Politecnico di Bari

Facoltà di Ingegneria Elettronica

Corso di *Reti di Telecomunicazioni*

Protocolli di trasporto della famiglia TCP/IP

Introduzione

Il *livello transport* delle reti IP è basato fondamentalmente su due protocolli:

- **TCP** (*Transmission Control Protocol*), *orientato alla connessione* (RFC 793, 1122 e 1323);
- **UDP** (*User Data Protocol*), *non orientato alla connessione* (RFC 768).

Il secondo è di fatto IP con l'aggiunta di un breve header e fornisce un servizio di trasporto datagram (non affidabile). Lo vedremo brevemente più avanti.

Protocollo TCP

Il protocollo TCP è stato progettato per fornire un flusso di byte affidabile (reliable), da sorgente a destinazione, su una rete non affidabile. Dunque, offre un servizio reliable e connection oriented.

Ricordiamo che una **internet** (con la "i" iniziale minuscola) si distingue da una singola rete in quanto è formata da reti distinte che possono essere completamente diverse per quanto riguarda topologia, capacità, ritardi, dimensioni dei pacchetti e altro; allora, *TCP fu progettato per adattarsi dinamicamente alle proprietà dell'internet e per essere robusto nei confronti di tutta una serie di possibili "quasi"*.

Il protocollo TCP fu definito formalmente nella **RFC 793**. Con il passare del tempo, furono però scoperti errori ed inconsistenze, che furono eliminati con le RFC successive: una versione sostanzialmente definitiva del TCP è quella contenuta nella **RFC 1122**, mentre ci sono alcune estensioni definite nella **RFC 1323**.

Entità di trasporto TCP

Ogni host che supporta TCP possiede una cosiddetta **entità di trasporto TCP**: essa gestisce i **flussi di dati TCP** e si interfaccia con il livello inferiore (ossia quindi con IP).

Per un host che deve trasmettere qualcosa attraverso la rete, il TCP si comporta nel modo seguente:

- accetta i dati dal livello *application*;
- li spezza in unità, dette **segmenti**, di dimensione massima 64 Kbyte;
- consegna i segmenti al livello *network*, occupandosi anche di eventuali *ritrasmissioni* qualora siano richieste.

Per un host che, invece, riceve dati, l'entità TCP si comporta nel modo seguente:

- riceve segmenti dal livello *network*;
- li rimette in ordine, eliminando buchi e doppioni;
- consegna i dati, in ordine, al livello *application*.

TCP fornisce un servizio full-duplex con gestione di ACK e controllo del flusso. Ricordiamo che il livello IP non fornisce alcuna garanzia sulla consegna corretta dei pacchetti IP; quindi è compito di TCP ritrasmetterli quando necessario. Analogamente, dato che i pacchetti IP possono arrivare nell'ordine sbagliato, è compito di TCP riordinarli nella sequenza corretta. In breve, quindi, TCP fornisce l'affidabilità che IP invece non può fornire.

Indirizzamento TCP

I servizi del TCP si ottengono creando una **connessione di livello transport** tra sorgente e destinazione, identificata da una coppia di *punti d'accesso*, detti **socket**. Ogni socket ha un **socket number** (*identificatore del socket*) che consiste della seguente coppia di valori:

IP address: Port number

Il **port number** è lungo 16 bit e corrisponde sostanzialmente ad un indirizzo di I/O dell' host; l'**indirizzo IP** (lungo 32 bit per IPv4 oppure 128 bit per IPv6) è quello dell'host che ha creato il socket.

Avendo a disposizione 16 bit per il port number, si possono avere al più $2^{16}=1024$ possibilità; tuttavia, i port number minori di 256 corrispondono alle cosiddette **well-known port**, riservate solo per i *servizi standard*:

Port number	Servizio
7	Echo
20	Ftp (control)
21	Ftp (data)
23	Telnet
25	Smtp
80	Http
110	Pop versione 3

Quindi, ad esempio, qualsiasi processo che desideri stabilire una connessione con un host al fine di trasferire file tramite il TCP, può connettersi alla porta 21 dell'host destinazione per contattare il **demone FTP** (ossia un programma permanentemente attivo, che rileva la richiesta dell'host mittente e fa il possibile per avviare una connessione con esso).

Poiché le connessioni TCP, che sono full duplex e point-to-point ([1]), sono identificate dalla coppia di socket number alle due estremità, è possibile che su un singolo host più connessioni siano attestate localmente sullo stesso socket number. Quindi, un socket può essere usato da più di una connessione contemporaneamente: ogni connessione sarà caratterizzata dalla copia di socket da entrambi i lati, ossia (*socket1*,*socket2*).

Le connessioni TCP trasportano un flusso di byte. Questo significa che, ad esempio, se il *processo mittente* (di livello application) invia 4 blocchi da 512 byte l'uno, quello destinatario può ricevere:

- 8 "pezzi" da 256 byte;
- 1 "pezzo" da 2.048 byte;
- ecc.

Ci pensano le entità TCP a suddividere il flusso in arrivo dal livello application in segmenti, a trasmetterli e a ricombinarli in un flusso che viene consegnato al livello application di destinazione.

Quando una applicazione passa dei dati a TCP, quest'ultimo potrebbe sia spedirli immediatamente sia salvarli in un buffer, a sua discrezione. Questa idea di **bufferizzare** i dati prima di inviarli è nata per migliorare l'efficienza della

[1] Ricordiamo che una connessione si dice **full-duplex** quando supporta la trasmissione contemporanea in entrambe le direzione; si parla inoltre di connessione **point-to-point** per indicare che essa ha solo due estremi: questo significa che TCP, al contrario di IP, non supporta il multicasting e tanto meno il broadcasting.

comunicazione, in quanto consente di spedire i dati solo quando sono in grande quantità. Ci sono tuttavia due eccezioni:

- in alcune occasioni, l'applicazione stessa desidera che i dati vengano trasmessi immediatamente; in questo caso, l'applicazione deve impostare ad 1 un apposito *flag*, nell' header TCP, denominato **PUSH**, che appunto forza il TCP a non ritardare la trasmissione;
- l'altra possibilità, per il livello application, di forzare l'invio immediato di dati è quando viene impostato ad 1 il *flag* **URGENT**: in pratica, quando l'applicazione mittente ha bisogno di inviare immediatamente delle informazioni di importanza critica, inserisce alcune informazioni di controllo nel flusso di dati e le passa al TCP assieme al flag URGENT settato ad 1; questo evento forza TCP a smettere di accumulare dati ed a trasmettere immediatamente tutto ciò che riguarda quella connessione. In ricezione, quando arrivano i dati con il flag URGENT, l'applicazione corrente viene interrotta, in modo che ci si possa dedicare ad esaminare i dati urgenti.

Caratteristiche principali del TCP

Le caratteristiche più importanti del protocollo TCP sono le seguenti:

- ogni byte del flusso TCP è numerato con un **numero di sequenza** a 32 bit, usato sia per il controllo di flusso che per la gestione degli ACK ([2]). In realtà, non è che ogni byte TCP porta con sé il proprio numero di sequenza, ma viene adottato un meccanismo che descriveremo più avanti;
- le entità TCP mittente e ricevente si scambiano dati sotto forma di **segmenti TCP**; un segmento TCP non può superare i **65535 byte** ed è il software TCP a stabilire la dimensione dei segmenti di volta in volta;
- un segmento TCP è formato da:
  - un **header**, a sua volta costituito da:
    - una parte fissa di 20 byte;
    - una parte opzionale;

[2] Nel caso di un host che spedisce dati con continuità alla velocità di 10 Mbps, teoricamente i numeri di sequenza possono essere riutilizzati dopo un'ora, ma in realtà è necessario molto più tempo;

  - i **dati** da trasportare, lunghi da zero ad un numero di byte tale che la lunghezza totale (incluso cioè l' header) non superi i 65535 byte;

- per la politica di trasmissione dei pacchetti, TCP usa un meccanismo di **sliding window** (*finestra scorrevole*) di tipo **go-back-n con timeout**: al momento di trasmettere un segmento, il mittente inizializza un timer; quando il segmento arriva a destinazione, l'entità TCP ricevente spedisce indietro un nuovo segmento (contenente eventualmente anche dati) che trasporta un **numero di ACK** uguale al successivo numero di sequenza che attende di ricevere. Se però il *timeout* della sorgente scade prima di ricevere l'ACK, il segmento viene ritrasmesso. Si noti che le dimensioni della finestra scorrevole e i valori degli ACK sono espressi in numero di byte, non in numero di segmenti.

Ci sono poi numerosi altri dettagli sui quali non entriamo in questa sede.

E' importante segnalare che esistono due limiti che impongono restrizioni sulla dimensione dei segmenti:

- il primo limite è che un segmento (incluso l' header) deve comunque entrare nel payload di un pacchetto IP ed anche i pacchetti IP hanno lunghezza totale (header+payload) massima di 65535 byte;
- il secondo limite è che ogni rete possiede una **unità massima di trasferimento** (**MTU**, *Maximum Transfer Unit*) per quanto riguarda i frame di livello fisico e ogni segmento deve entrare in un MTU. Generalmente, l'MTU è lungo poche migliaia di byte e quindi è proprio l'MTU a stabilire, di fatto, il limite superiore della dimensione del segmento. Se un segmento attraversa una serie di reti senza essere frammentato e poi arriva ad una rete in cui la MTU è inferiore alla dimensione del segmento, è compito del **router di confine** di tale rete frammentare il segmento in due o più segmenti più piccoli; in questo caso, ogni nuovo segmento ottiene il proprio header IP e TCP, il che fa si che lo spazio totale "sprecato" aumenti, in quanto ogni segmento addizionale aggiunge 40 byte di header extra, teoricamente inutili (sono gli stessi per tutti i segmenti in cui è stato scomposto il segmento di partenza).

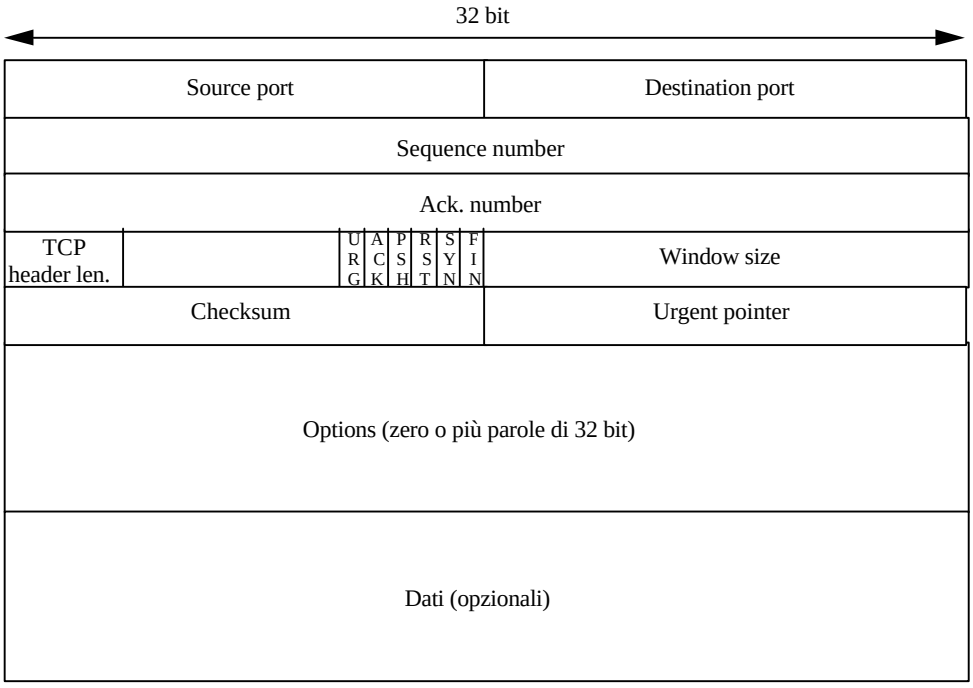
La possibilità che un singolo segmento venga frammentato da un router determina a sua volta una serie di rischi:

- può capitare che una parte dei segmenti trasmessi arrivino e siano confermati dall'entità TCP ricevente, ma la rimanente parte venga persa;
- i singoli segmenti possono arrivare fuori ordine;
- tutti i segmenti possono risultare così ritardati dalla rete da costringere il mittente a rispedirli;
- un segmento ritrasmesso può seguire un percorso diverso dall'originale e venir frammentato diversamente, cosicché arrivano sporadicamente in ricezione sezioni sia dell'originale che del duplicato.

Il TCP deve essere in grado di affrontare e risolvere efficacemente questi ed altri problemi.

Header di un segmento TCP

La figura seguente mostra la struttura di un **segmento TCP**, formato da un header ed una parte dati. L'header comincia sempre con una **parte fissa** da 20 byte ed una **parte opzionale** di lunghezza variabile (sempre multipla di parole da 32 bit):



Formato del segmento TCP

Ovviamente, dato che la lunghezza massima di un segmento TCP è 65535 byte, le opzioni TCP (se presenti) possono occupare fino a 65535-20-20=65495 byte di dati, dove 20 byte sono dell' header TCP e 20 dell' header IP (il segmento TCP deve sempre entrare in un pacchetto IP). Sono possibili anche segmenti senza dati, che sono generalmente usati per gli *ACK* ed i *messaggi di controllo*.

I campi dell' header TCP hanno le seguenti funzioni:

- i campi **Source port** e **Destination port** identificano gli *end point* (*locali* ai due host) della connessione. Essi, assieme ai corrispondenti *numeri IP*, sono gli identificatori dei due estremi della connessione TCP (da 48 bit l'uno). Ogni host decide come allocare le proprie porte successive alla 256.
- i campi **Sequence Number** (numero di sequenza) e **Acknowledgement Number** (numero di ack) sono lunghi entrambi 4 byte ed hanno il loro normale significato: il primo contiene il numero d'ordine del primo byte contenuto nel campo dati ([3]), mentre il secondo contiene il numero d'ordine del prossimo byte atteso (proveniente dal TCP mittente). Il motivo per cui tali campi sono da 4 byte è che, come detto, in un flusso TCP ogni byte viene numerato;
- il campo **TCP Header Length** esprime la lunghezza dell'header TCP in termini di parole da 32 bit. E' necessario in quanto il campo *options* è di dimensione variabile;
- a questo punto, ci sono 6 bit non utilizzati, seguiti da un campo di 6 flag:
  - il **flag URG** vale 1 se il campo *urgent pointer* viene usato mentre vale 0 altrimenti; il campo **Urgent Pointer** è un "puntatore ai *dati urgenti*": viene usato per indicare uno scostamento in byte, a partire dal numero di sequenza attuale, dove possono essere trovati i dati urgenti (questo consente di usare messaggi di interruzione, in quanto il mittente può mandare un segnale al ricevente senza coinvolgere il TCP nel motivo dell'interruzione);
  - il **flag ACK** vale 1 se l'**ACK number** è valido (cioè se il segmento convoglia anche un ACK) e 0 altrimenti: se ACK=0, il segmento non contiene un ACK e quindi il campo *ACK number* viene ignorato;

[3] Si spiega così la proprietà del TCP secondo cui ogni byte del flusso TCP è numerato: in realtà, ciascun segmento TCP porta con sé il numero di sequenza del primo byte contenuto, per cui i numeri di sequenza dei byte successivi (appartenenti allo stesso segmento) sono dedotti di conseguenza.

- il **flag PSH** indica la presenza di *dati urgenti* (pushed data), che cioè dovranno essere consegnati all'applicazione destinataria senza aspettare che il buffer si riempia;
- il **flag RST** indica la richiesta del mittente di *reset della connessione*: serve in presenza di "problemi", come ad esempio una connessione diventata instabile oppure l'arrivo di un segmento non valido oppure il rifiuto di aprire una connessione;
- il **bit SYN** viene usato per creare una connessione TCP: la richiesta di connessione è caratterizzata da SYN=1 e ACK=0 (per indicare che il campo ACK Number non è valido); la risposta a tale segmento è caratterizzata poi da SYN=1 e ACK=1, in quanto si tratta di un ACK. In pratica, dunque, il bit SYN viene usato per denotare messaggi di tipo `CONNECTION REQUEST` (pacchetto SYN) e `CONNECTION ACCEPTED` (pacchetto SYN+ACK);
- il **bit FIN** viene invece usato per rilasciare una connessione, in quanto specifica che il mittente non ha ulteriori dati da spedire. Tuttavia, dopo aver chiuso una connessione, un processo può comunque continuare a ricevere dati indefinitamente. Da notare che sia i segmenti SYN sia i segmenti FIN possiedono numeri di sequenza e quindi vengono elaborati nell'ordine corretto;

- il controllo di flusso, per una connessione TCP, è di tipo *sliding window* di dimensione variabile: allora, il campo **Window size** dice quanti byte possono essere spediti a partire da quello (compreso) che viene confermato con l'ACK number ([4]). Ad esempio, se il valore di questo campo fosse pari a 0, significherebbe che i byte fino a quello di numero `ACKNumber-1` (incluso) sono stati ricevuti, ma che il ricevente ha bisogno di una pausa e non desidera ulteriori dati per il momento (come dire "Ricezione completata, ma adesso fermati per un po', riprenderai quando ti arriverà un uguale *ACK number* ma con un valore di window size diverso da zero");
- il campo **TCP Checksum** contiene un valore da usarsi per verificare l'eventuale presenza di errori di trasmissione (con un meccanismo molto

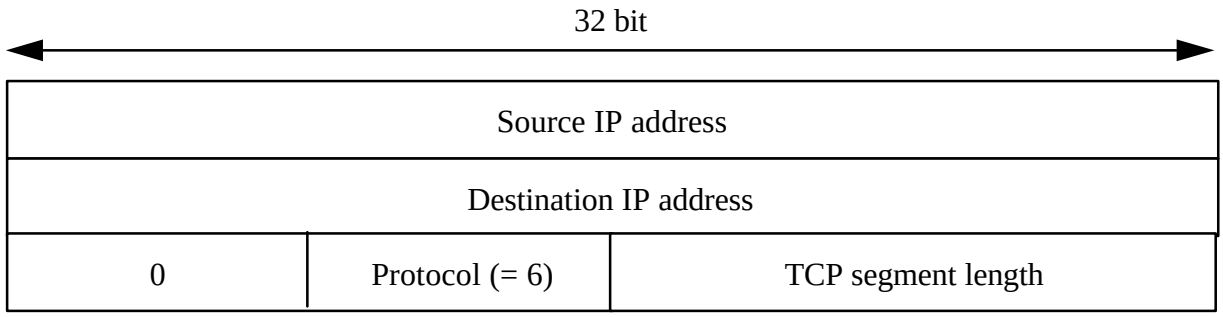
[4] E' ovvio che il TCP, tramite il meccanismo degli ACK, non conferma i singoli byte, bensì i segmenti. Tuttavia, dato che il campo sequence number contiene il numero d'ordine del primo byte contenuto nel payload del segmento TCP, è evidente che è del tutto equivalente dire che gli ACK sono riferiti ai segmenti oppure al primo byte di dati in essi contenuti.

simile a quello usato da IP); il calcolo include però anche uno *pseudoheader* , di cui si parla più avanti;

- infine, il campo **Options** serve essenzialmente ad aggiungere servizi extra non compresi nell' header regolare. Tra le più importanti opzioni, negoziabili al setup della connessione, citiamo le seguenti: dimensione massima dei segmenti da spedire; uso di selective repeat al posto del go-back-n; uso di NAK.

Pseudoheader e calcolo della checksum

Il campo **checksum** dell' header TCP è usato per garantire la correttezza dei dati. Esso verifica la correttezza dei bit presenti nell' header TCP, nel payload TCP e nel cosiddetto **pseudoheader**, avente la seguente struttura:



PseudoHeader TCP

Il fatto che nel calcolo del checksum entri anche lo **pseudoheader** è in effetti in aperta violazione della "gerarchia a livelli", dato che il livello TCP in questo calcolo opera su indirizzi IP, ossia su dati appartenenti al livello sottostante ([5]).

Lo pseudoheader non è un "qualcosa" che viene trasmesso, ma che precede solo concettualmente l' header. I suoi campi hanno le seguenti funzioni:

Source IP address, destination IP address	indirizzi IP (a 32 bit) di sorgente e destinatario.
Protocol	il codice numerico del protocollo TCP (pari a 6).
TCP segment length	il numero di byte del segmento TCP, header compreso.

[5] Da notare che il TCP "conosce" gli indirizzi IP del mittente e del destinatario in quanto essi, insieme ai port number, individuano gli estremi della connessione TCP.

L'inclusione dello pseudoheader nel calcolo della checksum TCP serve per individuare i pacchetti consegnati alla destinazione sbagliata, sia pure in violazione della gerarchia come si diceva poco fa.

Quando viene eseguito il calcolo della checksum in trasmissione, si procede nel modo seguente:

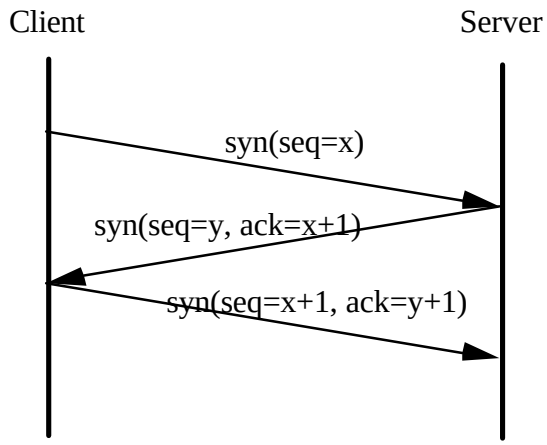
- in primo luogo, il campo *checksum* dell'header TCP viene posto uguale a 0;
- in secondo luogo, il payload viene incrementato con 8 bit posti tutti a 0 nel caso che la sua lunghezza (espressa in numero di byte) sia un numero dispari;
- a questo punto, l'**algoritmo di checksum** somma tutte le parole di 16 bit in complemento ad 1 e poi memorizza, nel campo *checksum*, il complemento ad 1 di tale somma. In questo modo, quando il ricevente esegue a sua volta la checksum sull'intero segmento (incluso il campo *checksum*) per verificare la bontà dei dati ricevuti, il risultato da ottenere deve essere 0. Se invece il risultato è diverso da 1, significa che il segmento ha subito degli errori di trasmissione (si dice che è "corrotto") e viene scartato dalla destinazione, che aspetterà la ritrasmissione dalla sorgente.

Attivazione della connessione TCP

Quando un host (diciamo il *client*) vuole attivare una connessione TCP con un altro host (diciamo il *server*), usa il protocollo **three-way handshake**:

- il server esegue due primitive, `listen()` e poi `accept()`, rimanendo così in attesa di una richiesta di connessione su un determinato port number e, quando essa arriva, accettandola;
- il client esegue la primitiva `connect()`, specificando host, port number e altri parametri (quali la dimensione massima dei segmenti) per stabilire la connessione; tale primitiva causa l'invio di un segmento TCP col bit `syn` a uno e il bit `ack` a zero;
- quando tale segmento arriva a destinazione, l'entità di livello transport controlla se c'è un processo in ascolto sul port number in questione:

- se non c'è nessuno in ascolto, invia un segmento di risposta col bit rst a uno, per rifiutare la connessione;
- altrimenti, consegna il segmento arrivato al processo in ascolto; se esso accetta la connessione, l'entità invia un segmento di conferma, con entrambi i bit syn ed ack ad uno, secondo lo schema sotto riportato.



*Attivazione di una connessione TCP con il meccanismo del three-way handshake: il **pacchetto SYN** inviato dal mittente serve a richiedere la connessione; il ricevente, se accetta la connessione, risponde con un pacchetto SYN+ACK; ricevendo tale pacchetto, il mittente lo conferma con un ACK*

I valori di x e y sono ricavati dagli host sulla base dei loro clock di sistema; il valore si incrementa di una unità ogni 4 microsecondi.

Il numero di sequenza iniziale su una connessione è sempre diverso da 0 per motivi di sicurezza.

Per ulteriore sicurezza, quando un host si guasta, non può essere re-inizializzato per il tempo massimo di vita di un pacchetto (pari a 120 secondi), in modo da essere sicuri che nessun pacchetto delle connessioni precedenti stia ancora viaggiando nella rete.

Rilascio della connessione TCP

Il rilascio di una connessione TCP avviene considerando la connessione full-duplex come una coppia di connessioni simplex indipendenti e si svolge nel seguente modo:

- quando una delle due parti non ha più nulla da trasmettere, invia un **segmento FYN** (ossia un segmento con il flag FYN posto ad 1);

- quando esso viene confermato, la connessione in uscita viene rilasciata; tuttavia, i dati possono comunque continuare a fluire indefinitamente nell'altra direzione;
- quando anche l'altra parte completa lo stesso procedimento e rilascia la connessione nell'altra direzione, la connessione full-duplex termina.

In pratica, quindi, si tratta di chiudere ciascuna connessione simplex in modo indipendente dalla gemella. Generalmente, sono necessari 4 segmenti TCP per chiudere una connessione; si potrebbe ridurre tale numero a 3 se il primo ACK ed il secondo FYN fossero inglobati in un unico segmento.

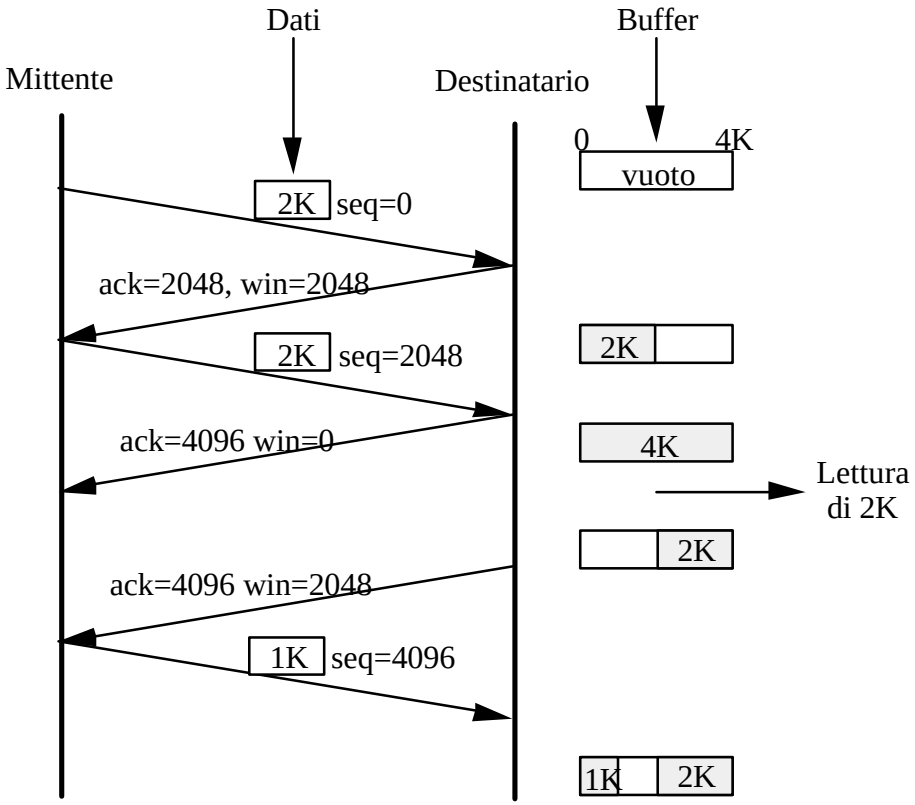
In linea di massima, entrambi gli estremi di una connessione TCP possono spedire contemporaneamente un segmento FYN: quando questi vengono confermati nel modo descritto prima, la connessione viene chiusa. Non esiste quindi alcuna differenza tra una chiusura simultanea e una chiusura in sequenza.

Il protocollo di gestione delle connessioni si rappresenta comunemente come una **macchina a stati finiti**. Questa è una rappresentazione molto usata nel campo dei protocolli, perché permette di definire, con una certa facilità e senza ambiguità, protocolli anche molto complessi. Ad ogni modo, non ce ne occupiamo in questa sede.

Politica di trasmissione TCP

L'idea di fondo applicata dal TCP nei riguardi della gestione delle finestre è la seguente: la dimensione delle **finestre scorrevoli** (*sliding window*) non è strettamente legata agli ACK (come invece di solito avviene per la maggior parte dei protocolli data link), ma viene continuamente adattata mediante un *dialogo* fra destinazione e sorgente. In particolare, quando la destinazione invia un ACK di conferma, dice anche quanti ulteriori byte possono essere spediti.

Ad esempio, supponiamo che il ricevente abbia un buffer di 4096 byte (4K) e che questo risulti completamente vuoto, come indicato nella figura seguente (nella parte destra, in alto):



Esempio di controllo del flusso TCP

Se il mittente trasmette un segmento da 2K e questo viene correttamente ricevuto, il ricevente conferma l'avvenuta ricezione (ack=2048) e, dato che il suo buffer possiede solo 2K liberi (almeno fino a quando l'applicazione non va a prendersi i dati), annuncia una finestra di ricezione di 2K a partire dal prossimo byte atteso (win=2048).

A questo punto, il mittente trasmette altri 2K, che vengono consegnati; il buffer del ricevente diventa pieno e quindi esso, insieme alla conferma di ricezione (ack=4096), invia anche l'annuncio di una finestra vuota (win=0). In questo caso, il mittente deve fermarsi fin quando l'applicazione sul lato ricevente non rimuove qualche altro dato dal buffer, consentendo al ricevente di annunciare una finestra non più vuota. Quando il ricevente dispone di nuovo spazio vuoto nel proprio buffer, invia al trasmittente un pacchetto identico all'ultimo inviato, ma con win≠0.

Può capitare che il ricevente abbia inviato un messaggio con win≠0, ma che questo sia andato perso, per cui il mittente continua erroneamente a credere di non poter inviare niente, il che porterebbe ad una situazione di stallo. Viene allora avviato un timeout dopo l'invio del pacchetto con win≠0: se il ricevente non dovesse ricevere dati entro la scadenza del timeout, manda al mittente un pacchetto cosiddetto **sonda** (*probe*); il mittente, ricevendo tale pacchetto, deduce quale sia

stato il problema ed invia conferma di ricezione del pacchetto sonda, dopodiché il ricevente invia nuovamente il pacchetto con win≠0.

Ad ogni modo, anche se riceve win=0, il mittente può comunque inviare **dati urgenti** o richieste di ri-invio dell'ultimo ACK spedito (per evitare lo stallo se esso si è perso).

Da notare che non è obbligatorio che il mittente invii i dati appena questi gli giungono dall'applicazione, così come anche gli ACK non devono necessariamente partire il più presto possibile; la gestione dei ritmi di trasmissione viene eseguita con criteri di ottimizzazione dell'efficienza di utilizzo della comunicazione.

"Opzioni" attualmente previste per una connessione TCP

Si è visto che il campo *Window Size* dell'header TCP non permette di esprimere dimensioni della finestra di trasmissione superiori a 64 Kbyte ([6]). Tuttavia, per un collegamento caratterizzato da una grande capacità o da un grande ritardo o da entrambe le cose, la finestra da 64 Kbyte è spesso un problema. Per esempio, consideriamo una **linea T3**, caratterizzata cioè da una capacità di 44.736 Mbps: il tempo necessario per inviare una finestra di 64 Kbyte (= 64×1024×8 bit) è di circa 12 ms; supponendo che il tempo di propagazione di andata e ritorno (il cosiddetto **round-trip time**, **RTT**) sia di 50 ms, è evidente che il mittente resta inattivo per circa i ¾ del tempo, aspettando l'ACK. Se poi consideriamo una comunicazione satellitare, la situazione è ancora peggiore, dato che il RTT è dell'ordine di 200-300 ms. Allora, una finestra più lunga consentirebbe al mittente di continuare a spedire dati, aumentando così l'efficienza di utilizzo della linea di comunicazione.

Per superare, allora, il limite dei 64 Kbyte imposto dal campo *Window Size*, nella **RFC 1323** è stata proposta una apposita *opzione* (da indicarsi, quindi, nel campo *Options* dell'header TCP), denominata **Windows Scale**: essa consente al mittente ed al ricevente di negoziare un *fattore di scala* della finestra. In pratica, questo fattore di scala consente di estendere la dimensione massima della finestra di trasmissione fino a $\mathbf{2^{32}}$ **byte**. Molte delle attuali realizzazione di TCP supportano questa opzione.

Un'altra opzione di rilievo, forse la più importante, è quella che consente ad un host di specificare il più grande segmento TCP che è in grado di accettare. Utilizzare segmenti grandi è generalmente più efficiente che utilizzare segmenti piccoli, in quanto l'header di 20 byte può essere meglio "ammortizzato" su una

[6] Vale la pena ricordare che la lettera "K" indica il fattore moltiplicativo 1024, mentre la lettere "k" corrisponde a 1000. Quindi, 64 Kbyte corrispondono a 64×1024 byte (cioè anche 64×1024×8 bit), mentre invece 64 kbyte corrisponderebbero a 64×1000 byte (ossia anche 64×1000×8 bit).

quantità superiore di dati. Tuttavia, gli host più piccoli potrebbero non essere in grado di gestire segmenti sufficientemente grandi: allora, durante la creazione della connessione TCP, ogni estremo può annunciare il suo massimo valore e vedere quello dell'altro estremo; viene ovviamente scelto il valore più piccolo tra i due. Se uno dei due estremi non usa questa opzione, il valore di default prevede 536 byte di payload, il che significa sostanzialmente che tutti gli host impieganti TCP devono poter accettare segmenti TCP lunghi 20+536=556 byte.

Un'altra opzione è stata prevista dalla **RFC 1106** e prevede che venga utilizzato, come protocollo per la gestione delle ritrasmissioni, il **selective repeat** al posto del *go-back-n* (che è quello di default): in questo modo, in presenza di un segmento risultato corrotto, non è necessario che il mittente riprenda la trasmissione da quel segmento e ritrasmetta anche i segmenti successivi (che potrebbero essere giunti senza errori), ma è sufficiente che venga ritrasmesso solo il segmento corrotto.

La RFC 1106 ha inoltre introdotto l'uso dei cosiddetti **NACK**, che il ricevente può utilizzare per richiedere la ritrasmissione di un segmento specifico (o più segmenti specifici). Con l'uso dei NACK, dopo aver ricevuto tutti i dati ed averli memorizzati nel proprio buffer, il ricevente può confermare la loro ricezione in un colpo solo, riducendo così il numero di dati trasmessi.

Controllo congestione con TCP

Quando il carico di dati immesso in una rete è maggiore di quello che la rete è in grado di smaltire, si presenta il problema della **congestione**. Nonostante anche il livello network (cioè IP) si preoccupi delle congestioni, il grosso del lavoro è eseguito da TCP, in quanto l'unica soluzione possibile per una congestione è la riduzione del flusso dei dati.

Almeno a livello teorico, la congestione potrebbe essere affrontata in modo molto semplice: basterebbe non inserire un nuovo pacchetto nella rete fin quando uno vecchio non ne esce (cioè non viene consegnato). TCP cerca di ottenere proprio questo obbiettivo tramite una manipolazione dinamica delle dimensioni delle cosiddette **finestre**.

Il primo problema da affrontare è quello di individuare la presenza di una **congestione**. Attualmente, dato che le moderne linee di trasmissione sono estremamente affidabili, l'eventuale scadenza dei timeout è addebitabile praticamente sempre a problemi di congestione e non più ad eventuali errori di trasmissione che hanno fatto scartare pacchetti in ricezione. Quindi, tutti gli

algoritmi del protocollo TCP assumono che, se gli ACK non tornano in tempo, ciò sia dovuto a congestione della subnet piuttosto che a errori di trasmissione. Di conseguenza, TCP è preparato ad affrontare due tipi di problemi: scarsità di buffer a destinazione o congestione della subnet.

Ciascuno dei problemi viene gestito da una specifica **finestra** mantenuta dal mittente:

- la **finestra del buffer** del ricevitore, che indica quanti byte il ricevitore ha liberi nel proprio buffer di ricezione;
- la **congestion window**, che rappresenta quanto il mittente può spedire senza causare congestione.

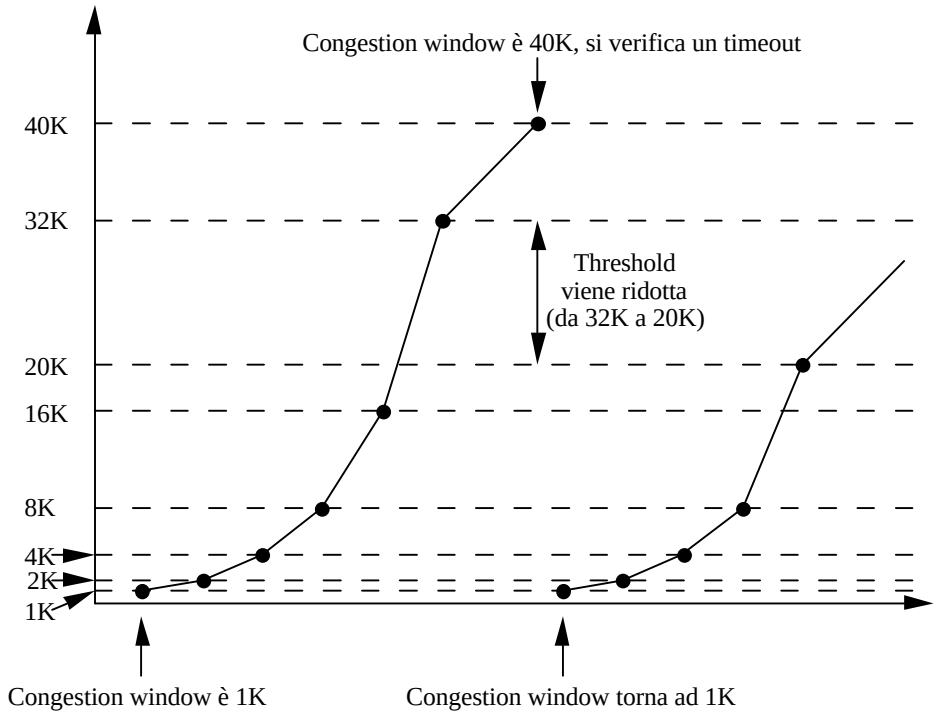
Il mittente si regola sulla più piccola delle due, ossia spedisce i propri dati sulla base della finestra più piccola. Ad esempio, se il ricevente dice "Spedisci 8K" ma il mittente sa che un insieme di dati maggiore di 4K intaserebbe la rete, spedisce 4K. Viceversa, se il ricevente dice "Spedisci 8K" ma il ricevente ne vorrebbe spedire 32K, alla fine spedisce 8K

La **congestion window** viene gestita in questo modo:

- il valore iniziale è pari alla dimensione del massimo segmento usato nella connessione;
- ogni volta che un ACK torna indietro in tempo (cioè prima che scada il timeout), la finestra si raddoppia: ad esempio, se è arrivato l'ACK di un segmento di 8K, la finestra successiva è di 16K; se arriva l'ACK anche del segmento successivo, la finestra viene aumentata a 32K e così via; se tutti gli ACK arrivano regolarmente, si ottiene un aumento esponenziale della dimensione della finestra
- questo raddoppio della dimensione di una finestra per ogni ACK ricevuto può però proseguire fino a un valore **threshold** (*soglia*), inizialmente pari a 64 Kbyte: quando esso viene raggiunto, la dimensione della finestra viene aumentata linearmente della dimensione massimo di 1 segmento;
- quando si verifica un timeout per un segmento:
    - il valore di threshold viene impostato alla metà della dimensione della congestion window attuale;

- la dimensione della congestion window viene impostata alla dimensione del massimo segmento usato nella connessione.

Vediamo un esempio. ipotizzando segmenti di dimensione 1 Kbyte, threshold a 32 Kbyte e congestion window arrivata a 40 Kbyte:



Esempio di controllo della congestione TCP

I segmenti sono dunque ciascuno da 1K e quindi la congestion window è inizialmente anch'essa di 1K. Vengono spediti i primi pacchetti e arrivano regolarmente gli ACK per i primi di essi: la congestion window assume, in progressione, i valori 2K (primo ACK), 4K (secondo ACK), 8K (terzo ACK), 16K (quarto ACK), 32K (quinto ACK); a questo punto, è stata raggiunta la soglia, per cui la congestion window aumenta adesso di 1 K ad ogni nuovo ACK ricevuto in tempo: la dimensione della congestion window assume dunque i valori 33K, 34K e così via , crescendo quindi in maniera più "lenta" rispetto a prima (come evidenziato dalla minore pendenza del grafico). Si arriva in tal modo fino a 40 K. Si suppone, a questo punto, che scada il timeout sul 14° segmento; dando per scontato che questo sia accaduto per problemi di congestione, il valore di soglia della finestra viene posto a 20 K (metà della dimensione attuale della finestra) e la dimensione della finestra torna a 1K (massima lunghezza dei segmenti utilizzati).

Algoritmo a partenza lenta (slow start)

Si è detto che la finestra di congestione cresce (in modo esponenziale) fino a quando non scade un timeout oppure fin quando non si raggiunge la finestra del ricevente. Questo meccanismo parte dalla seguente idea di fondo: se trasmissioni da 1024 byte, 2048 byte e 4096 byte funzionano perfettamente, mentre una trasmissione di 8192 byte subisce la scadenza del timeout, per evitare congestioni la finestra di congestione dovrebbe essere uguale a 4096. Fino a quando la finestra di congestione rimane di 4096 byte, nessun segmento più lungo verrà mai inviato indipendentemente dalla finestra del ricevente.

Questo algoritmo viene detto **slow start** (*a partenza lenta*), anche se in realtà non è affatto "lento", in quanto esponenziale. Tutte le implementazioni concrete di TCP devono supportarlo.

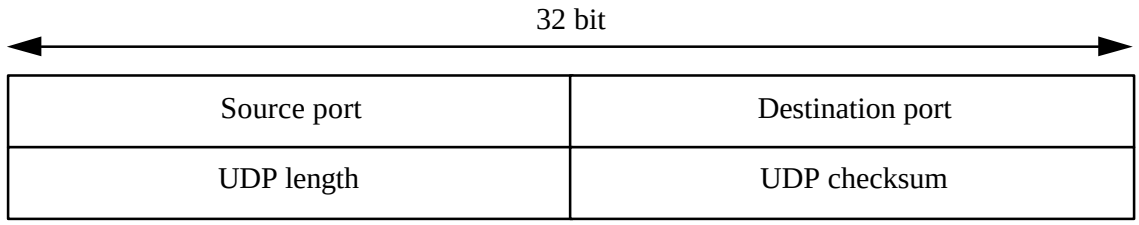
Quindi, riprendendo anche l'esempio del paragrafo precedente, nel momento in cui scade un timeout, la soglia viene posta a metà della finestra corrente (ad esempio, da 40K a 20K) e l'algoritmo slow start riparte di nuovo.

Se non scadono timeout, la finestra di congestione continua a crescere fino alla dimensione della finestra del ricevente; a quel punto, essa rimarrà costante fin quando non scadrà un timeout oppure fin quando la finestra del ricevente non cambia dimensione.

Il protocollo UDP

Il livello di trasporto di una architettura TCP/IP (ad esempio Internet ma non solo) fornisce anche un protocollo non connesso e non affidabile, utile per inviare dati senza stabilire connessioni. Si tratta del **protocollo UDP**.

Un **segmento UDP** consiste di un **header** del tipo riportato nella prossima figura, di lunghezza fissa (8 byte), e di una **parte dati** di lunghezza variabile:



Header UDP

I campi dell' header hanno un ovvio significato: *Source Port* e *Destination Port* hanno lo stesso significato che assumono nell' header TCP; il campo *UDP length* contiene la lunghezza complessiva del segmento UDP (8 byte di header + 1 o più byte di dati); il campo *UDP checksum* copre lo stesso pseudoheader usato per il TCP, l'header TCP e il payload UDP.

Nel caso in cui la lunghezza complessiva del pacchetto UDP non sia data da un numero pari di byte, si aggiungono in coda il numero di bit necessari (**bit di padding**) per ottenere questa caratteristica.

La funzione di **calcolo della checksum** può essere disattivata, tipicamente nel caso di *traffico in tempo reale* (come voce e video), per il quale è in genere più importante mantenere un elevato tasso di arrivo dei segmenti piuttosto che evitare i rari errori che possono accadere. Se viene disattivato, il campo checksum riporta il valore 0. Se invece il calcolo della checksum viene effettuato ed il risultato è 0, per evitare confusione tale risultato viene codificato come sequenza di 1.

Una cosa interessante da notare è che l' header di un segmento UDP ha lunghezza fissa, mentre quello di un segmento TCP ha una lunghezza variabile (a causa della variabilità del campo destinato alle opzioni).

Autore: **Sandro Petrizzelli**
e-mail: sandry@iol.it
sito personale: http://users.iol.it/sandry