L}{R}$ secondi e risulta essere il tempo richiesto per trasmettere tutti i bit del pacchetto sul collegamento.

#### Ritardo di propagazione

Una volta immesso sul collegamento, il pacchetto deve propagarsi fino al successivo nodo della rete. Questo tempo si chiama ritardo di propagazione. Il ritardo di propagazione è dato da $d/v$ dove $d$ è la distanza tra i due router e $v$ è la velocità di propagazione nel collegamento (in $m/s$). Questo ritardo è puramente fisico: indica quanto tempo impiega un singolo bit a "viaggiare" dal mittente al destinatario una volta che è stato trasmesso. Questo ritardo solitamente è trascurabile.

#### Confronto tra ritardi di trasmissione e di propagazione

Il ritardo di trasmissione è la quantità di tempo impiegata dal router per trasmettere in uscita il pacchetto ed è funzione della lunghezza del pacchetto e della velocità di trasmissione sul collegamento, ma non ha niente a che fare con la distanza tra i due router. Come detto poc'anzi, è il tempo richiesto per trasmettere tutti i bit del pacchetto sul collegamento.

Il ritardo di propagazione, invece, è il tempo richiesto per la propagazione di un solo bit da un router a quello successivo ed è funzione della distanza tra i due router e della velocità di propagazione nel mezzo trasmissivo ma non ha niente a che fare con la dimensione del pacchetto o la velocità di trasmissione.

Nota come velocità di trasmissione $\neq$ velocità di propagazione.

| | Nome (simbolo) | Significato | Unità tipica |
|---|---|---|---|
| 1 | **Velocità di propagazione** ($v$) | Velocità con cui **un segnale elettrico/ottico** si propaga fisicamente lungo il mezzo (rame, fibra, aria, ecc.). | metri/secondo (m/s) |
| 2 | **Velocità di trasmissione** (o *data rate*) ($R$) | Velocità con cui **i bit vengono trasmessi** sul collegamento, cioè quanti bit al secondo vengono "iniettati" nel canale. | bit/secondo (bps) |

Table 1.1: Confronto tra velocità di propagazione e velocità di trasmissione.

**Formula**

Siano $d_{elab}$, $d_{acc}$, $d_{tra}$, $d_{prop}$, i ritardi, rispettivamente di elaborazione, accodamento, trasmissione e propagazione. Allora, il ritardo totale di un nodo è dato da:

$$d_{elab} + d_{acc} + d_{tra} + d_{prop}$$

### 1.4.2 Ritardo di accodamento e perdita di pacchetti

A differenza degli altri tre ritardi, quello di accodamento può variare da pacchetto a pacchetto. Ciò vuol dire che mentre elaborazione, trasmissione e propagazione sono costanti per ogni pacchetto che viaggia su quella specifica tratta, il ritardo di accodamento è variabile. Per tale motivo, nel caratterizzare il ritardo di accodamento si fa uso di misure statistiche quali il ritardo di accodamento medio e la sua varianza.

Denotiamo con $a$ la velocità di arrivo di pacchetti nella coda espressa in pacchetti al secondo, $R$ la velocità di trasmissione alla quale i bit vengono trasmessi in uscita alla coda espressa in bit al secondo e supponiamo che tutti i pacchetti abbiano dimensione $L$ bit. Pertanto arrivano al nodo $La$ bit/s.

Con $La/R$ si denota l'intensità di traffico e il risultato è un numero adimensionale (es. $\frac{500bps}{250bps} = 2$)

- **(1)** Se $La/R > 1$ vuol dire che stanno entrando più pacchetti di quanti il nodo può trasferirne in uscita e la coda cresce verso $\infty$ bit.

  Questo perché $\frac{La}{R} > 1$ implica che $La > R$

- **(2)** Se $La/R \leq 1$ succede quanto segue: Innanzitutto nota che se ogni pacchetto ha dimensione $L$ vuol dire che il nodo impiega $\frac{L}{R}$ secondi per smaltirlo. Ora ipotizziamo che $N$ pacchetti giungano simultaneamente ogni $N(\frac{L}{R})$ secondi. Se ogni pacchetto viene smaltito ogni $\frac{L}{R}$ secondi, in $N(\frac{L}{R})$ secondi ne vengono smaltiti esattamente $N$.

  In questo caso il primo pacchetto inizia a uscire dalla coda immediatamente, il secondo pacchetto deve aspettare $L/R$ secondi prima che il suo primo bit esca dalla coda, il terzo deve aspettare $2 * \frac{L}{R}$, il quarto deve aspettare $3 * \frac{L}{R}$ secondi e così via. Più in generale, il pacchetto N° deve aspettare $(N-1) * \frac{L}{R}$.

  Per rendere più chiaro il concetto facciamo un esempio. Si supponga che arrivino 10 pacchetti all'istante 0 ognuno grande 50bit con una velocità di trasmissione $R$ pari a 1000bit/s. Il primo pacchetto inizia a uscire dalla coda subito, ma il primo bit del secondo pacchetto deve aspettare $\frac{L}{R} = \frac{50}{1000} = 0.05s$ prima che l'intero pacchetto precedente sia completamente uscito. Il decimo pacchetto deve aspettare $(9) * 0.05 = 0.45s$ e, nel momento in cui il suo trasferimento sul collegamento sarà terminato, a tempo 0.5, ne subentrano altri 10, di cui il primo non deve aspettare nulla prima che inizia a essere trasferito.

  Ciò dimostra che la condizione $La/R \leq 1$, non garantisce assenza di attese.

**Perdita di pacchetti**

La quantità di pacchetti perduti aumenta in proporzione all'intensità di traffico in quanto la coda potrebbe non essere abbastanza capiente per soddisfare la richiesta. Pertanto, le prestazioni di un nodo sono misurate non solo in termini di ritardo ma anche dalla probabilità di perdita di pacchetti.

### 1.4.3 Ritardo end to end

Nella sottosezione 1.3.1 (paragrafo "trasmissione store e forward") abbiamo detto che, supponendo di avere N-1 nodi tra sorgente e destinazione (e quindi N collegamenti) il ritardo end-to-end è

$$d_{e2e} = N(d_{trasmissione}) = N(\frac{L}{R})$$

ma prendevamo in considerazione solo il ritardo di trasmissione. La formula generale è

$$d_{e2e} = N(d_{elab} + d_{accodamento} + d_{trasmissione} + d_{prop})$$

#### Traceroute

Traceroute è un programma eseguibile su qualsiasi host di Internet. Quando l'utente specifica il nome di un host di destinazione, il programma invia pacchetti speciali verso di essa, che, durante il loro percorso, passano attraverso una serie di router. Quando uno di questi router riceve questo pacchetto speciale, invia un messaggio che torna alla destinazione che contiene il nome e l'indirizzo del router.

Il funzionamento è il seguente: Supponiamo di avere N-1 router dalla sorgente alla destinazione. La sorgente invia N pacchetti ognuno dei quali contiene l'indirizzo della destinazione. Quando l' n-esimo router riceve il pacchetto marcato come n, anziché indirizzarlo verso la destinazione risponde al mittente come poc'anzi descritto. L'N-esimo pacchetto, che raggiunge la destinazione, invita anche quest'ultima a rispondere.

In questo modo l'origine può ricostruire il percorso intrapreso dai pacchetti ed è inoltre in grado di determinare i ritardi per ogni nodo.

#### Sistemi periferici, applicazioni e altri ritardi

Oltre ai ritardi descritti nelle sezioni precedenti, si potrebbero manifestare ulteriori ritardi nei sistemi periferici. A titolo esemplificativo, un sistema periferico può volontariamente ritardare la sua trasmissione in quanto condivide il mezzo trasmissivo con altri sistemi periferici.

Un altro esempio è nel VoIP: In VoIP il mittente deve prima di tutto riempire il pacchetto con conversazione digitalizzata prima di inviarlo su Internet. Questo tempo per riempire un pacchetto è detto ritardo di pacchettizzazione.

### 1.4.4 Throughput nelle reti di calcolatori

Un'altra misura critica delle prestazioni in una rete di calcolatori è il throughput end-to-end.

Consideriamo un trasferimento di file da A a B. Il throughput istantaneo in ogni istante di tempo è la velocità di bit/s alla quale B sta ricevendo il file. È misurato in B e non in A in quanto è l'unico modo per individuare la velocità effettiva di trasferimento dopo tutti i ritardi e colli di bottiglia che intercorrono nel percorso.

Se il file consiste di F bit e il trasferimento richiede T secondi affinché la destinazione riceva tutti i bit, allora il throughput medio è $F/T$ bit/s.

Supponiamo che A e B si stiano trasferendo un file e che tra di loro ci sia un router C. Supponiamo $R_1$ essere il throughput da A a C e $R_2$ il throughput da C a B. Se $R_1 \leq R_2$ allora la velocità di collegamento da A a B (throughput end to end) è $R_1$ ma se $R_1 > R_2$ la velocità di collegamento è $R_2$ in quanto il router fa da collo di bottiglia. Quindi il throughput end to end è $MIN(R_1, R_2)$ o più in generale $MIN(R_1, ..., R_{n+1})$ se ci sono $n$ router.

## 1.5 Livelli dei protocolli e loro modelli di servizio

### 1.5.1 Architettura a livelli

Un'architettura a livelli consente di discutere e analizzare una parte specifica e ben definita di un sistema complesso. Ciò permette di introdurre un ulteriore vantaggio: la modularità, che rende molto più facile cambiare l'implementazione di un servizio fornito da un determinato livello. Fino a quando il livello fornisce lo stesso servizio allo strato superiore e utilizza gli stessi servizi dello strato inferiore, la parte rimanente del sistema (ovvero gli altri livelli) rimane invariata al variare dell'implementazione del livello.

#### Stratificazione dei protocolli

I protocolli, nonché l'hardware e il software che li implementano, sono organizzati in livelli. Ciascun protocollo appartiene a uno dei livelli. Ogni livello fornisce il suo servizio (1) effettuando determinate azioni all'interno del livello stesso e (2) utilizzando i servizi del livello immediatamente inferiore (se c'è).

Un livello può essere implementato via hardware, software o con una combinazione di essi.

L'insieme dei protocolli di tutti i livelli viene chiamato pila di protocolli. Internet è formato da una pila a 5 livelli. Questi livelli sono: fisico, collegamento, rete, trasporto e applicazione. In ambito accademico si studia anche la pila a 7 livelli denominata ISO/OSI.

Un protocollo, per sua natura, serve per scambiare messaggi. Pertanto, un protocollo "vive" su più sistemi di rete.

Degli svantaggi dell'architettura a strati sono la possibilità che un livello duplichi le funzionalità di un livello inferiore e che un livello possa prelevare informazioni da altri livelli bypassandone i servizi esposti. Ciò viola lo scopo della separazione tra livelli.

#### Livello di applicazione

Il livello di applicazione è la sede dei protocolli usati dalle applicazioni di rete. Alcuni dei protocolli a questo livello sono HTTP, SMTP e FTP. I pacchetti di informazione del livello di rete vengono denominati messaggi.

#### Livello di trasporto

Il livello di trasporto trasferisce i messaggi del livello di applicazione. Vi sono due protocolli di trasporto: TCP e UDP.

- TCP: fornisce alle applicazioni un servizio orientato alla connessione (cioè l'utente deve stabilire una connessione, usarla e quindi rilasciarla), garantisce la consegna dei messaggi di applicazione e il loro ordine. Gestisce il controllo di flusso (ovvero la corrispondenza tra le velocità del mittente e destinatario) e ha inoltre un controllo di congestione della rete.
- UDP: questo protocollo è molto semplice. Non è orientato alla connessione e non garantisce affidabilità, controllo di flusso e controllo della congestione.

I pacchetti a livello di trasporto sono denominati segmenti.

#### Livello di rete

Il livello di rete si occupa di trasferire i pacchetti a livello di rete, detti datagrammi, da un host a un altro.

Il livello di trasporto passa al livello di rete il proprio segmento e un indirizzo di destinazione.

Il livello di rete comprende il protocollo IP.

#### Livello di collegamento

Per trasferire un pacchetto da un nodo a quello successivo sul percorso, il livello di rete si affida ai servizi del livello di collegamento. Esempi di protocolli a livello di collegamento includono Ethernet e WiFi.

Un datagramma potrebbe essere gestito da differenti protocolli a livello di collegamento lungo le diverse tratte che costituiscono il suo percorso.

Chiameremo frame i pacchetti a livello di collegamento.

#### Livello fisico

Mentre il compito del livello di collegamento è spostare frame tra nodi adiacenti, il ruolo del livello fisico è spostare i singoli bit. I protocolli di questo livello dipendono dall'effettivo mezzo trasmissivo.

### 1.5.2 Incapsulamento

Dalla sorgente, i dati scendono lungo la pila dei protocolli e vi risalgono nella destinazione. Nei commutatori, i dati scendono e salgono fino a un determinato livello: per i commutatori a livello di collegamento fino al livello 2, per i router fino al livello 3. Gli host implementano tutti e cinque i livelli.

In altre parole, presso un host mittente, un messaggio a livello di applicazione viene passato al livello di trasporto. Questo livello prende il messaggio e gli concatena informazioni aggiuntive (le informazioni di intestazioni) che verranno utilizzate dal protocollo di trasporto nell'host ricevente. Messaggio livello applicazione + intestazioni del protocollo di trasporto costituiscono il segmento.

Il protocollo di trasporto passa il segmento al livello di rete, il quale gli concatena le proprie intestazioni, formando il datagramma e così via scendendo nella pila.

Quindi a ciascun livello il pacchetto ha due tipi di campi: quello di intestazione e quello di payload (il carico utile). Il payload è tipicamente un pacchetto proveniente dal livello superiore.

# Capitolo 2

# Livello di applicazione

Le applicazioni sono la ragion d'essere delle reti di calcolatori. Senza di esse predisporre una rete sarebbe inutile.

## 2.1 Principi delle applicazioni di rete

I programmi che fanno uso dei protocolli a livello applicazione sono eseguiti sui sistemi periferici che comunicano tra loro via rete. Esempi di programmi sono i browser e i web server.

Un'applicazione di rete è un servizio che utilizza la rete per permettere la comunicazione o lo scambio di dati tra host diversi. È formata da uno o più programmi applicativi (es. browser, client e-mail) che comunicano tra loro tramite protocolli di livello applicazione (es. HTTP, SMTP, IMAP).

Esempi: Web, posta elettronica (e-mail), messaggistica, streaming.

La loro implementazione è detta programma. Un programma in esecuzione è un processo.

### 2.1.1 Architettura delle applicazioni di rete

L'architettura di rete fornisce alle applicazioni uno specifico insieme di servizi. Il compito dello sviluppatore è scegliere tra utilizzare l'architettura client-server o l'architettura Peer to Peer.

Nell'architettura client-server vi è un host sempre attivo, chiamato server, che risponde alle richieste di altri host, detti client i quali sono i primi a inizializzare la comunicazione. I client non si connettono direttamente fra loro.

In un'architettura P2P si sfrutta invece la comunicazione diretta fra pari (peer), che a differenza di un server centralizzato possono (e spesso lo sono) collegati in maniera intermittente. I peer non appartengono a un fornitore di servizi ma appartengono agli utenti. Uno dei punti di forza dell'architettura P2P è la sua intrinseca scalabilità e ed è anche economicamente conveniente perché non richiede server prestanti nè una banda elevata.

Alcune applicazioni presentano un'architettura ibrida, combinando sia elementi P2P che client-server.

### 2.1.2 Processi comunicanti

I processi su due sistemi terminali comunicano scambiandosi messaggi attraverso la rete. Il processo mittente crea e invia messaggi sulla rete e il processo destinatario li riceve e, quando previsto, invia messaggi di risposta.

#### Processi client e server

Per ciascuna coppia di processi comunicanti ne etichettiamo uno come client e l'altro come server. In alcune applicazioni un processo può essere sia client che server (come ad esempio in un programma P2P). Nonostante ciò possiamo comunque etichettarli basandoci sul contesto di una specifica sessione.

Un processo che richiede il servizio e instaura la connessione è indicato come client mentre quello che eroga il servizio o procura le informazioni è detto server.

#### L'interfaccia tra il processo e la rete

Ogni messaggio inviato da un processo a livello applicativo a un altro remoto deve passare attraverso la rete sottostante. Il processo presuppone l'esistenza di un'infrastruttura esterna che trasporterà il messaggio attraverso la rete.

Un processo invia messaggi nella rete e riceve messaggi dalla rete attraverso un'interfaccia software detta socket. Una socket è l'interfaccia tra il livello di applicazione e quello di trasporto all'interno di un host. Viene chiamata anche API tra l'applicazione e la rete.

#### Indirizzamento

I processi devono poter avere un indirizzo per ricevere i messaggi inviati da un processo in esecuzione su un altro host. Per identificare un processo ricevente è necessario specificare due informazioni: (1) l'indirizzo dell'host e (2) un identificatore del processo ricevente sull'host di destinazione. Il punto (2) serve perché su un host potrebbero esserci più processi in esecuzione simultaneamente.

In Internet gli host vengono identificati attraverso un indirizzo IP, composto da 32 bit che per il momento possiamo pensare che identifica univocamente un host. Il processo mittente deve anche identificare il processo destinatario, più specificatamente, la socket che deve ricevere il dato. Un numero di porta di destinazione assolve questo compito. Alle applicazioni più note sono stati assegnati numeri di porta specifici.

### 2.1.3 Servizi di trasporto disponibili per le applicazioni

Il protocollo a livello di trasporto ha la responsabilità di consegnare i messaggi alla socket del processo ricevente. I protocolli di trasporto li possiamo classificare in 4 dimensioni: trasferimento dati affidabile, throughput garantito, temporizzazione e sicurezza

#### Trasferimento dati affidabile

In alcune applicazioni la perdita di informazioni potrebbe causare gravi conseguenze. Dunque, per supportare tali applicazioni occorre garantire che i dati inviati siano consegnati corretti e completi. Se un protocollo fornisce ciò si dice che fornisce un trasferimento dati affidabile.

In questo caso il processo mittente può passare alla socket i messaggi e sapere con assoluta certezza che verranno recapitati al processo ricevente.

Invece, le applicazioni che tollerano le perdite sono le applicazioni audio/video.

#### Throughput garantito

Un protocollo a livello di trasporto potrebbe fornire un throughput garantito. Con tale servizio l'applicazione potrebbe chiedere che il throughput sia almeno di $r$ bps.

Le applicazioni che hanno requisiti di throughput si dicono applicazioni sensibili alla banda, che si oppongono alle cosiddette applicazioni elastiche che possono funzionare anche senza garanzia di throughput.

#### Temporizzazione

Un protocollo a livello di trasporto potrebbe anche fornire garanzie di temporizzazione, ovvero garantire che ogni bit che il mittente invia venga consegnato al destinatario in non più di $t$ secondi.

#### Sicurezza

Un protocollo a livello di trasporto può fornire a un'applicazione servizi di sicurezza, per esempio cifratura e decifratura dei dati così come integrità e autenticazione.

### 2.1.4 Servizi di trasporto offerti da internet

Nelle sezioni precedenti abbiamo dato una panoramica dei servizi che un protocollo a livello di trasporto potrebbe fornire in teoria, ma Internet non fornisce tutti e 4.

In Internet ci sono due protocolli di trasporto: TCP e UDP.

#### Servizi di TCP

TCP prevede un servizio orientato alla connessione e il trasporto affidabile dei dati.

- Servizio orientato alla connessione: TCP fa in modo che client e server si scambino informazioni di controllo prima che i dati veri e propri comincino a fluire. Questa procedura, detta handshaking, prepara gli host alla comunicazione. Dopo la fase di handshaking c'è la connessione tra le socket full-duplex, ovvero possono scambiarsi contemporaneamente segmenti. L'applicazione deve chiudere la connessione quando il trasferimento dei dati termina.
- Servizio di trasferimento affidabile: i processi comunicanti possono contare su TCP per trasportare dati senza errori e nel giusto ordine.

TCP prevede anche un meccanismo di controllo della congestione che beneficia l'intero Internet, non solo i processi comunicanti.

#### Servizi di UDP

UDP è un protocollo di trasporto leggero non orientato alla connessione e dunque non necessita di handshaking. Non offre alcun tipo di affidabilità: il protocollo non garantisce che i segmenti arrivino al destinatario, e se lo fanno, potrebbero non giungere in ordine. Non include neanche un meccanismo di controllo della congestione.

#### Servizi non forniti dai protocolli di trasporto di Internet

Garanzie di throughput e temporizzazione non sono forniti dagli odierni protocolli di trasporto di Internet.

### 2.1.5 Protocolli a livello di applicazione

Un protocollo a livello di applicazione definisce come i processi di un'applicazione in esecuzione su sistemi periferici diversi si scambiano messaggi.

Un protocollo a livello di applicazione definisce:

- Quali tipi di messaggio esistono (richieste, risposte, notifiche)
- La sintassi dei messaggi
- La semantica dei messaggi
- Quando e in che ordine i messaggi vengono scambiati

È importante distinguere tra applicazioni di rete e protocolli a livello di applicazioni di rete.

Un protocollo è solo una parte di un'applicazione.

Il Web è un'applicazione di rete che consiste in uno standard per i formati dei documenti, il browser, il web server e infine il protocollo a livello di applicaione: HTTP.

## 2.2 Web e HTTP

### 2.2.1 Panoramica di HTTP

HTTP è un protocollo a livello di applicazione del Web. Questo protocollo è implementato in due programmi: client e server.

Una pagina Web è composta da uno o più oggetti. Un oggetto è un singolo file accessibile tramite un URL, come ad esempio una pagina HTML, un'immagine, un foglio di stile CSS o uno script JavaScript.

Nella maggior parte delle pagine Web, c'è un file HTML principale che definisce la struttura della pagina e referenzia altri oggetti necessari per la visualizzazione completa della pagina (immagini, stili, script, ecc.). L'insieme di questi oggetti, HTML principale più oggetti referenziati, costituisce la pagina Web complessiva.

Ogni URL ha almeno due componenti: il nome dell'host del server che ospita l'oggetto e il percorso dell'oggetto.

$$\underbrace{www.school.edu}_{Nome\ dell'host}\ /\ \underbrace{chemistry/lesson1.jpg}_{percorso\ dell'oggetto}$$

Un browser implementa il lato client di HTTP e un web server implementa il lato server di HTTP che ospita oggetti web indirizzabili tramite URL.

HTTP definisce in che modo i client web richiedono le pagine ai web server e come quest'ultimi le trasferiscono ai client.

HTTP utilizza TCP come protocollo di trasporto. In questo modo HTTP non si deve preoccupare dei dati smarriti, di perdite e riordinamento dei pacchetti.

Il client invia richieste e riceve risposte HTTP tramite la propria interfaccia socket. Analogamente, il server riceve richieste e invia risposte tramite la propria.

Il server invia i file richiesti al client senza memorizzare alcune informazione di stato su di esso. Per questo motivo viene detto protocollo senza memoria di stato (stateless).

### 2.2.2 Connessioni persistenti e non persistenti

Ciascuna coppia richiesta/risposta deve essere inviata su una connessione TCP separata o devono essere inviate tutte sulla stessa connessione? Nel primo caso si dice che l'applicazione usa connessioni non persistenti, nel secondo caso usa connessioni persistenti.

#### HTTP con connessioni non persistenti

Ecco cosa avviene quando HTTP usa connessioni non persistenti, supponendo che il client chieda la seguente pagina Web: www.school.edu/chemistry/index.html

1. Il processo client HTTP inizializza una connessione TCP con il server sulla porta 80.

2. Il client HTTP invia al server un messaggio di richiesta che include il percorso /chemistry/index.html

3. Il processo server recupera l'oggetto /chemistry/index.html, lo incapsula in un messaggio HTTP e viene inviato al client attraverso la socket

4. Quando il server si è assicurato che il messaggio è arrivato alla destinazione, comunica a TCP di chiudere la connessione.

5. Il client riceve il messaggio di risposta e la connessione TCP termina

Come abbiamo visto, ciascuna connessione TCP trasporta solo un messaggio di richiesta e uno di risposta. Se la pagina index.html avesse referenziato 10 oggetti al suo interno, avrebbe generato 11 connessioni TCP, ognuna per ciasun oggetto.

Definiamo essere Round Trip Time (RTT) il tempo che intercorre tra l'invio di una richiesta e la ricezione (del primo bit) della risposta. Quando il client vuole ottenere una risorsa HTTP, apre una connessione TCP con il server il quale risponde con una conferma. Il tempo per fare ciò è almeno 1 RTT. Dopodiché il client manda un messaggio al server che contiene (1) a sua volta una conferma di avvenuta ricezione e (2) la richiesta HTTP della risorsa. A questo punto il server risponde con il file HTML. Il tempo per fare ciò è almeno un altro RTT.

#### HTTP con connesioni persistenti

Le connessioni non persistenti presentano alcuni limiti: il primo è che per ogni oggetto richiesto occorre stabilire e mantenere una nuova connessione. In secondo luogo ciascun oggetto subisce un ritardo di 2 RTT. Il primo per stabilire la connessione TCP il secondo per richiedere e ricevere un oggetto.

Con HTTP a connessione persistente il server lascia la connessione TCP aperta dopo l'invio di una risposta, per cui le richieste e le risposte successive useranno la stessa connessione. In questo modo il server può anche spedire più oggetti contemporaneamente. Le richieste possono infatti essere fatte senza aspettare prima il completamento delle richieste precedenti (pipelining). Il server HTTP chiude la connessione TCP quando essa rimane inattiva per un dato lasso di tempo. La modalità di default di HTTP impiega connessioni persistenti con pipelinig.

### 2.2.3 Formato dei messaggi HTTP

Le specifiche HTTP includono la definizione dei due formati HTTP: richiesta e risposta.

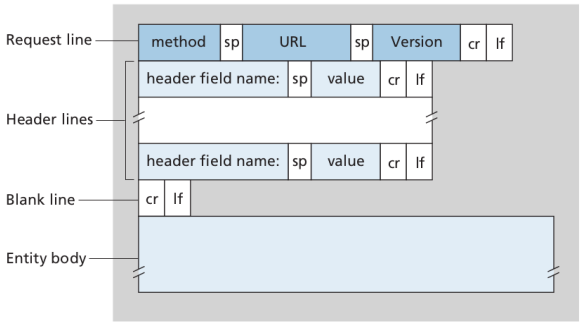
#### Messaggio di richiesta HTTP

Un messaggio di richiesta HTTP è

```
GET /somedir/index.html HTTP/1.1\r\n
Host:  www.school.edu\r\n
Connection:  close\r\n
User-agent:  Mozilla/5.0\r\n
Accept-language:  it\r\n
```

Notiamo fin da subito che il testo è scritto in codifica ASCII. Consiste di righe, ognuna delle quali con dei ritorni a capo. Un messaggio di richiesta può essere costituito da un numero indefinito di righe, anche una sola (la prima è obbligatoria). La prima riga è definita riga di richiesta mentre le altre sono righe di intestazione (header).

- La prima riga presenta tre campi: il campo metodo, il campo URL e il campo versione di HTTP. Il campo metodo è GET quando viene richiesto un oggetto, POST quando l'utente riempie un form (l'utente sta ancora richiedendo una pagina web al server, ma i contenuti specifici della pagina dipendono da ciò che l'utente ha inserito). Si può usare GET anche per riempire un form, includendo i dati nell'URL (query string). Poi vi è il metodo HEAD, e quando un server riceve un metodo di questo tipo risponde ma tralascia gli oggetti richiesti. PUT consente di inviare un oggetto a un percorso specifico e DELETE permette di cancellare un oggetto su un server. In realtà questi metodi (o verbi) HTTP sono solo convenzioni. Il protocollo HTTP non impone al server di comportarsi in un certo modo.
- Host specifica l'host su cui risiede l'oggetto. Si potrebbe pensare sia superflua dato che già in corso una connessione TCP con l'host ma quest'informazione serve per la cache dei proxy
- Con la terza linea il browser specifica che vuole che il server chiuda la connessione dopo aver inviato l'oggetto richiesto.
- User-agent specifica quale browser sta effettuando la richiesta
- Accept-language specifica la lingua richiesta.




Formato generale dei messaggi di richiesta HTTP

Il corpo è (solitamente, per convenzione) vuoto nelle richieste GET.

#### Messaggio di risposta HTTP

Un esempio di messaggio di risposta HTTP è

```
HTTP/1.1 200 OK
Connection:  close
```

```
Date:  18 Aug 2025
Server:  Apache
Last-Modified:  11 Jun 2025
Content-Length:  3145
Content-Type:  text/html
<html>...</head>
```

Qui per prevità abbiamo omesso i caratteri di carriage return e new line. Troviamo la riga di stato, un numero variabile di intestazioni e il corpo.

- La riga di stato presenta tre campi: la versione del protocollo, un codice di stato e un messaggio relativo al codice di stato
- Connection: close sta ad indicare che il server ha intenzione di chiudere la connessione TCP dopo l'invio del messaggio
- Date indica la data di invio del messaggio
- Server indica il tipo di server che sta fornendo il messaggio
- Last-Modified indica quando l'oggetto è stato modificato per l'ultima volta
- Content-Length indica quanti byte è il corpo
- Content-Type indica il tipo del corpo, in questo caso un file HTML.

### 2.2.4 Interazione utente-server: i cookie

Abbiamo detto che HTTP è stateless. Tuttavia, è spesso auspicabile che i web server possano tener traccia degli utenti per limitare l'accesso ad alcune pagine a taluni o per fornire contenuti in funzione della loro identità.

A questo scopo HTTP adotta i cookie, che consentono ai server di tener traccia degli utenti.

Per poter utilizzare i cookie sono necessari 4 componenti:

- Una riga di intestazione nel messaggio di risposta
- Una riga di intestazione nel messaggio di richiesta
- Un file mantenuto sul sistema dell'utente gestito dal browser
- Un database sul server

Il server crea un identificativo univoco e lo inserisce all'interno del proprio database. A questo punto il server risponde al browser inserendo nel messaggio HTTP un header chiamato Set-cookie che contiene il numero identificativo. Quando il browser riceve il messaggio, aggiunge una riga nel proprio file. Questa riga contiene il nome del server e il numero identificativo.

Ogni volta che l'utente richiede una pagina web, il suo browser consulta il suo file dei cookie, estrae il suo numero identificativo per il sito e lo pone nella richiesta HTTP. In tal modo il server può monitorare l'attività dell'utente nel sito.

### 2.2.5 Web caching

Una web cache (o cache proxy) è un'entità di rete che, dal punto di vista del client, si comporta come il server web rispondendo alle richieste HTTP al suo posto. Il proxy ha una propria memoria in cui conserva copie di oggetti recentemente richiesti.

Ecco cosa succede:

1. Il browser stabilisce una connessione TCP con il cache proxy server e gli invia una richiesta HTTP per l'oggetto specificato.
2. Il proxy controlla la presenza di una copia dell'oggetto memorizzato localmente. Se l'oggetto viene rilevato, il proxy lo inoltra al client

3. Se la cache non dispone dell'oggetto, apre una connessione con il server di origine, richiedendo l'oggetto in questione. Il server gli risponde

4. Il proxy salva l'oggetto nella propria cache e lo manda al client.

Il proxy è contemporaneamente sia client che server a seconda dei casi.

I proxy possono (1) ridurre i tempi di risposta ai client e (2) riducono il carico dei web server e dei loro collegamenti. Ciò implica che l'intero Internet ne trae beneficio.

### GET condizionale

L'oggetto ospitato nel web server potrebbe essere stato modificato rispetto alla copia presente nel cache proxy. HTTP presenta un meccanismo che permette alla cache di verificare se i suoi oggetti sono aggiornati. Questo meccanismo è chiamato GET condizionle.

Un messaggio di richiesta HTTP viene chiamato messaggio di GET condizionale se (1) usa il metodo GET e (2) include una riga di intestazione `If-Modified-Since`.

Vediamone il funzionamento:

1. Un proxy invia un messaggio di richiesta a un web server per conto di un browser richiedente, il quale gli risponde.

2. In cache memorizza sia l'oggetto che la data di ultima modifica presente nell'intestazione `Last-Modified`

3. Se dopo un certo periodo di tempo un client richiede la stessa risorsa, la cache non può rispondere direttamente con quella che ha in memoria perché potrebbe essere stata modificata sul server originale. Allora la cache effettua un controllo di aggiornamento inviando una GET condizionale al server, specificando nell'header `If-Modified-Since` la data che ha in memoria precedentemente prelevata dall'header `Last-Modified`. Questo comunica al server di inviare l'oggetto solo se è stato modificato rispetto alla data specificata

4. Se è stato modificato, il server risponde con un messaggio HTTP normale compreso di oggetto e la cache aggiorna il valore di ultima modifica. Altrimenti manda un messaggio HTTP con corpo vuoto (per non sovraccaricare inutilmente la banda) con messaggio di stato `304 Not Modified` che comunica al proxy che può procedere a inoltrare al client la copia dell'oggetto presente in cache

## 2.3 Posta elettronica in Internet

L'email rappresenta un mezzo di comunicazione asincrono.

L'email è composta da tre elementi principali: gli user agent (i programmi di posta), i server di posta e i protocolli quali SMTP, POP e IMAP.

Quando il mittente ha finito di comporre un messaggio, il suo user agent lo invia al suo server di posta. Se il destinatario ha un altro server di posta, il primo manda il messaggio al secondo. Infine, l'user agent del destinatario recupera il messaggio dal suo server. Ogni utente ha una casella di posta (uno spazio a lui dedicato) su un server di posta su cui vengono depositate le mail a lui indirizzate.

Se il server del mittente non può consegnare la posta a quello del destinatario, la trattiene in una coda dei messaggi e riprova l'invio diverse volte. Dopo averci provato più volte e ogni volta l'invio fallisce, il server del mittente comunica al mittente la mancata consegna.

SMTP presenta il principale protocollo a livello di applicazione per la posta elettronica. Fa uso del protocollo TCP sulla porta assegnata numero 25. Esso presenta un lato client e un lato server. Un server di posta può agire sia da client che da server SMTP: quando la invia agisce come client, quando la riceve agisce come server.

### 2.3.1 SMTP

SMTP trasferisce i messaggi dall'user agent del mittente al proprio server, nonché dal server del mitente a quello del destinatario.

Vediamo cosa accade nel dettaglio:

1. Il mittente invoca il proprio user agent per la posta elettronica, fornisce l'indirizzo di posta del destinatario, compone il messaggio e dà istruzione allo user agent di inviare il messaggio.

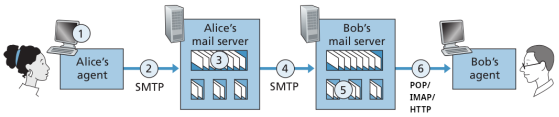
2. Lo user agent invia il messaggio al proprio server di posta tramite SMTP (vedremo più avanti che potrebbe usare anche HTTP)

3. Il lato client di SMTP, eseguito sul server mittente, apre una connessione TCP verso il server SMTP in esecuzione sul mail server del destinatario (se questi usa un server diverso da quello del mittente)

4. Il client SMTP (sul server mittente) invia il messaggio al server SMTP (server ricevente)

5. Il server SMTP del ricevente pone il messaggio nella casella del destinatario

6. Il destinatario, quando lo ritiene opportuno, invoca il proprio user agent per leggere il messaggio (usando POP, IMAP o HTTP che vedremo più avanti)




Esempio di funzionamento

Il client (o i client) riusano la stessa connessione TCP se hanno altri messaggi da inviare al server, pertanto, SMTP fa uso di connessioni persistenti.

### 2.3.2 Confronto con HTTP

HTTP trasferisce i file da un web server a un web client, mentre SMTP trasferisce messaggi di posta elettronica da un mail server a un altro.

Sia HTTP standard che SMTP fanno uso di connessioni persistenti.

Una differenza sostanziale è che HTTP è un protocollo di tipo pull, ovvero il client "tira" le informazioni del server a sè. In altre parole chi instaura la connessione è la macchina che vuole ricevere il file. Al contrario SMTP è un protocollo di tipo push, ovvero il client "spinge" le informazioni al server. In altre parole, la connessione TCP viene inizializzata dall'host che vuole spedire i file. Ciò non vìola la definizione di client/server in quanto il client (1) avvia la comunicazione e (2) richiede il servizio di invio mail al server.

Un'altra differenza è che SMTP obbliga l'intero messaggio a essere codificato in ASCII a 7 bit mentre HTTP non pone tale restrizione.

### 2.3.3 Formati dei messaggi di posta

Il corpo dei messaggi di posta è preceduto da degli header. Queste righe contengono testo leggibile, costituito da una parola chiave, seguita da due punti a loro volta seguiti da un valore (esempio: Subject: this is an email). Alcune di queste righe sono obbligatorie, come From e To. Attenzione a non confondersi tra i "From/To" dei messaggi SMTP e i "From/To" degli header di messaggio. I campi From/To del protocollo SMTP servono al trasporto dell'email fisicamente e non compaiono all'interno del contenuto leggibile della mail che invece fa visualizzare i campi From/To negli header. In alcuni casi questi possono differire.

### 2.3.4 Protocolli di accesso alla posta

Quando SMTP consegna il messaggio alla casella del destinatario, come fa quest'ultimo a recuperare il messaggio?

Verrebbe quasi naturale rispondere "posizioniamo un server SMTP presso il computer del destinatario", ma con questo approccio il computer del destinatario dovrebbe rimanere sempre acceso e connesso a Internet al fine di ricevere la posta. Ciò non è raccomandabile, pertanto si preferisce che l'utente acceda alla propria posta che risiede sul server quando e come egli vuole. Si noti che non possiamo usare SMTP qui in quanto è un protocollo push mentre in questo caso servirebbe un protocollo pull.

Ebbene, si usano dei protocolli per richiedere e trasferire la posta dal server al client. I principali sono POP3, IMAP e HTTP.

Ricapitolando, SMTP è usato per trasferire posta dallo user agent del mittente al proprio server di posta ed è usato per trasferire la posta dal server mittente al server destinatario, ma per trasferire la posta dal server destinatario al suo user agent è necessario un protocollo di accesso alla posta.

#### POP3

POP3 entra in azione quando lo user agent (il client), apre una connessione TCP verso il mail server sulla porta 110. POP3 procede in tre fasi: autorizzazione, transazione e aggiornamento.

Durante la prima fase lo user agent invia nome utente e password.

Durante la seconda fase lo user agent recupera i messaggi e può marcarne alcuni (anche quelli non scaricati) come da eliminare.

La fase di aggiornamento ha luogo quando il client conclude la sessione e in questo istante il server elimina i messaggi che sono stati marcati come da eliminare.

Uno user agent che usa POP3 può essere configurato dall'utente per "scaricare e cancellare" o per "scaricare e mantenere". Nella modalità scarica e cancella, come suggerisce il nome, il client dice al server di cancellare tutte le mail che ha scaricato. Se ciò avviene, il destinatario non può visualizzare i messaggi da un altro dispositivo.

#### IMAP

Con l'accesso tramite POP3, dopo aver scaricato i messaggi sulla macchina locale, l'utente può creare cartelle nelle quali inserire i messaggi. Questo paradigma pone però problemi per gli utenti mobili in quanto la struttura delle directory non si muove insieme a loro. Serve allora un modo per mantenere la gerarchia delle cartelle sul server alla quale si può accedere da diversi calcolatori.

Per risolvere questo e altri problemi viene in soccorso il protocollo IMAP. Un server che adotta il protocollo IMAP associa ogni messaggio arrivato al server in una cartella. L'utente può crearne di altre e spostare i messaggi al loro interno. IMAP fornisce anche comani per consentire agli utenti di effettuare ricerche in cartelle remote. Un'altra caratteristica è che consente di scaricare solo singole parti dei messaggi.

#### Posta basata sul Web

Al giorno d'oggi gli utenti possono inviare e leggere posta su un browser Web. In questo caso lo user agent è un browser web e l'utente comunica con il proprio server web via HTTP.

In ogni caso, anche se gli utenti finali comunicano con i loro server via HTTP, i server tra di loro comunicano via SMTP.

## 2.4 DNS: il servizio di directory di Internet

Gli host di internet possono essere identificati possono essere indentificati in due modi: tramite il nome host (es. www.eff.org), che sono facili da ricordare per le persone ma non adatti per un sistema di routing e tramite indirizzo IP, difficili da ricordare ma adatto per il sistema di routing in quanto otteniamo informazioni più dettagliate all'interno del collocamento della macchina nella rete ad esso associato.

Ogni indirizzo IP è formato da 4 byte con un punto che divide ogni byte espresso in decimale da 0 a 255 (es. 192.168.10.255). Gli indirizzi IP sono gerarchici perché leggendoli da sinistra verso destra otteniamo informazioni sempre più specifiche sulla collocazione dell'host all'interno di Internet.

### 2.4.1 Servizi forniti da DNS

Come poc'anzi anticipato, le persone prediligono usare un nome host mentre i router prediligono usare un indirizzo IP.

Al fine di conciliare i due approcci, ovvero far usare alle persone l'hostname e ai router l'indirizzo IP, è necessario un servizio che sia in grado di tradurre i nomi degli host nei loro indirizzi IP. Si tratta del compito del DNS. DNS è (1) un database distribuito e (2) un protocollo a livello di applicazione che consente agli host di interrogare il database.

Il DNS usa UDP sulla porta 53 e viene comunemente utilizzato da altri protocolli a livello di applicazione come HTTP e SMTP.

Questo è il funzionamento quando si richiede una pagina Web:

1. Il browser estrae il nome dell'host dall'URL e lo passa al lato client dell'applicazione DNS
2. Il client DNS invia un'interrogazione contenente l'hostname a un server DNS
3. Il client riceve una risposta che include l'indirizzo IP corrispondente all'hostname
4. Una volta ricevuto l'indirizzo IP dal DNS, il browser dò inizio a una connessione TCP sulla porta 80 a quell'indirizzo

Pertanto, il sistema DNS aggiunge un ritardo aggiuntivo alle applicazioni Internet che lo utilizzano. Oltre alla traduzione degli hostname in indirizzi IP, il DNS mette a disposizione altri servizi:

- Host aliasing: Un host dal nome complicato (hostname canonico) può avere uno o più sinonimi (alias). Il DNS può essere invocato per ottenere l'hostname canonico da un alias
- Mail server aliasing: simile all'host aliasing ma con piccole differenze che affronteremo più avanti nella trattazione
- Distribuzione del carico di rete: I siti con molto traffico vengono replicati su più server, ognuno eseguito su un host diverso con un indirizzo IP diverso. Viene dunque associato un hostname a un INSIEME di indirizzi IP. Quando un client effettua una query per questo hostname, il server risponde con l'intero insieme di indirizzi, ma di volta in volta stravolgendone l'ordine. Dato che generalmente un client invia il suo messaggio al primo indirizzo IP elencato, ciò contribuisce a distribuire il traffico su server replicati.

### 2.4.2 Panoramica del funzionamento di DNS

Tutte le query DNS e i messaggi di risposta vengono inviati all'interno di datagrammi UDP alla porta 53. Il client DNS sull'host riceve un messaggio di risposta contenente la corrispondenza desiderata che viene inoltrata all'applicazione che ne ha fatto richiesta. Pertanto, dal punto di vista dell'applicazione, la risoluzione dell'hostname è una scatola nera.

Per implementare il servizio DNS un primo approccio potrebbe essere quello di creare un DNS server contenente tutte le corrispondenze. Ciò sarebbe inappropriato per i seguenti motivi: un solo punto di fallimento, volume di traffico spropositato per un solo nodo, database geograficamente distante e manutenzione (dovrebbe essere aggiornato frequentemente per tener conto di ogni nuovo host).

#### Un database distribuito e gerarchico

In realtà il servizio DNS utilizza un gran numero di server, organizzati in maniera gerarchica e distribuiti. Nessun DNS server ha tutte le corrispondenze per tutti gli host di Internet. Questa gerarchia consiste di tre classi di DNS server: i root server, i top-level domain server e i server autoritativi.

Esaminiamo più in dettaglio le varie classi di server DNS:

- Root server: ne sono circa 400 in tutto il mondo e forniscono gli indirizzi IP dei TLD DNS Server
- TLD server: Questi server si occupano dei domini di primo livello quali org, com, edu, it, fr ecc. Sa quali server sono autoritativi per i domini sotto quel TLD
- Server autoritativi: Contiene gli indirizzi IP dei server per un determinato dominio.

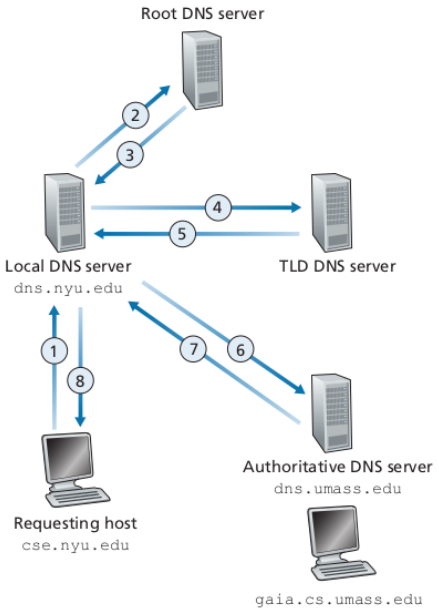
**Esempio** Si ipotizzi di voler risolvere www.example.com.

1. Il client chiede al root DNS server il TLD DNS che gestisce il top level domain .com
2. Il client chiede al TLD DNS server il DNS autoritativo che gestisce il dominio example.com
3. I client chiede al DNS autoritativo l'IP per www.example.com

Esiste un altro tipo di server DNS detto DNS server locale, che non appartiene alla gerarchia di server sopra descritta. Ciascun ISP ha un proprio DNS server locale, fornito ai clienti che vi si connettono. Quando un host effettua una richiesta DNS, la query viene inviata al DNS server locale, che opera da proxy e inoltra la query alla gerarchia DNS.

Ora vediamo più in dettaglio cosa avviene quando si tenta di risolvere www.domain.it

1. L'utente invia una richiesta DNS al server locale
2. Il server locale inoltra la query al root server, che, vedendo che il TLD è .it, risponde con uno o più indirizzi IP appartenenti ai TLD server che si occupano di .it
3. Il server locale, dunque, invia la richiesta a uno dei TLD server che risponde con uno o più indirizzi IP di DNS server autoritativo per domain.it
4. Il DNS server locale, annotato ciò, invia la richiesta a uno di questi, il quale risponde con l'indirizzo IP dell'hostname finale
5. Il DNS server locale inoltra questa informazione all'host richiedente.




Esempio di funzionamento

In questo esempio si fa uso sia di query ricorsive che di query iterative. La richiesta effettuata dal client al DNS server locale è ricorsiva in quanto il client chiede a quest'ultimo di ottenere l'associazione dell'hostname (cioè non rimbalza al client un risultato intermedio che ha trovato). Tutte le altre richieste sono iterative in quanto server A chiede a B di risolvere solo una parte dell'hostname. Server B gli risponde e sarà compito di A procedere nella ricerca. Ogni richiesta DNS può essere o iterativa o ricorsiva.

#### DNS caching

Il server DNS che riceve una risposta da un altro server DNS può mettere in cache le informazioni ricevute, in modo che successivamente possa rispondere senza interrogare altri server e conseguentemente incrementare i ritardi. Dopo un determinato periodo di tempo i server DNS svuotano l'informazione ricevuta. Ad esempio un DNS server locale può memorizzare in cache l'indirizzo IP dei TLD DNS server, aggirando la richiesta ai root DNS server.

### 2.4.3 Record e messaggi DNS

Un server DNS risponde con i cosiddetti resource record che sono così composti:
`(Name, TTL, Type, Value)`

Il TTL è il time to live e indica il tempo residuo in cache prima di essere rimosso. Il significato di `Name` e `Value` dipende da `Type`.

- Se `Type=A` allora `Name` è il nome dell'host e `Value` è il suo indirizzo IP. Pertando un record di tipo A fornisce la corrispondenza tra hostname e indirizzo IP.

- Se `Type=NS` (Nameserver) allora `Name` è un dominio e `Value` è l'hostname del DNS server autoritativo o del DNS TLD per quel dominio
- Se `Type=CNAME` (Canonical Name) allora `Name` è l'alias e `Value` è il nome canonico
- Se `Type=MX` (Mail Exchange) allora `Value` è il nome canonico di un mail server che ha `Name` come alias. Può sembrare simile a `CNAME` ma ci sono importanti differenze: quest'ultimo non permette ad un alias di avere più nomi canonici, mentre `MX` lo permette per una questione di fail-over (cioè se un host identificato da un nome canonico non funziona, prova con un altro). Per questo motivo `Type=MX` permette anche di specificare una priorità di scelta.

**Esempi**

**Type=A** `www.example.com. 3600 A 203.0.113.10`
(www.example.com è raggiungibile a 203.0.113.10)

**Type=NS** `example.com. 172800 NS ns1.dns-host.net.`
(example.com è un dominio e ns1.dns-host.net è l'hostname del nameserver autoritativo per quel dominio)

**Type=NS** `blog.example.com. 300 CNAME ghs.googlehosted.com.`
(blox.example.com è un alias per ghs.googlehosted.com)

**Type=MX** `example.com. 3600 MX 10 mx1.mailhost.net.`
(Le email inviate a example.com devono essere inviate a mx1.mailhost.net con priorità 10. Potrebbe esserci un altro record per esempio con priorità 5. In quel caso le email verrebbero inviate prima a quello)

Notare che quando interroghi un Root DNS server non ottieni di default l'indirizzo IP del TLD DNS. Allo stesso modo quando interroghi un TLD DNS non ottieni di default l'indirizzo IP del DNS autoritativo. Ciò comporta un'ulteriore query DNS per trovare l'IP di questi server. Per evitare una query aggiuntiva, nella stessa risposta i server possono includere nella sezione Additional gli A/AAAA dei nameserver indicati.

Inoltre si noti che un TLD DNS è semplicemente un DNS autoritativo per il dominio .com (ad esempio). Inoltre vi possono essere server autoritativi per east.example.com e west.example.com che restituiscono record A per www.east.example.com e www.west.example.com, ovverosia possono esserci più di 3 server DNS sul cammino.

Ogni server DNS autoritativo contiene un record A per un certo hostname ma anche un altro tipo di server DNS può contenere record A nella propria cache.

### Messaggi DNS

Sia le query che le risposte hanno lo stesso formato

### Inserimento di un record nel database DNS

Come vengono inseriti i record nel database DNS? Un utente che vuole registrare un nome di dominio deve rivolgersi presso un registrar che si occupa di verificare l'unicità del nome di dominio e di inserirlo nel database DNS. Quando l'utente registra un nome di dominio presso un registrar bisogna fornirgli anche i nomi e gli indirizzi IP dei DNS server autoritativi.

Fino a poco tempo fa i contenuti di ciascun server DNS venivano inseriti in maniera statica per mezzo di un file di confiurazione. Con le nuove versioni del protocollo DNS è stata aggiunta l'opzione UPDATE per consentire l'aggiunta o la cancellazione dinamica dei dati del database attraverso messaggi DNS.

## 2.5 Distribuzione di file P2P

Nell'architettura P2P ci sono coppie di host connessi spesso in modo intermittente, chiamati peer, che comunicano direttamente l'uno con l'altro. I peer non appartengono ai fornitori di servizi, ma sono computer controllati dagli utenti.

Un'applicazione particolarmente adatta per una struttura P2P è la distribuzione di un file voluminoso. In una distribuzione di file client-server, il server deve inviare una copia del file a ciascun client, ponendo

un enorme fardello sul server e consumandone un'elevata quantità di banda. In una distribuzione di file P2P ciascun peer può ridistribuire agli altri qualsiasi porzione del file abbia ricevuto. Uno fra i più diffusi protocolli di distribuzione file P2P è BitTorrent.

**Scalabilità dell'architettura P2P**

Illustriamo ora la scalabilità dell'architettura P2P confrontandola con l'architettura client-server.

Definiamo il tempo di distribuzione essere il tempo richiesto affinché tutti gli $N$ client (o peer) ottengano una copia del file distribuito dall' $(N+1)$-esimo partecipante alla rete.

Facciamo dapprima un'analisi dell'architettura client-server.

Sia $u_s$ la banda di upload del collegamento di accesso del server, $u_i$ la banda di upload del collegamento di accesso dell'i-esimo client, $d_i$ la banda di download del collegamento di accesso dell'i-esimo client, $F$ la dimensione in bit del file da trasferire e $N$ il numero di client che vuole una copia del file. Ipotizziamo, inoltre, che il nucleo di Internet abbia banda superiore a qualunque $u$ e $d$: ciò implica che tutti i colli di bottiglia siano nelle reti di accesso.

Il tempo di distribuzione del file per l'architettura client-server la denotiamo con $D_{cs}$

Facciamo le seguenti osservazioni:

- Il server deve trasmettere una copia dei file a ciascuno degli $N$ peer, cioè $NF$ bit. Dato che la banda di upload del server è $u_s$, il tempo per distribuire il file è almeno $\frac{NF}{u_s}$.
- Sia $d_{min}$ la banda di download più bassa tra tutti i client. Ovverosia $d_{min} = \{d_1, d_2, ..., d_N\}$. Allora questo client non può ricevere tutti i bit in meno di di $F/d_{min}$ secondi.

Mettendo insieme queste due osservazioni avremo che:

$$D_{cs} \geq MAX \left\{ \frac{NF}{u_s}, \frac{F}{d_{min}} \right\}$$

Ciò fornisce un limite inferiore al tempo di distribuzione per l'architettura client-server ma è possbile che $D_{cs}$ sia esattamente la parte destra della disequazione? Assolutamente sì, ottenendo

$$D_{cs} = MAX \left\{ \frac{NF}{u_s}, \frac{F}{d_{min}} \right\}$$

nel caso in cui nessun nodo stia occupando la propria banda per inviare/ricevere alti bit e che il cuore di Internet sia sufficientemente grande.

Per $N$ sufficientemente grande, il tempo di distribuzione è dato dal termine $\frac{NF}{u_s}$ ciò implica che da un $N$ in poi, il tempo di distribuzione minimo aumenta linearmente con $N$.

Ora svolgiamo un'analisi analoga per l'architettura P2P, mantenendo le stesse ipotesi del modello client/server. La differenza è che, invece di un "server", consideriamo un peer che avvia la trasmissione ed è l'unico a possedere inizialmente il file che chiamiamo $P_1$.

Ora facciamo le seguenti osservazioni:

- All'inizio della distribuzine solo $P_1$ dispone del file. Esso deve necessariamente inviare ciascun bit del file almeno una volta. Quinid il tempo di distribuzione minimo è $F/u_s$. Si noti che un bit inviato una volta dal server potrebbe non essere inviato di nuovo (ecco perché non è $NF/u_s$)
- Il peer con la velocità di download più bassa non può ottenere tutti i bit del file in meno di $F/d_{min}$. Quindi il tempo di distribuzione è perlomeno $F/d_{min}$
- Devono essere consegnati $F$ bit a ciascuno degli $N$ peer per un totale di $NF$ bit. Questi $NF$ bit non possono viaggiare sulla rete a una velocità più elevata di $u_{tot} = u_s + u_1 ... + u_N$ quindi il tempo di distribuzione minimo è almeno $NF/u_{tot}$

Mettendo insieme queste considerazioni otteniamo che

$$D_{P2P} \geq MAX \left\{ \frac{F}{U_s}, \frac{F}{d_{min}}, \frac{NF}{u_{tot}} \right\}$$

Si può dimostrare che è una disuguaglianza stretta, ovvero è possibile schedulare il trasferimento del file in modo tale che

$$D_{P2P} = MAX \left\{ \frac{F}{U_s}, \frac{F}{d_{min}}, \frac{NF}{u_{tot}} \right\}$$

La dimostrazioneè in un paper denominato "Peer-Assisted File Distribution: The Minimum Distribution Time (Rakesh Kumar, Keith W. Ross)". La dimostrazione mostra che la disuguaglianza è in effetti stretta. Suddetta dimostrazione è particolarmente laboriosa da un punto in poi e si preferisce darla per buona.

Inoltre, si può dimostrare che $D_{P2P} < D_{cs}$ per ogni $N \geq x$, cioè da un numero di peer/client in poi il tempo di distribuzione minimo nell'architettura Peer to Peer è sempre minore dell'architettura Client/Server, per quanto veloce siano i calcolatori e le reti di quest'ultima.

**BitTorrent**

BitTorrent è un protocollo P2P per la distirbuzione di file. L'insieme di tutti i peer che partecipano alla distribuzione di un particolare file è chiamato torrent. I peer in un torrent si scambiano l'un l'altro chunk di file di uguale dimensione.

Quando un peer entra a far parte di un torrent per la prima volta, non ha alcun chunk. Col passare del tempo accumula sempre più parti che, mentre scarica, invia agli altri peer, anche se il file non è stato ancora scaricato del tutto. Una volta che un peer ha acquisito un intero file può egoisticamente lasciare il torrent o altruisticamente rimanerci dentro e continuare ad aiutare nella distribuzione.

Qualunque peer può lasciare il torrent in qualsiasi momento, anche solo con un sottoinsieme di chunk del file completo e rientrare a far parte del torrent in seguito.

Per ciascun torrent vi è un nodo chiamato tracker. Quando un peer entra a far parte di un torrent, si registra presso il tracker e periodicamente lo informa che è ancora nel torrent. In questo modo il tracker tiene traccia dei peer che stanno partecipando al torrent.

Si supponga Alice decida di unirsi al torrent. Il tracker seleziona in modo casuale un sottoinsieme di peer dall'insieme dei peer che stanno partecipando a quel torrent e invia gli indirizzi IP di questi host ad Alice, la quale stabilisce contrmporaneamente con loro delle connessioni TCP. Chiamiamo questo sottorinsieme "peer vicini". I peer vicini cambiano nel tempo perché (1) potrebbero disconnettersi e (2) dei peer potrebbero stabilire di loro iniziativa connessioni con Alice.

Periodicamente Alice chiede ai suoi peer vicini la lista del chunk del file in loro possesso e richiede quelli che ancora le mancano. La modalità con cui richiede i chunk si chiama rarest first, ovvero richiede per prima quelli con il minor numero di copie tra i peer. In questo modo si tenta di mantenere il numero di copie per ogni chunk simile.

Ora che Alice ha un algoritmo che le dica da chi e come ottenere i chunk, similmente ha bisogno di un algoritmo che le dica a chi inviare i chunk. L'idea di base è che Alice attribuisca priorità ai vicini che le stanno mandando i chunk alla velocità più alta: per ciascuno dei suoi vicini, Alice misura costantemente la velocità a cui riceve dati e determina i 4 peer che le stanno inviando i chunk alla velocità più elevata e manda a questi i chunk che le richiedono. Ogni 10 secondi ricalcola la velocità. Questi 4 peer vengono chiamati unchoked. Inoltre ogni 30 secondi, tra i propri vicini, sceglie un vicino casuale in più al quale gli invia i chunk. Questo vicino in più si chiama optimistically unchoked. Ciò consente ai nuovi peer senza alcun chunk di iniziare a ottenerne qualcuno.

Due peer continueranno a effettuare scambi tra loro finché uno non trovi un partner migliore. L'effetto è che i peer in grado di inviare dati a velocità simili tendono ad accoppiarsi.

Tutti i peer ad eccezione di quei cinque sono detti choked e quindi non ricevono nessun chunk da Alice.

Questo meccanismo di incentivazione viene detto tit-for-that.

Oltre ai tracker esistono anche le DHT (Distributed Hash Table) che assolvono lo stesso compito. Sono database distributi fra tutti i peer.

## 2.6 Streaming video e reti per la distribuzione di contenuti

### 2.6.1 Video su Internet

Un video è una sequenza di immagini visualizzate a velocità costante. La caratteristica più saliente dei video è l'elevato throughput con cui è necessario inviare bit sulla rete. Ciò può tradursi in enormi quantità di traffico e di spazio di archiviazione. Il throughput medio deve essere più alto del bitrate del video per avere una riproduzione continua.

### 2.6.2 Streaming HTTP e DASH

Nello streaming HTTP un video viene memorizzato in un server come un file ordinario referenziato da un URL. Il client stabilisce una connessione TCP con il server al quale invia una richiesta GET sull'URL specificato.

Istante dopo istante i byte ricevuti vengono memorizzati in un buffer. Quando il numero di byte nel buffer supera una soglia prefissata, l'applicazione client inizia la riproduzione.

Lo streaming HTTP presenta un grande svantaggio: i client ricevono sempre la stessa versione del video indipendentemente dalla larghezza di banda che hanno a disposizione. Per superare questo problema è stato sviluppato un nuovo tipo di streaming basato su HTTP chiamato DASH. In DASH, i video vengono codificati in varie versioni, ognuna a un bitrate diverso. Il client richiede segmenti in modo dinamico: quando ha più banda richiede video di maggiore qualità e viceversa. I video sono sempre memorizzati su un server HTTP, ognuno identificato da un URL diverso. Oltre ai video, però, vi è anche un file di manifesto che contiene, per ogni versione del video, il bitrate e l'URL sul quale raggiungerlo. Il client seleziona a ogni istante un blocco. Mentre scarica un blocco, il client misura la banda di ricezione e, con questa informazione, al prossimo istante seleziona il blocco più adeguato.

### 2.6.3 Reti per la distribuzione di contenuti

L'approccio più diretto per le aziende di streaming sarebbe quello di creare un enorme server nel quale memorizzare tutti i video e mandarli in streaming a chiunque li richieda. Questo presenta tre svantaggi principali: (1) il client potrebbe essere lontano dal server. Più il cammino di un pacchetto è lungo più è probabile incontrare un collo di bottiglia, nonché i vari ritardi si accumulano; (2) un video molto popolare verrebbe inviato tante volte sullo stesso collegamento ; (3) c'è un singolo punto di rottura.

Per superare questi problemi si usano le CDN (content distirbution network). Una CDN gestisce server distribuiti e memorizza copie dei video e di altri contenuti in uno o più. Quando un utente richiede una risorsa, la CDN lo regirige verso il server in grado di offrire il servizio migliore.

Vi sono due approcci alla dislocazione dei server della CDN:

- Enter deep: entrare profondamente nelle reti di accesso, installando server direttamente negli ISP di accesso sparsi in tutto il mondo, in questo modo diminuendo il numero di collegamenti tra l'utente finale e il server. Il rovescio della medaglia è che richiede una forte manutenzione.
- Bring home: si costruiscono server in pochi punti chiave posti in luoghi ai PoP degli ISP di livello 1. Qui vi è meno manutenzione ma meno qualità del servizio.

I video raramente richiesti non vengono copiati su tutti i cluster, ma usano una strategia per cui se un client richiede un video a un cluster che non lo ha memorizzato, quest'ultimo lo recupera da un archivio centrale o da un altro cluster e ne memorizza una copia mentre lo manda in streaming al client. I video meno richiesti vengono rimossi quando lo spazio disponibile si esaurisce.

#### Come funziona la CDN

Quando un browser chiede di recuperare uno specifico video, la CDN deve intercettare la richiesta in modo da poter determinare il cluster più appropriato per quel cliente a quell'istante e dirigere la richiesta in uno dei server del cluster.

Vediamo cosa succede quando l'utente chiede https://www.youtube.com/XYZABC:

- Il suo host invia una richiesta DNS a www.youtube.com
- Il server autoritativo di YouTube passa risponde con un server autoritativo di KingCDN tipo kingcdn.com
- KingCDN rileva XYZABC e risponde al LDNS con il server più adatto
- Il client si collega a questo server

#### Strategie di selezione dei cluster

La CDN deve selezionare un cluster adatto.

Una semplice strategia consiste nell'assegnare a un client il cluster geograficamente più vicino. Gli indirizzi IP dell'LDNS vengono assegnati a un luogo specifico e la CDN risponde con il cluster più vicino.

Tuttavia, per alcuni client potrebbe non andare bene perché il cluster più vicino geograficamente potrebbe essere diverso da quello più vicino dal punto di vista della rete. Inolte, l'LDNS potrebbe trovarsi lontano dal client.

Un secondo approccio determina il cluster migliore per un client basandosi sulle condizioni di traffico correnti effettuuando misure in tempo reale delle prestazioni tra i loro cluster e i client. Per esempio, facendo sondare periodicamente dai propri cluster tutti gli LDNS del mondo. Il problema di tale approccio è che molti LDNS sono configurati per non rispondere a tali richieste per non parlare della complessità della cosa.

# Capitolo 3

# Livello di trasporto

Il livello di trasporto fornisce la funzione di fornire servizi di comunicazione tra i processi applicativi in esecuzione su host differenti. Si tratta di estendere il servizio offerto dal livello di rete che invece fornisce solo il servizio di comunicazione tra gli host.

## 3.1 Introduzione e servizi a livello di trasporto

Un protocollo a livello di trasporto mette a disposizione una comunicazione logica tra processi applicativi su host differenti.

Per comunicazione logica si intende, dal punto di vista dell'applicazione, che tutto proceda come se i processi fossero direttamente connessi perché non si preoccupano dei dettagli dell'infrastruttura.

I protocolli a livello di trasporto sono implementati nei sistemi periferici e quindi non nei router della rete.

Lato mittente, il livello di trasporto converte i messaggi che riceve da un processo applicativo in pacchetti a livello di trasporto detti segmenti, spezzando (se necessario) i messaggi in parti più piccole a aggiungendo informazioni header che a sua volta lo passa al livello di rete.

I router agiscono solo sul datagramma senza esaminare i campi del segmento incapsulato al suo interno.

### 3.1.1 Relazione tra i livelli di trasporto e la rete

Ribadiamo che un protocollo a livello di trasporto mette a disposizione una comunicazione logica tra processi che vengono eseguiti su host diversi (usando meccanismi interni di smistamento) mentre un protocollo a livello di rete fornisce comunicazione logica tra host (usando meccanismi interni di smistamento, quali il routing).

Un protocollo di trasporto non fornisce alcuna indicazione su come i messaggi siano trasferiti all'interno della rete.

Si noti che una rete può offrire più protocolli di trasporto ciascuno dei quali può offrire alle applicazioni di rete un servizio differente.

### 3.1.2 Panoramica del livello di trasporto di Internet

Internet offre due protocolli di trasporto:

- UDP (User Datagram Protocol): fornisce alle applicazioni un servizio non affidabile e non è orientato alla connessione. Fornisce un controllo degli errori molto semplice.
- TCP (Transmission Control Protocol): È orientato alla connessione e offre un servizio affidabile. Offre, inoltre, un servizio di controllo della congestione regolando la velocità alla quale il lato mittente della connessione TCP immette traffico in rete

TCP offre servizio di trasporto dati affidabile anche quando il sottostante protocollo di rete non è affidabile, infatti quest'ultimo prevede un servizio "best effort" ovvero fa del suo meglio per consegnare i pacchetti tra host comunicanti ma non offre alcuna garanzia: nè che i pacchetti arrivino in ordine nè che arrivino affatto.

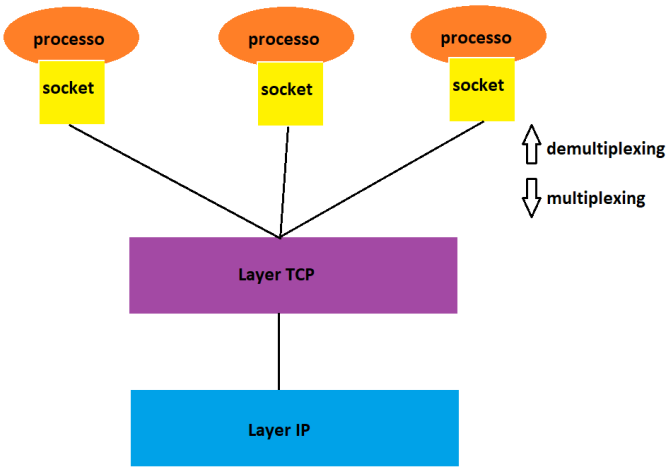
Il principale compito di TCP e UDP è l'estensione del servizio di consegna tra sistemi periferici (offerto da IP) a quello di consegna tra processi in esecuzione su sistemi periferici. Questa funzionalità è detta multiplexing o demultiplexing a seconda della direzione.

## 3.2 Multiplexing e demultiplexing

Il multiplexing e il demultiplexing sono i metodi con cui il servizio di trasporto da host a host fornito dal livello di rete diventa un servizio di trasporto da processo a processo.

Nell'host destinatario il livello di trasporto riceve segmenti dal livello di rete immediatamente sottostante. Il livello di trasporto ha il compito di consegnare i dati di questi segmenti al processo applicativo appropriato. In realtà non consegna i dati direttamente al processo ma piuttosto alla socket che funge da intermediario. Siccome a ogni dato istante possono esserci più socket sull'host di ricezione, ciascuna avrà un identificatore univoco.

Il compito di trasportare i dati dei segmenti a livello di trasporto verso la giusta socket (dati in ingresso) viene detto demultiplexing. Viceversa, il compito di radunare dati da diverse socket e incapsulare ognuno con intestazioni a livello di trasporto per creare segmenti (dati in uscita) viene detto multiplexing.




Multiplexing e demultiplexing

Gli identificatori univoci delle socket vengono dette porte, composti da 16 bit. I numeri da 0 a 1023 sono chiamati numeri di porta noti e sono riservati per essere usati da protocolli applicativi ben noti (es. HTTP sulla porta 80).

Quando un segmento arriva all'host di destinazione, il protocollo a livello di trasporto esamina il numero della porta di destinazione e dirige il segmento verso la socket corrispondente.

Generalmente il lato client usa un numero casuale di porta mentre lato server si assegna un numero di porta specifico. Questo si fa poiché il mittente deve conoscere dove contattare il processo con il quale vuole interagire, mentre il destinatario risponde semplicemente al numero di porta del mittente che ha iniziato la connessione. Si immagini che A voglia instaurare una connessione HTTP con B: allora A contatta B sulla porta 80 e mette nell'intestazione del segmento un numero di porta di origine casuale. Il server B, quando vuole rispondere ad A, farà assumere il valore del numero di porta di destinazione il numero di porta di origine che aveva ricevuto.

### Multiplexing e demultiplexing non orientati alla connessione (UDP)

Una socket UDP ricevente viene identificata completamente da una coppia che consiste di un indirizzo IP ricevente e un numero di porta ricevente.

### Multiplexing e demultiplexing orientati alla connessione (TCP)

Una differenza tra una socket UDP e una TCP risiede nel fatto che quest'ultima, a differenza della socket UDP, è identificata da quattro parametri: indirizzo IP di origine, porta di origine, indirizzo IP di

destinazione, porta di destinazione. Eccezion fatta per i segmenti TCP che trasportano la richiesta per stabilire la connessione.

Ne consegue che lato ricevente, a fronte di tutti i parametri che restano uguali tranne l'inidirzzo IP o numero di porta mittente, i pacchetti verrano rediretti a un altro processo.

L'host server può ospitare più socket TCP contemporaneamente, ognuna identificata da una specifica quaterna di valori.

Perché si fa così? Perché TCP è orientato alla connessione e deve poter distinguere connessioni diverse. Si consideri la seguente tabella:

| **Client IP** | **Client Porta** | **Server IP** | **Server Porta** |
|---|---|---|---|
| 10.0.0.2 | 50000 | 93.184.216.34 | 80 |
| 10.0.0.3 | 50000 | 93.184.216.34 | 80 |

Anche se Server IP e Server Porta sono identici, gli indirizzi IP dei client cambiano, pertanto il server ha due socket diverse ognuna delle quali con una connessione differente. Si noti che in TCP non c'è una corrispondenza 1:1 tra numero di porta e socket. In questo esempio, infatti, c'è un solo numero di porta ma con due socket.

Inoltre, non sempre esiste una corrispondenza 1:1 tra le socket e i processi. Infatti, si possono avere più socket collegate allo stesso processo.

Ricapitolando, un numero di porta può avere più socket. Più socket possono essere collegate allo stesso processo.

## 3.3 Trasporto non orientato alla connessione: UDP

Un livello di trasporto deve quantomeno fornire un servizio di multiplexing e demultiplexing al fine di trasferire i dati tra il livello di rete e il processo corretto a livello di applicazione.

A parte questa funzionalità e un controllo molto semplice sugli errori, UDP non aggiunge nient'altro a IP. Infatti, prende i messaggi dal processo applicativo, aggiunge il numero di porta di origine e destinazione e passa il segmento risultante al livello di rete sottostante. Se il segmento arriva a destinazione, UDP utilizza il numero di porta di destinazione per consegnare i dati al processo applicativo corretto. In UDP, inoltre, non vi è handshaking.

Perché preferire UDP, che non offre molto, rispetto a TCP? La risposta è che molte applicazioni sono più adatte con UDP per i seguenti motivi:

- Controllo più fine a livello di applicazione su quali dati sono inviati e quando: nonappena un processo applicativo passa dei dati a UDP, questi li impacchetta in un segmento che trasferisce immediatamente al livello di rete. TCP, invece, introduce in meccanismo di controllo della congestione che ritarda l'invio. Inoltre quest'ultimo manda continuamente segmenti sulla rete fin quando la destinazione non ha notificato l'avvenuta ricezione. TCP poi effettua un handshake, mentre UDP no e per tale motivo non introduce alcun ritardo nello stabilire una nuova connessione. Dato che le applicazioni in tempo reale spesso richiedono una velocità minima di trasmissione, non sopportano ritardi ma sopportano piccole perdite, TCP mal si adatta a quest'ultime.

- Nessun stato di connessione: TCP mantiene lo stato della connessione nei sistemi periferici. Inoltre conserva anche buffer di ricezione e di invio, parametri per il controllo della congestione e parametri sul numero di sequenza e di acknowledgment. UDP, invece, non conserva lo stato della connessione e non tiene traccia di alcun parametro e pertanto risulta più leggero.

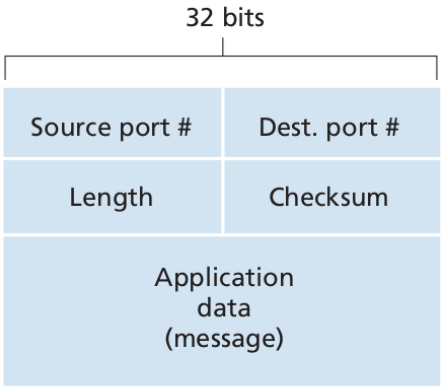
- Minor spazio usato per l'intestazione del segmento: l'intestazione TCP aggiunge 20 byte mentre UDP ne aggiunge 8.

È interessante notare che le applicazioni possono ottenere, volendo, un trasferimento affidabile anche su UDP se tale algoritmo è insito nel protocollo applicazione anziché quello di trasporto. Questo permette ai processi applicativi di poter comunicare in modo affidabile senza essere soggetti ai vincoli di TCP.

### 3.3.1 Struttura dei segmenti UDP

In un segmento UDP ci sono 5 campi:

- Numero di porta di origine
- Numero di porta di destinazione
- Lunghezza: numero di byte del segmento UDP (intestazione + dati)
- Checksum: serve per controllare se sono avvenuti errori. Più dettagli in seguito
- Dati: i dati dell'applicazione vengono conservati qui




Struttura di un segmento UDP

### Checksum UDP

Il checksum in UDP viene utilizzato per determinare se i bit del segmento sono stati alterati durante il loro trasferimento.

Il mittente effettua il complemento a 1 della somma di tutte le parole da 16 bit e conserva tale dato nel campo checksum (in altre parole suddivide l'intero segmento in porzioni da 16 bit, ne fa la somma e ne salva all'interno il complemento a uno [gli 0 diventano 1 e gli 1 diventano 0]). Il ricevente usa il complemento per verificare che il risultato della somma tra le somme delle parole che ha ricevuto e il complemento nel pacchetto faccia 111.... Se c'è almeno uno zero allora si è verificato un errore.

Ci si potrebbe chiedere perché UDP metta a disposizione un checksum se i protocolli a livello di collegamento (Ethernet, Wifi ecc) facciano lo stesso. I motivi sono sostanzialmente due: (1) non c'è garanzia che tutti i collegamenti sul percorso controllino gli errori e (2) l'errore si potrebbe verificare mentre il segmento è dentro un router.

Sebbene UDP metta a disposizione tale funzionalità, non fa nulla per risolvere le situazioni di errore.

## 3.4 Principi del trasferimento dati affidabile

Con un canale affidabile nessun bit dei dati trasferiti viene corrotto o va perso e tutti i bit sono consegnati nell'ordine di invio.

Il compito di un protocollo di trasferimento dati affidabile è quello di implementare questo canale anche se il protocollo al livello inferiore è inaffidabile. Ad esempio TCP è un protocollo di trasferimento affidabile implementato al di sopra di IP, notoriamente inaffidabile.

### 3.4.1 Costruzione di un protocollo di trasferimento dati affidabile

Svilupperemo delle Finite State Machines di un protocollo di trasferimento dati affidabile. Assumeremo che i pacchetti vengano consegnati nell'ordine in cui sono stati inviati, sebbene alcuni possano venir persi o danneggiati.

Il lato mittente del protocollo verrà invocato dal livello superiore tramite una chiamata `rdt_send` ("rdt" sta per Reliable Data Transfer) e invocherà `udt_send` ("udt" sta per Unreliable Data Transfer) del protocollo sottostante. In altre parole il livello $n+1$ chiede al livello $n$ di inviare dati in modo affidabile. Dato che il livello $n-1$ potrebbe non esserlo, il livello $n$ deve escogitare dei meccanismi per far sì di adempire al compito richiesto. Quando un pacchetto raggiungerà il lato ricevente, verrà invocato `rdt_rcv` e passerà i dati al livello superiore tramite `deliver_data`. Senza mancare di generalità ma semplificando il discorso tratteremo solo il caso di un trasferimento **dati** unidirezionale. Si noti che mittente e destinatario abbiano comunque bisogno di scambiarsi informazioni di **controllo** in entrambe le direzioni.

La teoria sviluppata in queste pagine si applica alla reti di calcolatori in generale e non solo al livello di trasporto e per tale motivo abbiamo usato il termine generale "pacchetto" anziché "segmento".

#### Trasferimento dati affidabile su un canale perfettamente affidabile: RDT 1.0

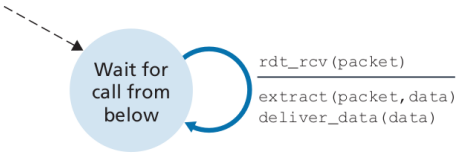
Innanzitutto tratteremo il caso più semplice, in cui il canale sottostante è completamente affidabile, ovverosia `udt_send` è già di per sè affidabile. Il protocollo che svilupperemo, denominato RDT 1.0 è banale.

Prima di procedere allo sviluppo dell'algoritmo facciamo le seguenti osservazioni:

- Esistono due FSM separate, una per il mittente e una per il destinatario
- Lo stato iniziale di una FSM è indicato da una freccia tratteggiata
- L'evento che causa la transizione è scritto sopra una linea orizzontale posta sulla freccia di transizione mentre le azioni intraprese a seguito di tale evento sono scritte sotto la linea orizzontale
- Quando un evento non determina un'azione e quando un'azione viene scaturita senza un evento useremo la lettera $\Lambda$ sotto la linea orizzontale per denotare la mancanza di un'azione e sopra la linea orizzontale per denotare la mancanza di un evento

La FSM per RDT 1.0 è molto semplice:


a. rdt1.0: sending side




b. rdt1.0: receiving side

RDT 1.0

Ciò sta a significare che quando viene invocata `rdt_send` lato mittente, quest'ultimo crea un pacchetto con i dati (`data`) ottenuti dall'alto e invia il pacchetto sul canale dati "inaffidabile" (che in questo caso è affidabile) tramite `udt_send(packet)`

Lato ricevente, viene invocato `rdt_rcv`. Il destinatario non deve far altro che estrarre i dati dal pacchetto e inviarli al livello superiore.

**Trasferimento dati affidabile su un canale con errori sui bit: RDT 2.0**

Adesso ipotizziamo che il canale sottostante possa corrompere i bit, ma non perdere l'intero pacchetto.

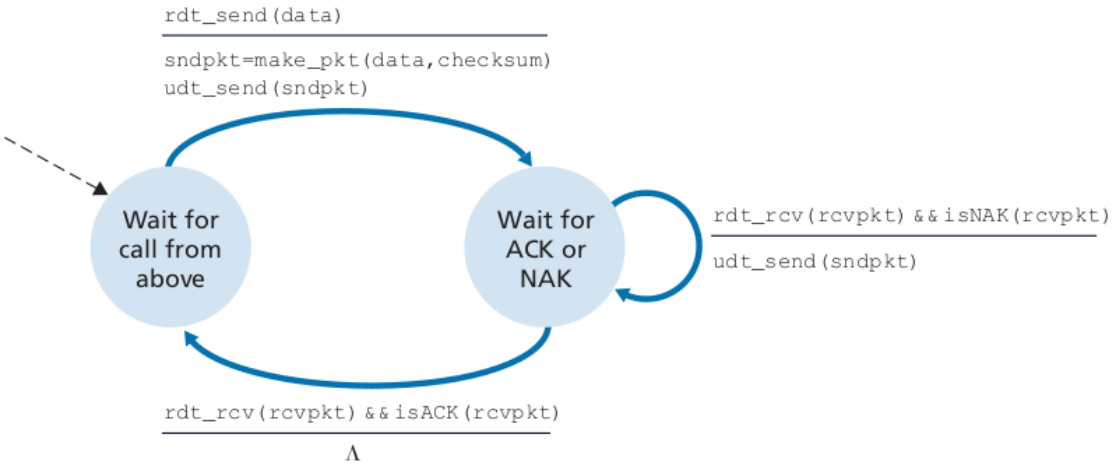
Immaginiamo cosa accadrebbe con due persone al telefono: il ricevente potrebbe dire "OK" dopo ogni messaggio che sente chiaramente, mentre dirà "puoi ripetere?" se non ha sentito chiaramente. Questo protocollo "vocale" utilizza notifiche (acknwoledgment) positive e negative che ci torneranno utili nello sviluppo nel nostro protocollo RDT 2.0.

I protocolli di trasferimento dati affidabile basati su ritrasmissioni sono noti come protocolli ARQ (automatic repeat request).

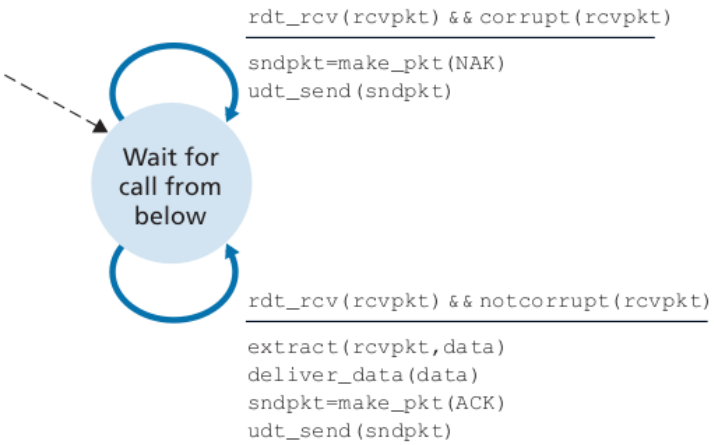
I protocolli ARQ devono avere tre funzionalità aggiuntive:

- Rilevamento dell'errore
- Feedback del destinatario: il destinatario deve dare un feedback esplicito al mittente che può essere positivo (ACK) o negativo (NAK)
- Ritrasmissione: Un pacchetto ricevuto con errori sarà ritrasmesso dal mittente quando quest'ultimo riceve un NAK

Ecco le FSM di RDT 2.0:




a. rdt2.0: sending side




b. rdt2.0: receiving side

RDT 2.0

Quando il lato mittente riceve `rdt_send(data)` costruisce un pacchetto come avviene per RDT 1.0 ma con la differenza che stavolta vi aggiunge un checksum e invia tale pacchetto tramite `udt_send(sndpkt)` dopodiché si mette in attesa di risposta di un ACK o di un NAK. Se riceve NAK rinvia il pacchetto e rimane in attesa, altrimenti ritorna nello stato iniziale.

Il destinatario, quando riceve il pacchetto controlla la correttezza del pacchetto. Se sono presenti errori crea un pacchetto NAK e lo invia al mittente, altrimenti gli manda un ACK e trasferisce i dati al livello superiore.

Si noti come quando il mittente è in attesa di una risposta dal destinatario non può ricevere una chiamata `rdt_send` e conseguentemente non può inviare un nuovo pacchetto. Protocolli di questo genere vengono chiamati protocolli stop-and-wait.

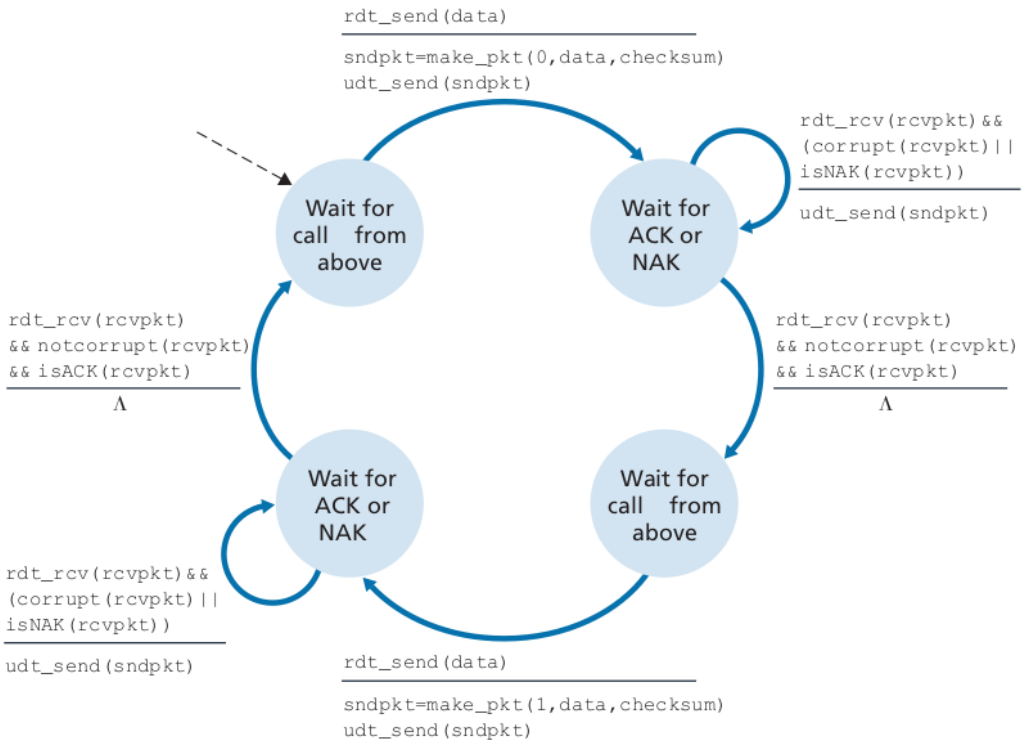
Il protocollo RDT 2.0 presenta però un grave problema, infatti, non si è tenuto conto della possibilità che i pacchetti ACK o NAK possano a loro volta essere alterati. Prendiamo allora in esame tre possibilità per gestire i pacchetti di acknowledgment corrotti:

1. Continuando con la nostra analogia telefonica, se chi detta non comprende il messaggio "OK" o il messaggio "puoi ripetere?" da parte del destinatario, probabilmente risponderà con "che cosa hai detto?". Ma se anche quest'ultimo venisse corrotto allora il destinatario potrebbe rispondere con "che cosa hai detto TU?" che a sua volta potrebbe essere corrotto. Tale approccio risulterebbe complicato

2. Potremmo aggiungere bit di checksum ai messaggi di acknowledgment che non solo rilevano, ma correggono anche gli errori. Questo però risolverebbe il problema di un ACK o NAK corrotto ma non perso

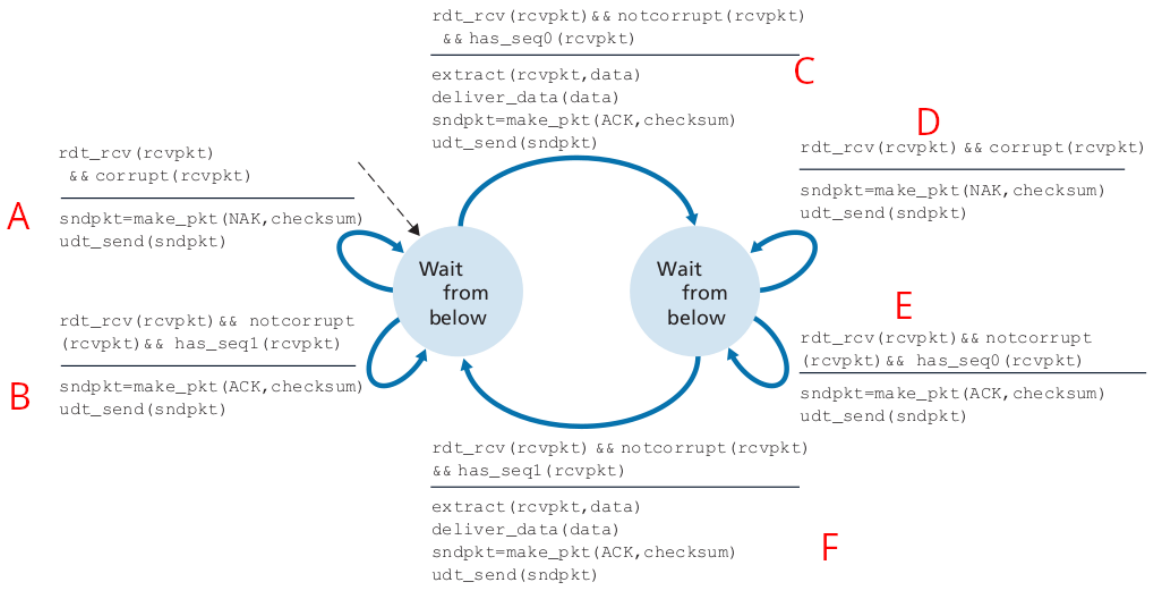
3. Un terzo approccio prevede che il mittente riinvi il pacchetto dati corrente a seguito di un pacchetto ACK o NAK alterato. Questo approccio potrebbe introdurre pacchetti nella rete duplicati in quanto il destinatario potrebbe aver mandato semplicemente un ACK e quindi con tale approccio riceverebbe due pacchetti uguali di seguito, ma non può sapere se il secondo è un nuovo pacchetto o se si tratti di una ritrasmissione. Per ovviare a ciò si può numerare ogni pacchetto con un numero di sequenza.

Creiamo quindi RDT 2.1: al destinatario sarà sufficiente controllare il numero di sequenza per capire se il pacchetto ricevuto rappresenti o meno una ritrasmissione. Nel primo caso il numero di sequenza del pacchetto avrà lo stesso valore di uno appena ricevuto, nel secondo caso sarà invece diverso. Per questo semplice protocollo stop-and-wait ci basta un solo bit per conservare il numero di sequenza. Dato che stiamo ipotizzando che la rete non perda pacchetti, il mittente saprà sempre che un ACK o NAK (alterato oppure no) farà sempre riferimento all'ultimo pacchetto inviato




RDT 2.1 Mittente

Questa FSM è molto simile a RDT 2.0 ma ha il doppio degli stati per gestire il numero di sequenza 0 e 1, che sono l'un l'altro speculari. L'unica cosa che cambia, infatti, è `make_pkt` Dopo ogni invio di pacchetto sulla rete, reinvia lo stesso pacchetto se riceve NAK o un ACK corrotto (e lui non sa che è un ACK).




RDT 2.1 Destinatario

- (A) Pacchetto corrotto; invia NAK
- (B) Pacchetto duplicato (has_seq sarebbe dovuto essere 0); invia di nuovo l' ACK
- (C) Pacchetto numero 0 ricevuto con successo, manda ACK e manda i dati al protocollo superiore
- (D) Pacchetto corrotto; invia NAK
- (E) Pacchetto duplicato (has_seq sarebbe dovuto essere 1); invia di nuovo l' ACK
- (F) Pacchetto numero 1 ricevuto con successo, manda ACK e manda i dati al protocollo superiore

### Trasferimento dati affidabile su un canale con perdite ed errori sui bit: rdt 3.0

Supponiamo ora che il canale di trasmissione, oltre a danneggiare i pacchetti, possa anche perderli.

Il protocollo deve preoccuparsi ora di due aspetti aggiuntivi: come rilevare lo smarrimento di pacchetti e cosa fare quando ciò avviene. Per gestire il primo problema assegneremo al mittente l'onere di rilevare e risolvere la perdita di pacchetti. Supponiamo che il mittente spedisca un pacchetto dati e che questo oppure l'acknowledgment di ritorno vada smarrito. In entrambi i casi il mittente non otterrà alcuna risposta da parte del destinatario. Se il mittente è disposto ad attendere un tempo sufficiente per essere certo dello smarrimento del pacchetto, può semplicemente ritrasmettere il pacchetto dati. Notiamo che il mittente potrebbe ritrasmettere un pacchetto perché esso o l'ACK di ritorno ha sperimentato un ritardo notevolmente lungo ma non è andato perso. Ciò permette che dei pacchetti vengano duplicati. Dunque il mittente non sa se un pacchetto dati sia andato perduto, se sia stato smarrito un ACK o se uno dei due ha semplicemente subito un ritardo. In tutti questi casi l'azione intrapresa è sempre la stessa: ritrasmettere.

Questo meccanismo richiede di introdurre quindi un timer in grado di segnalare al mittente l'avvenuta scadenza di un dato lasso di tempo. Il mittente dovrà quindi essere in grado di (1) inizializzare il contatore ogni volta che invia un pacchetto (sia che si tratti di un primo invio sia che si tratti di una ritrasmissione) (2) di rispondere a un interrupt generato dal timer con l'azione appropriata e (3) di fermare il contatore.

### 3.4.2 Protocolli per il trasferimento dati affidabile con pipeline

RDT 3.0 funziona ma si tratta di un protocollo stop-and-wait e pertanto molto lento.

Per valutarne la velocità supponiamo l'esistenza di due host il cui RTT di andata e ritorno sia 30 millisecondi. Supponiamo, inoltre, che il canale ha un tasso trasmissivo di 1Gbps e che si vogliano trasferire 8000bit.

Il tempo per trasmettere i bit sul collegamento è

$$\frac{L}{R} = \frac{8000bit}{10^9 bit/s} = 8\ microsecondi$$

L'ultimo bit entra sul canale a tempo 8 micros ed effettua un viaggio di 15ms verso il destinatario. Avremo quindi che il destinatario riceverà l'ultimo bit a tempo

$$t = \frac{RTT}{2} + \frac{L}{R} = 15.008ms$$

Per semplicità supponiamo che il tempo di trasmissione (L/R) di un pacchetto ACK sia trascurabile. L'ACK giunge al mittente all'istante

$$t = \frac{RTT}{2} + \frac{L}{R} + \frac{RTT}{2} = 30.008ms$$

Quindi in un arco di 30.008ms il mittente ha trasmesso per soli 0.008ms. Definiamo l'utilizzo del mittente come il seguente rapporto:

$$\frac{0.008}{30.008} = 0.00027$$

Quindi il mittente è stato attivo per un solo 0,27% del tempo. Visto in altro modo, il mittente è stato in grado di spedire solo 1000byte in 30.008s con un throughput effettivo di soli 267kbps sebbene il canale mettesse a disposizione 1Gbps. Inoltre abbiamo trascurato i tempi di elaborazione e di accodamento sui router intermedi che avrebbero ridotto ancora di più le prestazioni.

Per tale motivo anziché operare in modalità stop and wait, si consente al mittente di inviare più pacchetti senza attendere gli acknowledgment. Questa tecnica è nota come pipelining.

Le conseguenze su un protocollo di trasporto sono le seguenti:

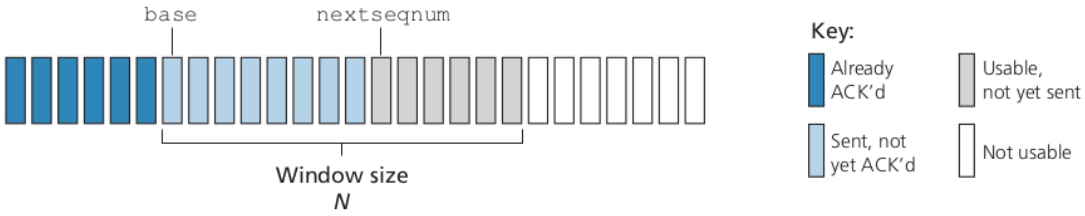
- L'intervallo dei numeri di sequenza deve essere incrementato in quanto ci possono essere più pacchetti in transito
- Il mittente dovrà avere un buffer in cui memorizza i pacchetti trasmessi ma di cui non ha ancora avuto esito
- Si possono identificare due approcci di base: Go-Back-N e ripetizione selettiva

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

In un protocollo GBN il mittente può trasmettere più pacchetti senza dover attendere alcun acknowledgment ma non può avere più di N pacchetti in attesa di acknowledgment. N viene chiamato ampiezza della finestra.

Se definiamo `base` come il numero di sequenza del pacchetto più vecchio che non ha ancora ricevuto esito e `nextseqnum` il numero di sequenza del prossimo pacchetto da inviare, possiamo identificare quattro intervalli:

1. I pacchetti con numero di sequenza `[0, base-1]` sono i pacchetti già trasmessi e che hanno ricevuto acknowledgment.
2. I pacchetti con numero di sequenza `[base, nextseqnum-1]` sono i pacchetti inviati ma che non hanno ricevuto ACK
3. I numeri di sequenza nell'intervallo `[nextseqnum, base+N-1]` possono essere utilizzati per i pacchetti da inviare immediatamente
4. I numeri di sequenza con un numero superiore a `base+N-1` non possono essere utilizzati fintantoché il mittente non riceva uno o più ACK dei pacchetti nella pipeline ancora privi di riscontro.




Visione del mittente sulla pipeline GBN

Il protocollo GBN viene detto protocollo a finestra scorrevole.

Il numero di sequenza è scritto in un campo di $k$ bit e dunque sono possibili $2^k$ numeri di sequenza (da 0 a $2^k - 1$). Tutte le operazioni aritmetiche che coinvolgono il numero di sequenza dovanno essere in modulo $2^k$ (ovvero dopo il valore $2^k - 1$ viene 0).

Il mittente GBN deve rispondere a tre tipi di evento:

1. Invocazione dall'alto: quando dall'alto si chiama `rdt_send()` come prima cosa controlla se la finestra sia piena, ossia se vi siano N pacchetti in sospeso senza acknowledgment. Se la finestra non è piena, invia il pacchetto e fa partire il timer.

2. Ricezione di un ACK: nel protocollo GBN l'ACK del pacchetto con numero di sequenza n verrà considerato come un acknlowledgment cumulativo che indica che tutti i pacchetti con numero di sequenza minore o uguale a n sono stati correttamente ricevuti. A questo punto si controlla se vi siano ancora pacchetti in attesa di riscontro (e quindi `base+1` $\neq$ `nextsum`). Se così fosse, il timer viene fatto ripartire.

3. Quando si verifica un timeout il mittente invia nuovamente tutti i pacchetti trasmessi e non riscontrati e successivamente fa ripartire il timer.

Le azioni del destinatario GBN sono semplici: Se un pacchetto con numero di sequenza n viene correttamente ricevuto ed è in ordine, il destinatario manda un ACK per quel pacchetto. In tutti gli altri casi il destinatario scarta i pacchetti e rimanda un ACK per il pacchetto corretto ed in ordine ricevuto più di recente. Per induzione se un pacchetto $k$ è stato ricevuto in ordine e corretto, tutti i pacchetti con numero inferiore a $k$ sono stati anch'essi ricevuti correttamente ed in ordine.

Si noti che un pacchetto senza errori ma non in ordine viene scartato e ciò può sembrare un'operazione sciocca da fare, ma vi è una buona ragione: supponiamo che sia atteso il pacchetto $n$ ma che arrivi il pacchetto $n+1$. Il destinatario potrebbe momentaneamente mettere in un buffer $n+1$ e consegnarlo al livello superiore solo dopo aver ricevuto $n$. Ma il mittente, non avendo avuto riscontro su $n$, stando alle regole di GBN, invia sia $n$ sia $n+1$ quindi il destinatario può semplicemente scartare quest'ultimo.

L'unica informazione che il destinatario deve memorizzare è il numero di sequenza del pacchetto che si aspetta.

### 3.4.4 Ripetizione selettiva (SR)

Esistono scenari in cui GBN ha problemi di prestazioni ad esempio quando nella pipeline si trovano numerosi pacchetti e un errore su uno può provocare un elevato numero di ritrasmissioni, molte delle quali non necessarie che possono saturare la banda. I protocolli a ritrasmissione selettiva evitano le ritrasmissioni non necessarie facendo ritrasmettere al mittente solo quei pacchetti su cui esistono sospetti di errore (smarriti o alterati).

Si userà di nuovo una finestra di dimensione $N$ ma stavolta si consentirà a dei pacchetti all'interno della finestra di essere marcati come ACK'd. Inoltre, anche il destinatario avrà una sua finestra della stessa dimensione.

Il mittente ha un timer per ogni pacchetto che invia, ritrasmettendolo qualora non riceva un ACK nel tempo stabilito. La sua finestra si muove in avanti verso il primo pacchetto non ancora ACK'd o non ancora usato quando riceve un ACK per `base`

Il destinatario invia un ACK per i pacchetti ricevuti sia in ordine sia fuori sequenza (che memorizzerà in un buffer fin quando non arrivino i pacchetti con numero di sequenza minore).

Se il destinatario si aspetta il pacchetto $n$ e arriva correttamente il pacchetto $n$, la finestra si muove in avanti di 1. Se il destinatario si aspetta il pacchetto $n$ e arriva $n+t$, memorizza $n+t$ nel buffer ma non sposta avanti la finestra. La sposterà solo quando arrivano tutti gli altri pacchetti da $n$ a $n+t-1$ (`base` assumerà il valore di $n+t+1$).

Le finestre del mittente e del destinatario potrebbero non coincidere, caso in cui il destinatario invia un ACK e quindi la sua finestra si sposta, ma non arriva al mittente. È pertanto fondamentale che $N$ sia minore o uguale alla metà dei numeri possibili di sequenza in quanto se ciò non fosse il caso verrebbero immessi sulla rete pacchetti distinti ma con identico numero di sequenza, causando errori ed incomprensioni.

Nei documenti digitali allegati vi è una cartella denominata "SelectiveRepeatProtocol" nel quale è presente una simulazione interattiva.

## 3.5 Trasporto orientato alla connessione: TCP

TCP si basa su molti dei principi precedentemente trattati.

### 3.5.1 Connessione TCP

TCP viene detto orientato alla connessione in quanto prima di effettuare lo scambio di dati si effettuta un handshake, ossia i processi sugli host devono inviarsi reciprocamente alcuni segmenti preliminari per stabilire i parametri del successivo trasferimento dati.

Dato che il protocollo TCP è presente solo sui sistemi periferici, i router intermedi sono completamente ignari delle connessioni TCP: essi vedono datagrammi, non connessioni.

TCP offre un servizio full duplex: i dati a livello applicazione possono fluire dal processo A al processo B nello stesso momento in cui fluiscono nella direzione opposta. Una connessione TCP, inoltre, è anche punto-a-punto ovvero ha luogo tra un singolo mittente e un singolo destinatario. Il multicast non è possibile.

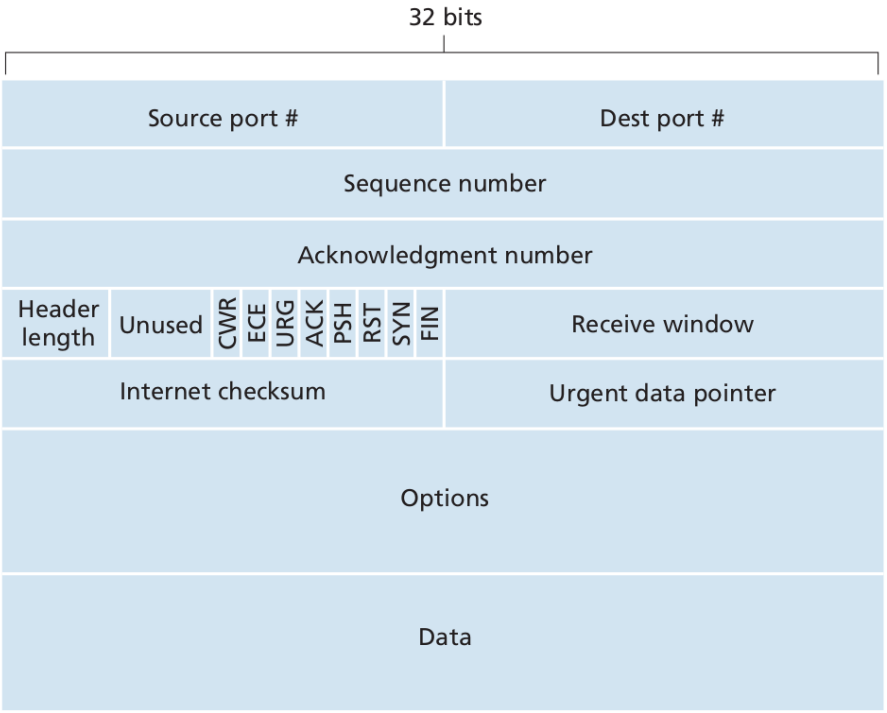
Vediamo come si instaura questo handshake: il client invia per primo uno speciale segmento TCP, il server risponde con un secondo segmento speciale TCP, infine il client risponde con un terzo segmento speciale. I primi due non trasportano payload mentre al terzo è consentito. Dato che gli host si scambiano tre segmenti, questa procedura è nota come handhsking a tre vie.

Una volta instaurata una connessione TCP, i due processi possono scambiarsi dati. Il primo manda un flusso dati verso la socket, la quale, quando riceve i dati, li dirige al buffer di invio allocato durante l'handshaking da cui preleverà blocchi dati e li passerà al livello rete.

Anche lato ricevente c'è un buffer che viene utilizzato per leggere i dati in ricezione.

La dimensione massima di segmento (MSS) è la massima quantità di dati a livello di applicazione all'interno del segmento (intestazioni TCP escluse). In altre parole è la dimensione massima di un payload TCP. L'MSS viene impostato determinado la lunghezza massima del frame più grande che può essere inviato a livello di collegamento, nota come Unità Trasmissiva Massima (MTU). Si sceglie una MSS tale per cui MSS + intestazioni TCP + intestazioni IP sia minore o uguale a MTU. Si sceglie quindi che un segmento TCP stia tutto in un datagramma IP e che un datagramma IP stia tutto in un singolo frame a livello di collegamento per questioni di efficienza e affidabilità. Quando il livello applicazione manda dati più grandi di MSS, TCP suddivide i pacchetti in modo che rispettino la dimensione massima.

### 3.5.2 Struttura dei segmenti TCP




Struttura di un segmento TCP

Il segmento TCP consiste di campi intestazione (header) e di un campo contenente un blocco di dati proveniente dal livello applicazione (payload) di dimensione massima MSS.

L'intestazione contiene i numeri di porta di origine e destinazione, il campo checksum, il numero di sequenza e il numero di acknowledgment. Abbiamo inoltre il campo finestra di ricezione per il controllo del flusso che indica il numero di byte che il destinatario è disposto ad accettare, le opzioni che sono facoltative, il campo lunghezza dell'intestazione che indica la dimensione dell'header che al variare del numero di opzioni varia la dimensione e 6 campi flag.

#### Numeri di sequenza e numeri di acknowledgment

TCP vede i dati a livello applicazione come un flusso di byte ordinati ma non strutturati.

Il numero di sequenza contenuto in un pacchetto è il numero del primo byte contenuto in esso. Facciamo un esempio: ipotizziamo ci sia un flusso da 500.000 byte e che MSS sia 1000 byte e che il primo byte sia numerato con zero. Allora TCP costruirà 500 segmenti i cui numeri di sequenza saranno 0, 1000, 2000, ... In realtà i numeri di sequenza non partono sempre da zero, infatti, mittente e destinatario usano un numero casuale come numero iniziale per evitare che segmenti attinenti a vecchie connessioni TCP tra gli stessi host ancora in rete possano essere interpretati come segmenti validi e pertanto mandare in confusione la trasmissione.

Il numero di acknowledgment che l'host A scrive nei propri segmenti è il numero del byte successivo che attende di ricevere dall'host B

- Se B manda ad A un segmento contenente i byte numerati da 0 a 999, il prossimo segmento di A avrà un numero di acknowledgment di 1000
- Se B banda ad A 3 segmenti: [0-999], [1000-1999], [2000-2999] ma A riceve solo il primo e l'ultimo, allora A metterà sempre 1000 nei suoi ACK fin quando non arriva [1000-1999].

Dato che TCP effettua l'acknowledgment solo dei byte fino al primo byte mancante nel flusso, si dice che tale protocollo offra acknowledgment cumulativi. Per fare un esempio, inviare ACK 5000 indica che i byte 0-4999 sono arrivati correttamente.

C'è una coppia (numero di sequenza, numero di acknowledgment) per il mittente e una coppia (numero di sequenza, numero di acknowledgment) per il destinatario.

Che cosa fa un host quando riceve segmenti fuori sequenza? RFC non impone nulla a riguardo pertanto chi implementa il protocollo TCP ha due strade:

1. Il destinatario scarta immediatamente i segmenti non ordinati
2. Il destinatario mantiene i byte non ordinati in un buffer e attende qualli mancanti per colmare i vuoti.

#### Telnet: un caso studio

Telnet è un protocollo a livello applicazione impiegato per il login remoto TCP ed è un'applicazione interattiva.

Ogni carattere immesso dall'utente client viene spedito all'host remoto il quale lo rimanda indietro e che solo alla sua ricezione sul client verrà mostrato sul monitor del mittente. Sembrerebbe un'operazione inutile ma questo "eco" permette di assicurarsi che i caratteri visibili all'utente siano stati ricevuti e processati dall'host remoto. Pertanto nel lasso di tempo che intercorre tra la digitazione di un carattere e la sua visualizzazione sul monitor, il carattere ha viaggiato nella rete due volte.

Ipotizziamo una comunicazione telnet tra Host A e Host B: quest'ultimi si acordano per usare, rispettivamente, numero di sequenza iniziale 42 e 79. Ecco cosa avviene:

1. Il mittente manda la tripla `(Seq=42, Ack=79, data='C')` che significa "il primo byte in questo messaggio ha numero di sequenza 42 e mi aspetto da te un segmento con numero di sequenza 79. Invio il carattere 'C'.
2. Il destinatario risponde con la tripla `(Seq=79, Ack=43, data='C')`. Questo segmento ha un duplice obiettivo: confermare l'avvenuta ricezione del primo segmento (quindi si tratta di un acknowledgment) e trasportare dati al client. Per questo motivo si dice che l'acknowledgment è "piggybacked"
3. Il mittente conferma l'avvenuta ricezione del segmento con la coppia `(Seq=43, Ack=80)`. Qui anche se l'acknowledgment non è piggybacked (ovvero non trasporta dati), Seq va avanti di uno per indicare al server che questo è un segmento nuovo.

### 3.5.3 Timeout e stima del tempo di andata e ritorno

TCP utilizza un meccanismo di timeout per recuperare i segmenti persi. Chiaramente, il timeout dovrebbe essere più grande del tempo di andata e ritorno sulla connessione (RTT). Ma di quanto deve essere maggiore? E come stimare RTT?

#### Stima del tempo di andata e ritorno

L'RTT misurato per un segmento, denotato come `SampleRTT` è la quantità di tempo che intercorre tra l'istante in cui il segmento viene passato a IP e quello della ricezione del suo acknowledgment.

`SampleRTT` viene valutato per ogni segmento non ritrasmesso.

Per effettuare una stima dell'RTT medio si usa la seguente formula, calcolata per ogni `SampleRTT`:

$$\texttt{EstimatedRTT} = (1 - \alpha) \times \texttt{EstimatedRTT} + \alpha \times \texttt{SampleRTT}$$

dove $\alpha$ è una costante fissa il cui valore raccomandato è 0,125.

Tale media attribuisce maggiore importanza ai campioni recenti rispetto a quelli vecchi. Ciò è naturale in quanto ciò riflette la congestione attuale della rete. Tale media viene definita media mobile esponenziale ponderata.

Oltre ad avere una stima di RTT vorremmo avere anche una misura della sua variabilità che denoteremo come `DevRTT` come una stima di quanto `SampleRTT` generalmente si discosta da `EstimatedRTT`. La formula è la seguente:

$$\texttt{DevRTT} = (1 - \beta) \times \texttt{DevRTT} + \beta \times |\texttt{SampleRTT} - \texttt{EstimatedRTT}|$$

che è simile alla formula precedente: con $|\texttt{SampleRTT} - \texttt{EstimatedRTT}|$ vede quanto si discosta `SampleRTT` da `EstimatedRTT` e vi applica un fattore $\beta$ generalmente di 0,25.

#### Impostazione e gestione del timeout di ritrasmissione

Dati i valori di `EstimatedRTT` e `DevRTT`, quale valore dovremmo usare per il timeout di TCP? Non può chiaramente essere minore di `EstimatedRTT` altrimenti si verrebbero a creare trasmissioni non necessarie nè tantomento dovrebbe essere molto maggiore di tale valore altrimenti TCP non ritrasmetterebbe rapidamente un segmento perduto.

Si dovrebbe pertanto impostare il timeout a `EstimatedRTT` più un certo margine che dovrebbe essere grande quando c'è molta variabilità di `EstimatedRTT`, piccolo altrimenti. La formula che si usa è:

$$\texttt{TimeoutInterval} = \texttt{ExtimatedRTT} + 4 \times \texttt{DevRTT}$$

Il valore iniziale di `TimeoutInterval` è pari a 1.

### 3.5.4 Trasferimento dati affidabile

Il livello rete non è affidabile: non garantisce la consegna, né la consegna in sequenza e né l'integrità dei dati.

TCP crea un servizio di trasporto dati affidabile al di sopra del servizio inaffidabile del livello di rete, assicurando che il flusso di byte che il processo ricevente leggerà dal buffer è esattamente quello spedito, ovvero senza alterazioni, buchi, duplicazioni e in sequenza.

Con i precedenti algoritmi teorici assegnavamo un timer per ogni segmento trasmesso per cui non si sia ricevuto ancora un acknowledgment ma TCP utilizza un solo timer di ritrasmissione, anche in presenza di più segmenti senza acknowledgment. Lo si può pensare come un timer associato al segmento più vecchio che non ha ancora ricevuto acknowledgment.

Esistono tre eventi che un mittente TCP deve gestire:

1. Dati provenienti dall'applicazione: TCP li incapsula in uno o più segmenti che vengono successivamente passati a IP

2. Timeout: quando si verifica un timeout, TCP risponde con il segmento che lo ha causato e successivamente resetta il timer. In altre parole ritrasmette il segmento con il più basso numero di sequenza di cui non si è ancora ricevuto l'ACK

3. Arrivo del segmento di acknowledgment: TCP confronta il valore contenuto nell'ACK con la propria variabile `Base` che rappresenta il numero di sequenza del più vecchio byte che non ha ancora ricevuto un acknowledgment. Conseguentemente `Base - 1` è il numero di sequenza dell'ultimo byte che si sa essere stato ricevuto correttamente. TCP usa acknowledgment cumulativi: Pertanto un ACK `y` conferma la ricezione di tutti i byte precedenti al byte numero `y`. A questo punto se ci sono altri segmenti in attesa di acknowledgment riavvia il timer altrimenti lo ferma.

#### Raddoppio dell'intervallo di timeout

Ogni volta che scatta un timer, TCP imposta il successivo intervallo di timeout al doppio del tempo del precedente anziché derivarlo da `EstimatedRTT` e `DevRTT`. Il timer viene reimpostato al suo valore normale dopo uno degli altri due eventi posisbili ((1) e (3)).

Ciò consente di iniziare a creare un piccolo controllo della congestione in quanto ciascun mittente ritrasmette dopo intervalli sempre più lunghi, rispettando la rete.

#### Ritrasmissione rapida

Il mittente può in molti casi rilevare la perdita dei pacchetti ben prima che si verifichi l'evento di timeout grazie agli ACK duplicati. Dobbiamo innanzitutto capire perché esistono ACK duplicati.

Quando il destinatario TCP rileva un buco nel flusso dei dati (ovvero un segmento mancante), il destinatario non può inviare un NAK esplicito per questi segmenti al mittente in quanto TCP non prevede ciò. Dunque manda invece un ACK relativo all'ultimo byte che ha ricevuto correttamente e in ordine, duplicando così un ACK. Facendo un esempio, quando riceve `[0-999]` invia `ACK(1000)`, dopodiché riceve `[2000-2999]` al quale risponde ancora una volta con `ACK(1000)`.

Nel caso in cui siano giunti tre ACK duplicati (e quindi in totale quattro ACK identici), il mittente effettua una cosiddetta ritramissione rapida, rispendendo il segmento mancante prima della scadenza del timer.

Perché proprio tre? Perché se fossero uno o due, c'è ancora una probabilità sufficiente a far pensare che quel segmento abbia semplicemente subìto un ritardo, ma se ne ricevo tre vuol dire che oltre a quello smarrito ho inviato altri tre segmenti, dei quali ho anche ricevuto un ACK. Allora è molto probabile che il segmento (o l'ACK) sia stato perso.

Queste sono le azioni intraprese da TCP:

1. Arrivo di un segmento con numero di sequenza atteso. Tutti i segmenti fino a questo sono già stati riscontrati: attende 500ms per l'arrivo ordinato di un altro segmento. Se non arriva manda un ACK (ciò consente di avere un ACK cumulativo)

2. Arrivo di un segmento con numero di sequenza atteso. C'è un segmento ordinato in attesa di trasmissione dell'ACK (caso 1): Invia immediatamente un ACK cumulativo per entrambi. Caso 1 e caso 2 insieme fanno sì che possano esistere ACK cumulativi ma al massimo per due segmenti insieme.

3. Arrivo non ordinato di segmento con numero di sequenza superiore a quello atteso (viene rilevato un buco): Invia immediatamente un ACK duplicato relativo all'ultimo byte correttamente ricevuto.

4. Arrivo di segmento che colma parzialmente o completamente un buco: Invia un ACK per questo segmento solo se colma l'estremo sinistro di un buco, altrimenti invia un ACK duplicato relativo all'ultimo byte ricevuto in ordine.

#### GBN o SR?

TCP è un protocollo GBN o SR? Si presenta come un ibrido tra i due.

### 3.5.5 Controllo di flusso

Se l'applicazione ricevente è troppo lenta nella lettura dei dati dal buffer può accadere che il mittente mandi in overflow il buffer di ricezione del destinatario, inviando dati troppo rapidamente. Per questo motivo è necessario un servizio di controllo di flusso che fortunatamente è offerto da TCP. Gli algoritmi di controllo di flusso confrontano la velocità di invio del mittente con quella di lettura del destinatario.

Inoltre la rete Internet potrebbe essere congestionata (indipendentemente dalle condizioni del buffer del ricevente) e pertanto TCP mette a disposizione il controllo della congestione.

Benché il controllo di flusso e il controllo della congestione risultino simili, facendo in modo che il mittente rallenti, sono causate da ragioni diverse.

Trattiamo brevemente il controllo di flusso TCP, supponendo che il destinatario scarti sempre i segmenti arrivati non in ordine. L'algoritmo è implementato facendo mantenere al mittente una variabile chiamata finestra di ricezione che gli fornisce dettagli sullo spazio libero disponibile nel buffer del destinatario. Anche il destinatario per lo stesso motivo mantiene la sua finestra di ricezione.

Entrambi gli host allocheranno un buffer di ricezione di dimensione `RcvBuffer`. Inoltre avremo `LastByteRead` che è il numero dell'ultimo byte nel flusso dati che il processo applicativo ha letto dal buffer e `LastByteRcvd` rappresenta il numero dell'ultimo byte arrivato dalla rete salvato nel buffer. Avremo dunque che la seguente disuguaglianza deve essere sempre soddisfatta per evitare un overflow:

$$\texttt{LastByteRcvd} - \texttt{LastByteRead} \leq \texttt{RcvBuffer}$$

Facciamo un esempio:

Se l'ultimo byte che ha letto dal buffer è il 20° del flusso dati e l'ultimo byte che ha ricevuto e memorizzato nel buffer è il 30°, allora il buffer deve essere grande almeno 10 byte, altrimenti va in overflow.

La finestra di ricezione, denotata con `rwnd` rappresenta la quantità disponibile nel buffer:

$$\texttt{rwnd} = \texttt{RcvBuffer} - (\texttt{LastByteRcvd} - \texttt{LastByteRead})$$

Il valore di `rwnd` viene messo nel campo apposito del segmento inviato da un host all' altro.

Vengono memorizzate ulteriori variabili: `LastByteSent` che rappresenta il numero dell'ultimo byte del flusso inviato e `LastByteAcked` che rappresenta l'ultimo byte per cui si è ricevuto un acknowledgment. Ma allora $\texttt{LastByteSent} - \texttt{LastByteAcked}$ rappresenta quanti dati ci sono nel buffer dell'altro host. Per tale motivo chi invia dati deve mentenere la seguente disuguaglianza sempre vera:

$$\texttt{LastByteSent}_{\texttt{mittente}} - \texttt{LastByteAcked}_{\texttt{mittente}} \leq \texttt{rwnd}_{\texttt{destinatario}}$$

Vediamo che cosa succede quando `rwnd = 0` nell'host destinatario B e che non abbia ACK (∗) o altri dati da inviare al mittente A. Mano a mano il buffer di B si svuoterà ma poiché B non ha nulla da inviare ad A (e quindi non può far pervenire il nuovo valore di `rwnd`) e A non manda segmenti a B perché nota che la finestra è piena, ci troveremo in una situazione di stallo. Per ovviare a ciò le specifiche TCP impongono che sebbene `rwnd` sia zero, il mittente continui a mandare segmenti con un byte di dati. In questo caso se il buffer è pieno vengono semplicemente scartati, ma quando avrà iniziato a svuotarsi, manderà un ACK contenente il nuovo valore di `rwnd`.

(∗) Gli ACK vengono inviati quando si ricevono segmenti, non quando si leggono dal buffer. Quindi B continua a leggere dal suo buffer senza mandare ACK

### 3.5.6 Gestione della connessione TCP

Vediamo come viene stabilita e rilasciata una connessione TCP. Fare ciò può aggiungere ritardi.

Quando la parte client TCP vuole aprire una connessione con la corrispondente parte server, questi sono i passi intrapresi:

1. TCP lato client invia uno speciale segmento TCP al server che non contiene dati a livello applicativo, ma il bit SYN è posto a 1. Per tale motivo questo segmento viene chiamato segmento SYN. Inoltre il client sceglie a caso un numero di sequenza iniziale (`client_isn`) e lo pone nel campo numero di sequenza

2. Il server alloca il buffer e le variabili necessarie e invia un segmento di connessione approvata al client. Anche questo segmento ha il campo SYN impostato a 1 e non contiene dati a livello applicativo. Inoltre, sceglie il proprio numero di sequenza iniziale (`server_isn`) e imposta ACK al valore `client_isn+1`. Questo segmento viene chiamato SYNACK

3. Alla ricezione del segmento SYNACK, anche il client alloca variabili e buffer e invia al server un altro segmento che contiene nel campo ACK il valore `server_isn+1` e il valore 0 nel campo SYN (d'ora in poi assumerà sempre questo valore). Il campo dati del segmento può contenere dati del livello applicazione (piggybacking).

Una volta completati questi tre passi, client e server possono scambiarsi segmenti contenenti dati effettivi. Dato che questo handshake richiede tre passi, viene detto handshake a tre vie.

Ciascuno dei due processi che partecipa alla connessione può terminarla. In tal caso le risorse sugli host vengono deallocate. Per far sì che la connessione termini, uno dei due host (diciamo host A) invia un segmento speciale con il bit FIN impostato a 1 a host B. Quando B riceve il segmento, invia un segmento ACK, immediatamente seguito da un ulteriore segmento con bit FIN a 1. Infine A manda a sua volta un ACK e la connessione termina.

Nell'arco di una connessione TCP, i protocolli attraversano vari stati: il client TCP parte dallo stato "closed" quando non vi è alcuna connessione TCP in esecuzione per poi passare allo stato "SYN sent" una volta spedito il segmento SYN. Ricevuto l'ACK dal server, lo stato del protocollo client passa a "estabilished", stato che permette di inviare e ricevere dati utili. Supponendo che l'applicazione client decida di voler chiudere la connessione, esso spedisce il segmento FIN al server e passa allo stato "FIN wait 1" che diventa "FIN wait 2" quando riceve l'ACK dal server. Quando riceve FIN e invia l'ultimo ACK passa nello stato "Time wait" in cui vi si inserisce per qualche secondo prima di ritornare allo stato "closed" nel quale dealloca tutte le risorse. Lo stato "Time wait" serve affinché il client abbia abbastanza tempo per ritentare l'invio dell'ultimo ACK nel caso vada perduto.

Se un host riceve un segmento TCP la cui porta di destinazione non corrisponde ad alcuna socket in ascolto, quest'host invierà al mittente un segmento speciale con il bit RST (reset) impostato a 1 che comunica alla controparte di non rispedire lo stesso segmento. Nel caso di UDP, invece, il destinatario manda uno speciale datagramma ICMP.

## 3.6 Princìpi del controllo di congestione

La ritrasmissione dei pacchetti tratta il sintomo della congestione della rete (la perdita di quest'ultimi causati da un buffer overflow nei router) ma non la causa (il tentativo di una o più sorgenti di inviare dati a ritmi troppo elevati). Per questo motivo sono richiesti meccanismi per adeguare l'attività dei mittenti in relazione al traffico.

### 3.6.1 Cause e costi della congestione

Il libro fa una serie di esempi che mostrano alcune cause che incrementano la congestione della rete:

- Avere un throughput vicino alla velocità massima del collegamento di uscita del router potrebbe sembrare ideale dal punto di vista del throughput ma non dal punto di vista del ritardo. Infatti più ci si avvicina al limite del collegamento più saranno lunghi i ritardi di accodamento e quindi la congestione aumenta.
- Il mittente deve effettuare delle ritrasmissioni quando i pacchetti vengono scartati. Questa è sia una causa della congestione della rete (più pacchetti immessi = più congestione), ma anche una conseguenza (più la rete è congestionata più è probabile che alcuni pacchetti vengono persi). Inoltre se il timer del mittente scatta prematuramente, immette nella rete un pacchetto in più che non era necessario inviare. Ciò causa ulteriore congestione.
- Si ipotizzi un pacchetto che attraversa N router che viene scartato dall'N-esimo a causa di un buffer pieno. Allora il lavoro svolto, nonché la banda utilizzata, dei precedenti N-1 router è stata sprecata e conseguentemente aiuta ad aumentare la congestione della rete (vengono elaborati e inviati pacchetti che verranno successivamente scartati)

### 3.6.2 Approcci al controllo della congestione

Esistono due approcci per controllare la congestione:

- Controllo di congestione end-to-end: se i router intermedi non forniscono supporto esplicito al controllo della congestione, quest'ultimo deve essere gestito sui sistemi periferici, osservando il comportamento della rete. Questo è proprio il caso della rete Internet, di cui TCP si fa carico (in quanto il livello rete non offre tale supporto)
- Controllo di congestione assistito dalla rete: I router forniscono un feedback esplicito al mittente sullo stato della congestione della rete.

## 3.7 Controllo di congestione TCP

TCP offre il meccanismo di controllo della congestione, imponendo a ciascun mittente un limite alla velocità di invio sulla propria connessione in funzione della congestione di rete percepita.

Per fare questo, TCP fa tener traccia agli estremi della connessione una variabile aggiuntiva chiamata finestra di congestione, indicata con `cwnd`. Ricordandoci della variabile `rwnd` precedentemente menzionata, si avrà che TCP mittente imporrà che la seguente diseguaglianza risulti sempre soddisfatta:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq min\{\texttt{cwnd}, \texttt{rwnd}\}$$

ricordandosi che `LastByteSent` - `LastByteAcked` equivale al numero di byte inviati ma di cui non si è ricevuto alcun acknowledgment.

Assumiamo che `rwnd` sia sufficientemente grande da essere ignorata e assumiamo che il mittente abbia sempre dati da inviare. Notiamo dunque che possono essere inviati al massimo `cwnd` byte di cui certamente non si può ricevere l'ACK in meno di RTT secondi. Ciò implica che il massimo della velocità trasmissiva è $\frac{\texttt{cwnd}}{RTT}$ byte/s. Ciò dimostra che modificando il valore di `cwnd` il mittente può regolare la velocità di invio sulla propria connessione.

La congestione viene rilevata tramite un "evento di perdita", ovverosia l'occorrenza o di un timeout o di una ricezione di tre ACK duplicati (e quindi quattro ACK identici). Queste due condizioni sono diretta conseguenza di un congestionamento di rete perché sia il ritardo che la perdita di un segmento (quest'ultimo dovuto a un overflow del buffer di qualche router intermedio) sono dovuti a un congestionamento. TCP risponde a tale evento con una riduzione dell'ampiezza della finestra di congestione.

D'altro canto quando un mittente riceve degli ACK non duplicati, interpreta ciò come un'indicazione che tutto vada bene e che la rete non è congestionata. TCP risponde aumentando l'ampiezza della finestra di congestione.

La strategia di TCP è quello di incrementare la velocità di trasmissione in risposta all'arrivo di ACK finché non si verifica un evento di perdita che causa una riduzione della velocità di trasmissione. A questo punto la velocità di trasmissione viene nuovamente incrementata per rilevare a che tasso trasmissivo iniziano a verificarsi perdite. Cerca pertanto un punto di equilibrio tra la domanda e l'offerta della rete.

Per il comportamento di TCP sopracitato, si dice che TCP è auto-temporizzato.

L'algoritmo di congestione di TCP presenta tre componenti: (1) slow start, (2) congestion avoidance e (3) fast recovery

#### Slow start

Quando si stabilisce una connessione TCP, il valore di `cwnd` viene impostato a 1 MSS e si incrementa di 1 MSS ogniqualvolta un segmento trasmesso riceve un ACK. Questo processo ha come effetto una velocità di trasmissione iniziale lenta per poi crescere esponenzialmente.

Cresce esponenzialmente perché dopo la ricezione di un ACK del primo segmento trasmesso potrà trasmetterne due, dopodiché potrà trasmetterne 4,8,16 e così via.

Adesso vediamo cosa succede quando c'è un evento di perdita:

- Se l'evento di perdita è stato causato da un timeout, TCP imposta la variabile `ssthresh` (slow start threshold) a `cwnd`/2 (metà del valore di quando è stata rilevata la congestione) e pone il valore di `cwnd` di nuovo pari a 1 MSS e il processo inizia da capo. Quando `cwnd` raggiunge il valore `ssthresh`, anziché raddoppiare di nuovo (e quindi raggiungere di nuovo il valore in cui si era verificato un timeout), la fase di slow start termina e TCP va in modalità congestion avoidance.

- Se vengono rilevati tre ACK uguali, TCP effettua una ritrasmissione rapida (subsection 3.5.4) ed entra nella modalità fast recovery.

#### Congestion avoidance

Quando TCP entra in modalità congestion avoidance, il valore di `cwnd` è la metà di quello che aveva l'ultima volta in cui è stata rilevata la congestione, quindi invece di raddoppiare `cwnd` a ogni RTT, lo incrementa di 1 MSS a ogni RTT, assumendo un andamento lineare anziché esponenziale.

Congestion avoidance termina nei seguenti casi:

- Se l'evento di perdita è stato causato da un timeout, si comporta allo stesso modo di Slow Start

- In caso di tre ACK duplicati, la rete continua a consegnare pacchetti, pertanto la risposta di TCP è meno drastica: imposta il valore `ssthresh` a `cwnd`/2 e successivamente dimezza il valore di `cwnd` per poi entrare nello stato di fast recovery.

#### Fast recovery

Durante la fase di fast recovery il valore di `cwnd` è incrementato di 1 MSS per ogni ACK duplicato ricevuto relativamente al segmento perso che ha causato l'entrata di TCP in modalità fast recovery.

- Quando arriva un ACK per il segmento perso, TCP entra nello stato di congestion avoidance dopo aver ridotto il valore di `cwnd`

- Se si verifica un timeout vi è invece una transizione dallo stato di fast recovery a quello di slow start, intraprendendo le stesse azioni di Congestion avoidance e Slow Start: `ssthresh` è posto a metà di `cwnd` prima che quest'ultimo venga impostato a 1 MSS

L'algoritmo di congestione TCP svolge la funzione di un algoritmo distribuito di ottimizzazione asincrona.

#### Descrizione macroscopica del throughput TCP

Ignoriamo le fasi di slow start che generalmente durano poco in quanto il mittente aumenta esponenzialmente le sue trasmissioni.

Sia $W$ il numero di byte quando si verifica un evento di perdita. Allora il throughput massimo è $W/RTT$ e quello minimo è $\frac{W}{2}/RTT = W/2RTT$ e quindi il throughput medio è

$$0,75 \times \frac{W}{R}$$

### 3.7.1 Fairness

Consideriamo $K$ connessioni TCP attraverso un collegamento con capacità trasmissiva di $R$ bps. Si dice che un meccanismo di controllo della congestione sia fair se la velocità trasmissiva media di ciascuna connessione sia $R/K$ bps. In altre parole, ciascuna connessione ottiene la stessa porzione di banda del collegamento.

TCP tende a offrire ciò (si può dimostrare), pertanto è fair.

#### Fairness e UDP

La fonia e la videoconferenza non fanno uso di TCP: anche a costo di perdere qualche pacchetto, preferiscono che la loro banda non venga limitata anche se la rete è molto congestionata.

UDP pertanto non è fair e non coopera con i meccanismi anti-congestionamento di TCP. Ciò implica che il traffico UDP può soffocare il traffico TCP.

#### Fairness e connessioni TCP parallele

Benché TCP sia fair, nulla impedisce a un processo di aprire più connessioni TCP parallele, di fatto aumentando il throughput medio dello stesso durante un congestionamento.

## Notifica esplicita di congestione: controllo di congestione assistito dalla rete

Un mittente TCP non riceve alcuna notifica esplicita di congestione dal livello di rete ma ne deduce l'esistenza osservando la perdita di pacchetti. Di recente sono state proposte estensioni per IP e TCP che permettono al livello di rete di segnalare esplicitamente una congestione a mittente e ricevente TCP. Questa forma di controllo assistito dalla rete è nota come notifica esplicita di congestione.