Appunti del corso di
Reti di elaboratori

PROF. G. BONGIOVANNI

6) Il livello quattro (Transport)

Il livello transport è il cuore di tutta la gerarchia di protocolli. Il suo compito è di fornire un trasporto affidabile ed efficace dall'host di origine a quello di destinazione, indipendentemente dalla rete utilizzata.

Questo è il livello in cui si gestisce per la prima volta (dal basso verso l'alto) una conversazione diretta, cioè senza intermediari, fra sorgente e destinazione.

Da ciò discende che il software di livello transport è presente solo sugli host, e non nei router della subnet di comunicazione.

Servizi offerti dal livello transport
I servizi principali offerti ai livelli superiori sono vari tipi di trasporto delle informazioni fra una transport entity su un host e la sua peer entity su un altro host.

Naturalmente, tali servizi sono realizzati dal livello transport per mezzo dei servizi ad esso offerti dal livello network.

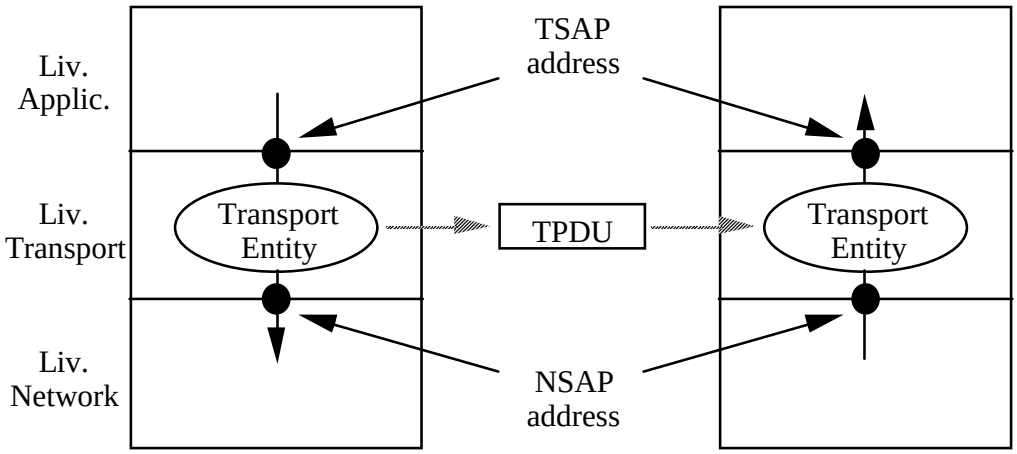

Figura 6-1: I servizi transport sono basati sui servizi network

Così come ci sono due tipi di servizi di livello network, ce ne sono due anche a livello transport:
- servizi affidabili orientati alla connessione (tipici di questo livello);
- servizi datagram (poco usati in questo livello).

Essi sono molto simili, come caratteristiche, a quelli corrispondenti del livello network, ed hanno gli analoghi vantaggi e svantaggi.

Ma allora, perché duplicare le cose in due diversi livelli? Il fatto è che l'utente (che accede ai servizi di rete dall'alto) non ha alcun controllo sulla subnet di comunicazione, e vuole comunque certe garanzie di servizio (ad esempio, il trasferimento corretto di un file). Dunque, tali garanzie devono essere fornite al di fuori della subnet, per cui devono risiedere in un livello superiore a quello network. In sostanza, il livello transport permette di offrire un servizio più affidabile di quanto la subnet sia in grado di fare.

Inoltre, ha un altro importante scopo, quello di isolare i livelli superiori dai dettagli implementativi della subnet di comunicazione. Ottiene ciò offrendo un insieme di primitive (di definizione dei servizi) semplici da utilizzare ed indipendenti dai servizi dei livelli sottostanti. Questo perché mentre solo poche persone scrivono componenti software che usano i servizi di livello network, molte scrivono applicazioni di rete, che si basano sui servizi di livello transport.

Un ultimo aspetto riguarda la possibilità di specificare la ***QoS*** (***Quality of Service***) desiderata. Questo in particolare è adatto soprattutto ai servizi connection oriented, nei quali il richiedente può specificare esigenze quali:

- massimo ritardo per l'attivazione della connessione;
- throughput richiesto;
- massimo ritardo di transito ammesso;
- tasso d'errore tollerato;
- tipo di protezione da accessi non autorizzati ai dati in transito.

In questo scenario:

- le peer entity avviano una fase di negoziazione per mettersi d'accordo sulla QoS, anche in funzione della qualità dei servizi di livello network di cui dispongono;
- quando l'accordo è raggiunto, esso vale per tutta la durata della connessione.

Primitive di definizione del servizio
Esse definiscono il modo di accedere ai servizi del livello.

Tipicamente sono progettate in modo da nascondere i dettagli della subnet (di più, in modo da essere indipendenti da qualunque subnet) ed essere facili da usare.

Ad esempio, questo può essere un tipico insieme di primitive:

Primitiva	TPDU spedito	Note
`accept()`	-	Si blocca finché qualcuno cerca di connettersi
`connect()`	`conn.request`	Cerca di stabilire una connessione
`send()`	`dati`	Invia dei dati
`receive()`	-	Si blocca finché arriva un TPDU
`disconnect()`	`disconn.request`	Chiede di terminare la connessione

Ad esempio, consideriamo i seguenti frammenti di codice di un'applicazione client-server:

Server

```
...
accept();
send();
receive();
...
```

Client

```
...
connect();
receive();
send();
...
disconnect();
```

Questi due frammenti sono in grado di portare avanti un dialogo senza errori (se il servizio utilizzato è affidabile, il che è la norma), ignorando completamente tutto quanto riguarda la meccanica della comunicazione.

6.1) Protocolli di livello transport

I protocolli di livello transport (sulla base dei quali si implementano i servizi) assomigliano per certi aspetti a quelli di livello data link. Infatti si occupano, fra le altre cose, anche di:

- controllo degli errori;
- controllo di flusso;
- riordino dei TPDU.

Ci sono però anche delle importanti differenze. Quella principale è che:

- nel livello data link fra le peer entity c'è un singolo canale di comunicazione;
- nel livello transport c'è di mezzo l'intera subnet di comunicazione.

Questo implica che, a livello transport:

- è necessario indirizzare esplicitamente il destinatario;
- è più complicato stabilire la connessione;

- la rete ha una capacità di memorizzazione, per cui dei TPDU possono saltare fuori quando la destinazione meno se li aspetta;
- buffering e controllo di flusso richiedono un approccio differente che nel livello data link, a causa del numero molto variabile di connessioni che si possono avere di momento in momento.

6.2) Indirizzamento

Quando si vuole attivare una connessione, si deve ovviamente specificare con chi la si desidera. Dunque, si deve decidere come è fatto l'indirizzo di livello transport, detto ***TSAP address*** (***Transport Service Access Point address***).

Tipicamente un TSAP address ha la forma

(NSAP address, informazione supplementare)

Ad esempio, in Internet un TSAP address (ossia un indirizzo TCP o UDP) ha la forma:

(IP address:port number)

dove IP address è il NSAP address, e port number è l'informazione supplementare.

Questo meccanismo di formazione degli indirizzi dei TSAP ha il vantaggio di determinare implicitamente l'indirizzo di livello network da usare per stabilire la connessione.

In assenza di tale meccanismo, diviene necessario che l'architettura preveda un servizio per effettuare il mapping fra gli indirizzi di livello transport e i corrispondenti indirizzi di livello network.

6.3) Attivazione della connessione

Questa operazione sembra facile ma non lo è, perché la subnet può perdere o duplicare (a causa di ritardi interni) dei pacchetti.

Ad esempio, a causa di ripetuti ritardi nell'invio degli ack può succedere che vengano duplicati e successivamente arrivino in perfetto ordine a destinazione tutti i pacchetti precedentemente generati nel corso di una connessione. Ciò in linea di principio può significare che l'attivazione della connessione e tutto il suo svolgimento abbiano luogo due volte.

Si immaginino le conseguenze di tale inconveniente se lo scopo di tale connessione fosse stato la richiesta (a una banca) di versare un miliardo sul conto di un personaggio dalla dubbia onestà.

Il problema di fondo risiede nella possibile esistenza di ***duplicati ritardatari*** che arrivano a destinazione molto dopo essere partiti.

Già sappiamo che, una volta che la connessione è stabilita, un qualunque protocollo a finestra scorrevole è adatto allo scopo. Infatti durante il setup della connessione le peer entity si accordano sul numero iniziale di sequenza, e quindi i doppioni ritardatari non vengono accettati. Viceversa, per risolvere il problema dei duplicati relativi alla fase di attivazione della connessione esiste una soluzione detta ***three-way handshaking*** (Tomlinson, 1975).

Il protocollo funziona così:
- il richiedente invia un TPDU di tipo `conn.request` con un numero x proposto come inizio della sequenza;
- il destinatario invia un TPDU di tipo ack contenente:
    - la conferma di x;
    - la proposte di un proprio numero y di inizio sequenza;
- il richiedente invia un TPDU di tipo dati contenente:
    - i primi dati del dialogo;
    - la conferma di y.

I valori x e y possono essere generati, ad esempio, sfruttando l'orologio di sistema, **in modo da avere valori ogni volta diversi**.

In assenza di errori, il funzionamento è il seguente:

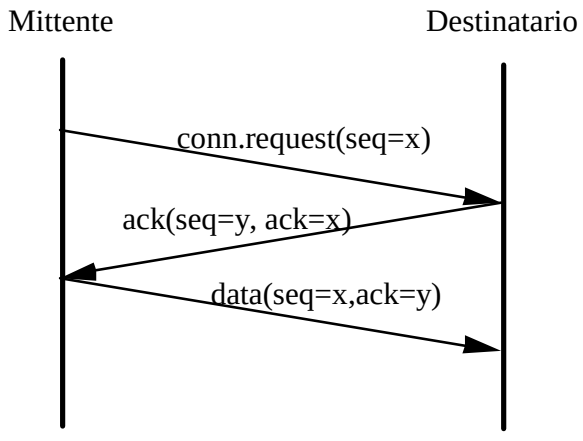

Figura 6-2: Three-way handshake

Se arriva a destinazione un duplicato della richiesta di attivazione, il destinatario risponde come prima (utilizzando **un nuovo numero di sequenza z**, diverso dal valore y usato la volta precedente) ma il mittente, che sa di non aver richiesto una seconda connessione, rifiuta tale risposta:

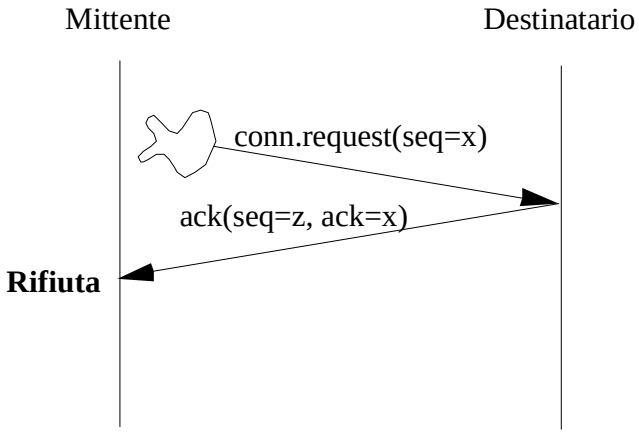

Figura 6-3: Duplicato della richiesta di attivazione

Se infine arrivano al destinatario sia un duplicato della richiesta di attivazione che un duplicato del primo TPDU dati, la situazione è la seguente:

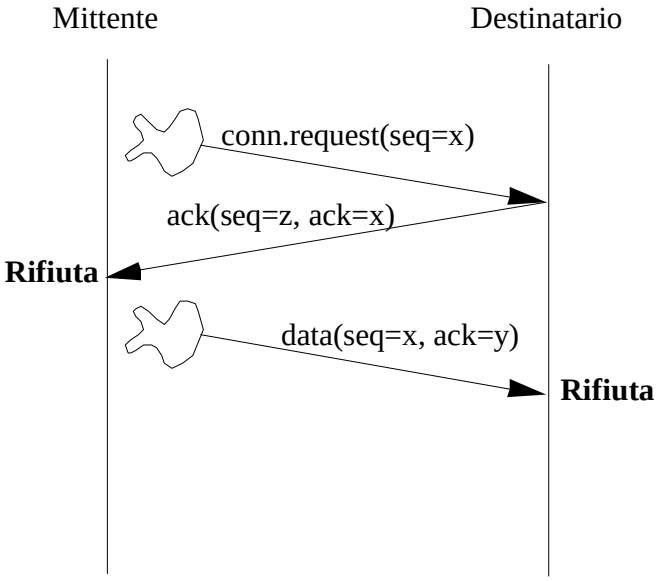

Figura 6-4: Duplicati della richiesta di attivazione e del primo TPDU dati

Anche in questo caso la connessione “spuria” non viene attivata. Infatti:

- il mittente rifiuta la risposta del destinatario, perché sa di non aver richiesto una seconda connessione (come nel caso precedente);
- il destinatario rifiuta il TPDU dati, perché questo reca un ack relativo ad un numero di sequenza (y) precedente e non a quello (z) da lui testé inviato.

6.4) Rilascio di una connessione

Rilasciare una connessione è più semplice che stabilirla, ma comunque qualche piccolo problema c'è anche in questa fase.

In questo contesto, rilasciare la connessione significa che l'entità di trasporto rimuove le informazioni sulla connessione dalle proprie tavole e informa l'utente di livello superiore che la connessione è chiusa.

Ci sono due tipi di rilasci:

- ***asimmetrico***;
- ***simmetrico***.

Nel primo caso (esemplificato dal sistema telefonico) quando una delle due parti "mette giù" si chiude immediatamente la connessione. Ciò però può portare alla perdita di dati, in particolare di tutti quelli che l'altra parte ha inviato e non sono ancora arrivati.

Nel secondo caso si considera la connessione come una coppia di connessioni unidirezionali, che devono essere rilasciate indipendentemente. Quindi, lungo una direzione possono ancora scorrere dei dati anche se la connessione lungo l'altra direzione è stata chiusa. Il rilascio simmetrico è utile quando un processo sa esattamente quanti dati deve spedire, e quindi può autonomamente decidere quando rilasciare la sua connessione in uscita.

Se invece le due entità vogliono essere d'accordo prima di rilasciare la connessione, un modo di raggiungere lo scopo potrebbe essere questo:

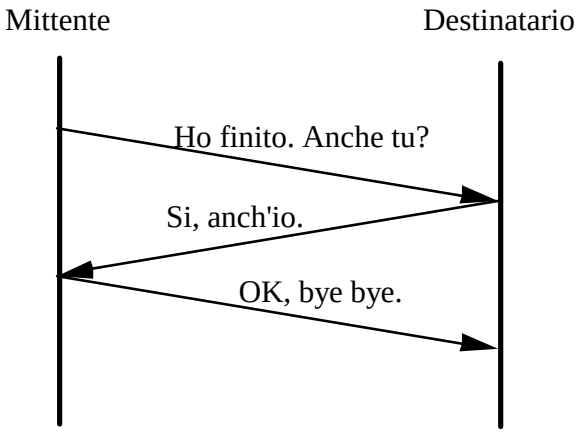

Figura 6-5: Semplice schema per il rilascio concordato della connessione

Purtroppo, le cose non sono così semplici: c'è un problema famoso in proposito, il ***problema delle due armate***:

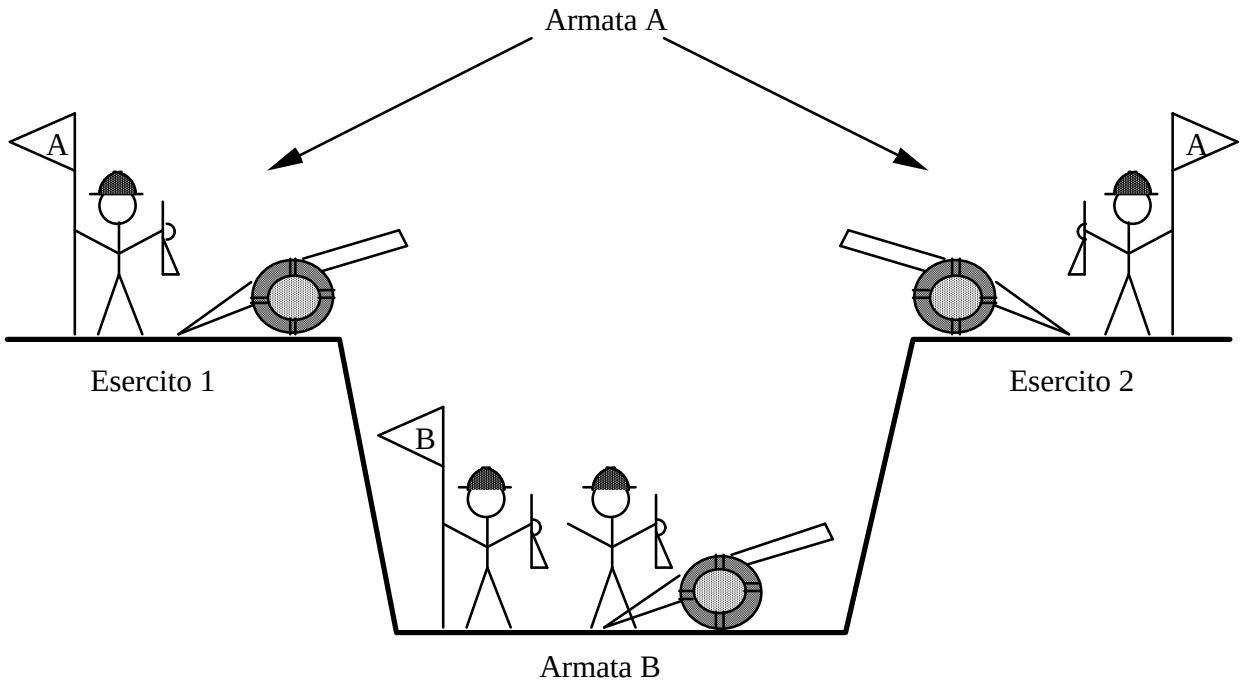

Figura 6-6: Il problema delle due armate

La definizione del problema è la seguente:
- i due eserciti che compongono l'armata A sono ciascuno più debole dell'esercito che costituisce l'armata B;
- l'armata A però nel suo complesso è più forte dell'armata B;
- i due eserciti dell'armata A possono vincere solo se attaccano contemporaneamente;
- i messaggi fra gli eserciti dell'armata A sono portati da messaggeri che devono attraversare il territorio dell'armata B, dove possono essere catturati.

Come fanno ad accordarsi gli eserciti dell'armata A sull'ora dell'attacco? Una possibilità è la seguente:
- il comandante dell'esercito 1 manda il messaggio "attacchiamo a mezzanotte. Siete d'accordo?";
- il messaggio arriva, un ok di risposta parte e arriva a destinazione, ma il comandante dell'esercito 2 esita perché non può essere sicuro che la sua risposta sia arrivata.

Si potrebbe pensare di risolvere il problema con un passaggio in più (ossia con un three-way handshake): l'arrivo della risposta dell'esercito 2 deve essere a sua volta confermato. Ora però chi esita è il comandante dell'esercito 1, perché se tale conferma si perde, l'armata 2 non saprà che la sua conferma alla proposta di attaccare è arrivata e quindi non attaccherà.

Aggiungere ulteriori passaggi non aiuta, perché c'è sempre un messaggio di conferma che è l'ultimo, e chi lo spedisce non può essere sicuro che sia arrivato. Dunque, non esiste soluzione.

Se ora sostituiamo la frase "attacchiamo l'armata B" con "rilasciamo la connessione", vediamo che si rischia effettivamente di non rilasciare mai una connessione.

Per fortuna, rilasciare una connessione è meno critico che attaccare un'armata nemica. Quindi, qualche rischio si può anche prendere.

Un protocollo di tipo three-way handshaking arricchito con la gestione di timeout è considerato adeguato, anche se non infallibile:
- il mittente invia un `disconn.request` e, se non arriva risposta entro un tempo prefissato (timeout), lo invia nuovamente per un massimo di n volte:
    - appena arriva una risposta (`disconn.request`) rilascia la connessione in ingresso e invia un ack di conferma;
    - se non arriva nessuna risposta, dopo l'ultimo timeout rilascia comunque la connessione in ingresso;
- il destinatario, quando riceve `disconn.request`, fa partire un timer, invia a sua volta un `disconn.request` e attende l'ack di conferma. Quando arriva l'ack o scade il timer, rilascia la connessione in ingresso.

Normalmente il funzionamento è il seguente:

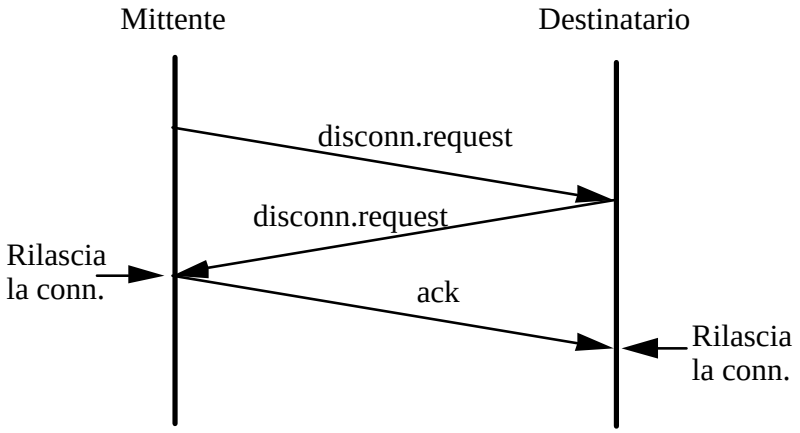

Figura 6-7: Rilascio concordato di una connessione transport

Se si perde l'ack:

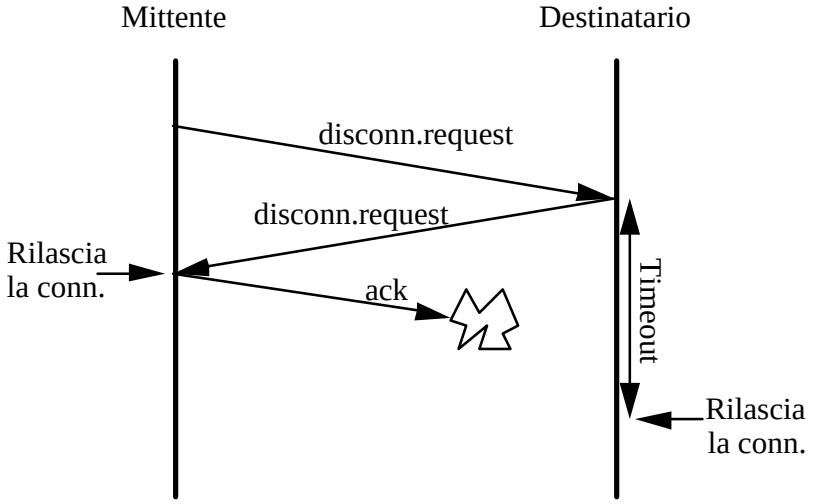

Figura 6-8: Perdita dell'ack durante il rilascio della connessione

Se invece si perde la risposta alla prima proposta di chiusura (supponendo che il timeout del destinatario sia molto più ampio di quello del mittente):

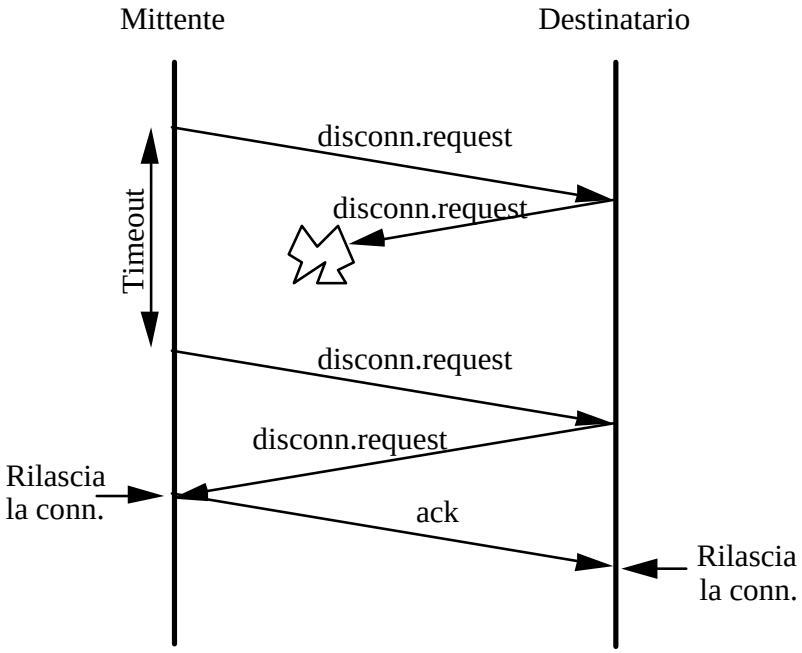

Figura 6-9: Perdita della risposta alla proposta di chiusura durante il rilascio della connessione

Infine, si può perdere la risposta alla prima proposta di chiusura e tutte le successive proposte di chiusura:

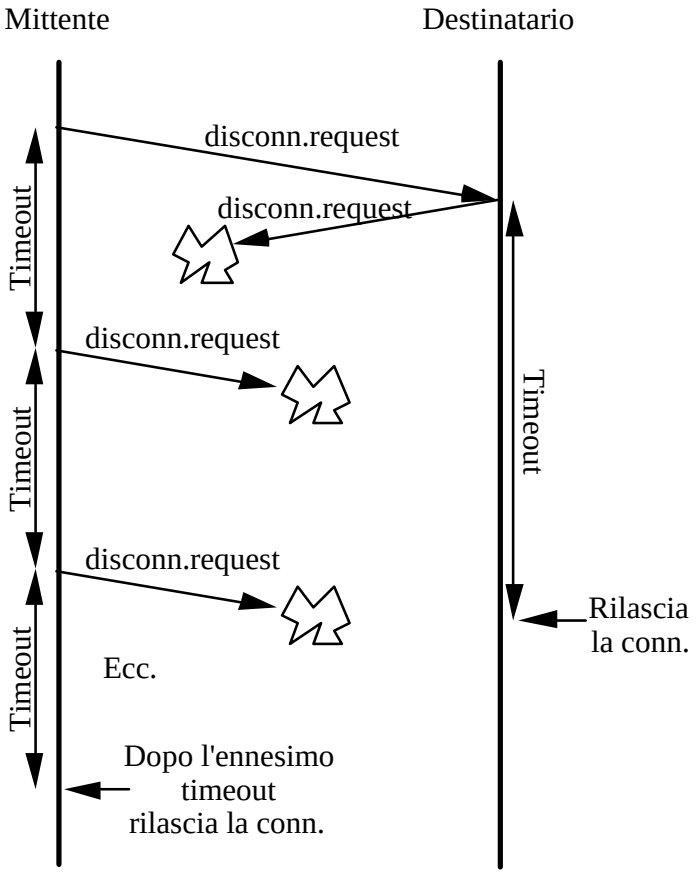

Figura 6-10: Perdita di tutti i messaggi tranne il primo

Il protocollo fallisce se tutte le trasmissioni di `disconn.request` della peer entity che inizia la procedura si perdono: in tal caso essa rilascia la connessione, ma l'altra entity non se ne accorge e la tiene aperta. Il risultato è una ***half-open connection***.

Un modo per risolvere il problema è che le parti rilascino una connessione quando passa un certo lasso di tempo (ad es. 60 secondi) senza che arrivino dati. Naturalmente, in tal caso, i processi devono scambiarsi dei ***dummy TPDU***, cioé dei TPDU vuoti, con una certa frequenza anche se non c'è necessità di comunicare alcunché, per evitare che la connessione cada.

6.5) Controllo di flusso e buffering

Per alcuni aspetti il controllo di flusso a livello transport è simile a quello di livello data link: è necessario un meccanismo, quale la gestione di una sliding window, per evitare che il ricevente sia sommerso di TPDU.

Però ci sono anche importanti differenze:
- il livello data link non ha alcun servizio, tranne la trasmissione fisica, a cui appoggiarsi. Il livello transport invece usa i servizi del livello network, che possono essere affidabili o no, e quindi può organizzarsi di conseguenza;
- il numero di connessioni data link è relativamente piccolo (uno per linea fisica) e stabile nel tempo, mentre le connessioni transport sono potenzialmente molte e di numero ampiamente variabile nel tempo;
- le dimensioni dei frame sono piuttosto stabili, quelle dei TPDU sono molto più variabili.

Se il livello transport dispone solo di servizi di livello network di tipo datagram:
- la transport entity sorgente deve mantenere in un buffer i TPDU spediti finché non sono confermati, come si fa nel livello data link;
- la transport entity di destinazione può anche non mantenere dei buffer specifici per ogni connessione, ma solo un pool globale: quando arriva un TPDU, se non c'è spazio nel pool, non lo accetta. Poiché il mittente sa che la subnet può perdere dei pacchetti, riproverà a trasmettere il TPDU finché non arriva la conferma. Al peggio, si perde un pò di efficienza.

Viceversa, ove siano disponibili servizi network di tipo affidabile ci possono essere altre possibilità per il livello transport:
- se il mittente sa che il ricevente ha sempre spazio disponibile nei buffer (e quindi è sempre in grado di accettare i dati che gli arrivano) può evitare di mantenere lui dei buffer, dato che ciò che spedisce:
    - arriva sicuramente a destinazione;
    - viene sempre accettato dal destinatario;
- se non c'è questa garanzia, il mittente deve comunque mantenere i buffer, dato che il destinatario riceverà sicuramente i TPDU spediti, ma potrebbe non essere in grado di accettarli.

Un problema legato al buffering è come gestirlo, vista la grande variabilità di dimensione dei TPDU:
- un pool di buffer tutti uguali: poco adatto, dato che comporta uno spreco di spazio per i TPDU di piccole dimensioni e complicazioni per quelli grandi;
- un pool di buffer di dimensioni variabili: più complicato da gestire ma efficace;

- un singolo array (piuttosto grande) per ogni connessione, gestito circolarmente: adatto per connessioni gravose, comporta spreco di spazio per quelle con poco scambio di dati.

In generale, data la grande variabilità di condizioni che si possono verificare, conviene adottare regole diverse di caso in caso:

- traffico bursty ma poco gravoso (ad esempio in una sessione di emulazione di terminale, che genera solo caratteri): niente buffer a destinazione, dove i TPDU si possono acquisire senza problemi man mano che arrivano. Di conseguenza, bastano buffer alla sorgente;
- traffico gravoso (ad esempio file transfer): buffer cospicui a destinazione, in modo da poter sempre accettare ciò che arriva.

Per gestire al meglio queste situazioni, di norma i protocolli di livello transport consentono alle peer entity di mettersi d'accordo in anticipo (al setup della connessione) sui meccanismi di bufferizzazione da usare.

Inoltre, spesso la gestione della dimensione delle finestre scorrevoli è disaccoppiata dalla gestione degli ack (vedremo il caso di TCP).

6.6) Multiplexing

Una importante opportunità è data dal multiplexing delle conversazioni di livello transport su quelle di livello network.

Se le connessioni di livello network (in una rete che le offre) sono costose da istituire, allora è conveniente convogliare molte conversazioni transport su un'unica connessione network (*upward multiplexing*).

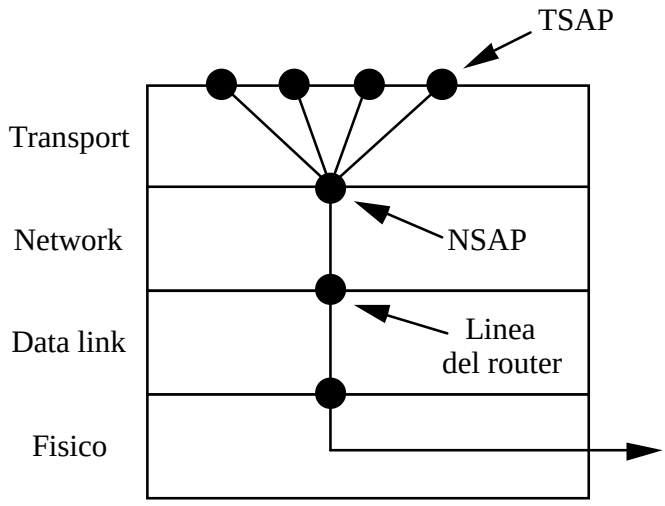

Figura 6-11: Upward multiplexing

Viceversa, se si vuole ottenere una banda superiore a quella consentita a una singola connessione network, allora si può guadagnare banda ripartendo la conversazione transport su più connessioni network (***downward multiplexing***).

Ciò può essere utile, ad sempio, in una situazione dove:
- la subnet applica a livello network un controllo di flusso sliding window, con numeri di sequenza a 8 bit;
- il canale fisico è satellitare ed ha 540 msec di round-trip delay.

Supponendo che la dimensione dei pacchetti sia di 128 byte, il mittente può spedire al massimo 2^8 - 1 pacchetti ogni 540 msec, e quindi ha a disposizione una banda di 484 kbps, anche se la banda del canale satellitare è tipicamente 100 volte più grande.

Se la conversazione transport è suddivisa su, ad esempio, 10 connessioni network, essa potrà disporre di una banda 10 volte maggiore.

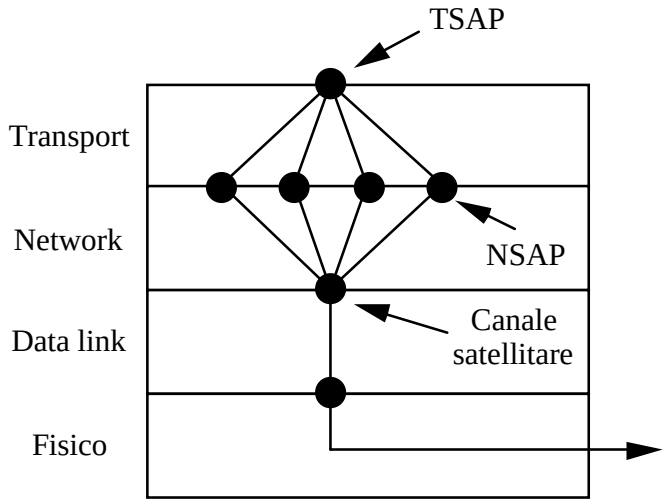

Figura 6-12: Downward multiplexing

6.7) Il livello transport in Internet

Il livello transport di Internet è basato su due protocolli:
- ***TCP*** (***Transmission Control Protocol***) RFC 793, 1122 e 1323;
- ***UDP*** (***User Data Protocol***) RFC 768.

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

Il protocollo TCP è stato progettato per fornire un flusso di byte affidabile, da sorgente a destinazione, su una rete non affidabile.

Dunque, offre un servizio reliable e connection oriented, e si occupa di:
- accettare dati dal livello application;
- spezzarli in ***segment***, il nome usato per i TPDU (dimensione massima 64 Kbyte, tipicamente circa 1.500 byte);
- consegnarli al livello network, eventualmente ritrasmettendoli;
- ricevere segmenti dal livello network;
- rimetterli in ordine, eliminando buchi e doppioni;
- consegnare i dati, in ordine, al livello application.

E' un servizio full-duplex con gestione di ack e controllo del flusso.

6.7.1) Indirizzamento

I servizi di TCP si ottengono creando connessione di livello transport identificata da una coppia di punti d'accesso detti ***socket***. Ogni socket ha un ***socket number*** che consiste della coppia:

IP address: Port number

Il socket number costituisce il TSAP.

I port number hanno 16 bit. Quelli minori di 256 sono i cosidetti ***well-known port***, riservati per i servizi standard. Ad esempio:

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

Poiché le connessioni TCP, che sono full duplex e point to point, 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.

Le connessioni TCP trasportano un flusso di byte, non di messaggi: i confini fra messaggi non sono né definiti né preservati. Ad esempio, se il processo mittente (di livello application) invia 4 blocchi di 512 byte, 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.

C'è comunque la possibilità, per il livello application, di forzare l'invio immediato di dati; ciò causa l'invio di un flag ***urgent*** che, quando arriva dall'altra parte, fa sì che l'applicazione venga interrotta e si dedichi a esaminare i dati urgenti (questo succede quando, ad esempio, l'utente durante una sessione di emulazione di terminale digita il comando ABORT (CTRL-C) della computazione corrente).

6.7.2) Il protocollo TCP

Le caratteristiche più importanti sono le seguenti:

- ogni byte del flusso TCP è numerato con un numero d'ordine a 32 bit, usato sia per il controllo di flusso che per la gestione degli ack;
- un segmento TCP non può superare i 65.535 byte;
- un segmento TCP è formato da:
    - uno header, a sua volta costituito da:
        - una parte fissa di 20 byte;
        - una parte opzionale;
    - i dati da trasportare;
- TCP usa un meccanismo di sliding window di tipo go-back-n con timeout. Se questo scade, il segmento si ritrasmette. Si noti che le dimensioni della finestra scorrevole e i valori degli ack sono espressi in numero di byte, non in numero di segmenti.
- Ogni ack è ***cumulativo***, ossia conferma l'intera sequenza di byte trasmessi, dal primo fino a quello di indice pari al valore dell'ack - 1.

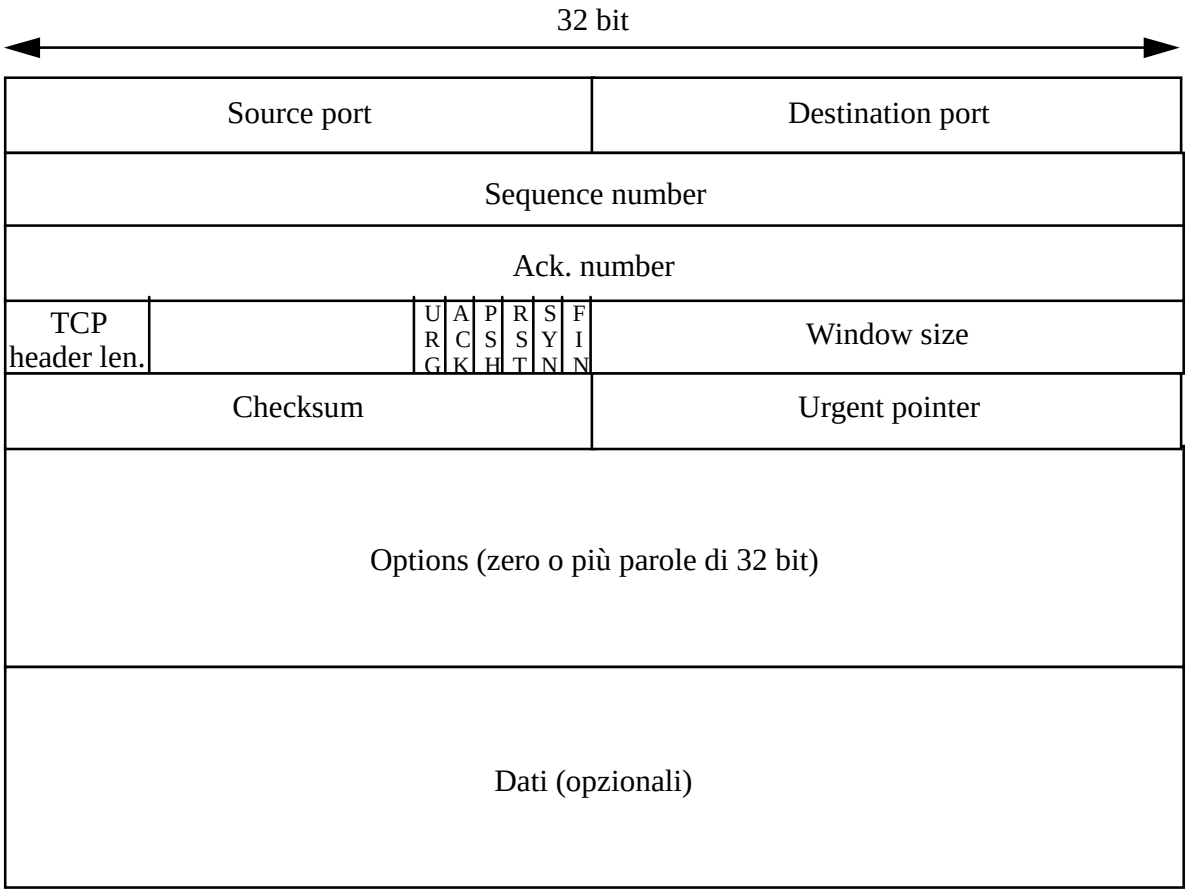

Figura 6-13: Formato del segmento TCP

I campi dell'header hanno le seguenti funzioni:

Source port, destination port	identificano gli end point (locali ai due host) della connessione. Essi, assieme ai corrispondenti numeri IP, formano i due TSAP.
Sequence number	il numero d'ordine del primo byte contenuto nel campo dati.
Ack. number	il numero d'ordine del prossimo byte aspettato.
TCP header length	quante parole di 32 bit ci sono nell'header (necessario perché il campo options è di dimensione variabile).
URG	1 se urgent pointer è usato, 0 altrimenti.
ACK	1 se l'ack number è valido (cioè se si convoglia un ack), 0 altrimenti.
PSH	dati urgenti (***pushed data***), da consegnare senza aspettare che il buffer si riempia.
RST	richiesta di reset della connessione (ci sono problemi!).
SYN	usato nella fase di setup della connessione: • SYN=1 ACK=0 richiesta connessione;

	• SYN=1 ACK=1 accettata connessione.
FIN	usato per rilasciare una connessione.
Window size	il controllo di flusso è di tipo sliding window di dimensione variabile. Window size dice quanti byte possono essere spediti a partire da quello (compreso) che viene confermato con l'ack number. Un valore zero significa: fermati per un pò, riprenderai quando ti arriverà un uguale ack number con un valore di window size diverso da zero.
Checksum	simile a quello di IP; il calcolo però si effettua sull'intero segmento TCP, non soltanto sull'header, ed include uno pseudoheader.
Urgent pointer	puntatore ai dati urgenti.
Options	fra le più importanti, negoziabili al setup: • dimensione massima dei segmenti da spedire; • uso di selective repeat invece che go-back-n; • uso di NAK.

Nel calcolo del checksum si considera l'intero segmento TCP, dato che il protocollo fornisce un servizio affidabile, ed entra in gioco anche uno ***pseudoheader*** , in aperta violazione della gerarchia, dato che il livello TCP in questo calcolo opera su indirizzi IP.

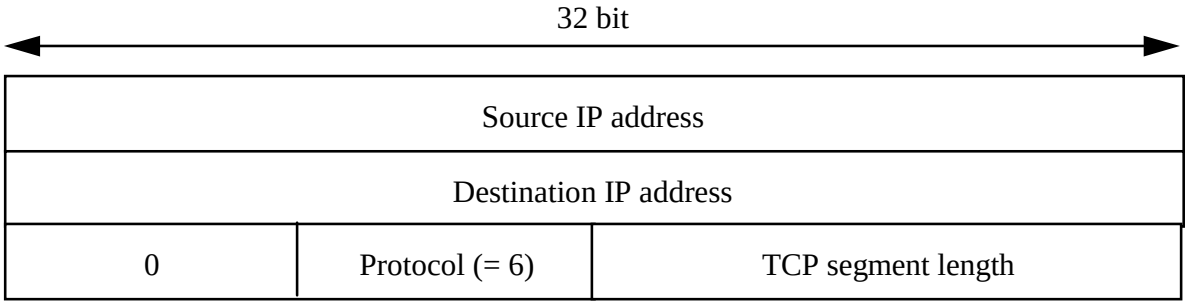

Figura 6-14: Formato dello pseudoheader TCP

Lo pseudoheader non viene trasmesso, ma precede 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.

6.7.3) Attivazione della connessione

Si usa il three-way handshake visto precedentemente:

- una delle due parti (diciamo 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;
- l'altra parte (diciamo un 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'entity 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'entity invia un segmento di conferma, con entrambi i bit `syn` ed `ack` ad uno, secondo lo schema sotto riportato.

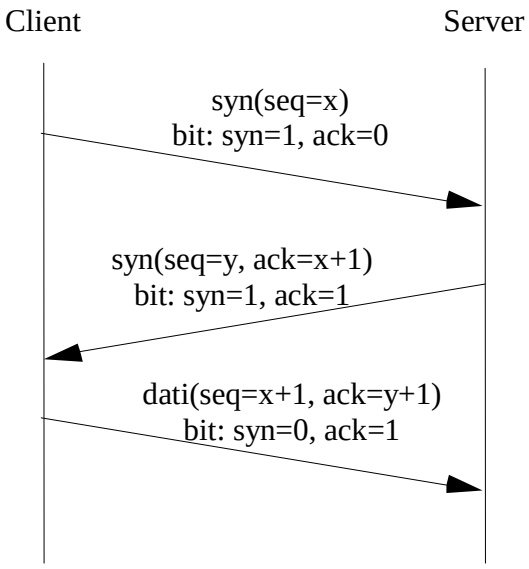

Figura 6-15: Attivazione di una connessione TCP

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. Quindi, prima che il valore si ripeta passano 2^{32}*4 microsecondi, cioè 16 miliardi di microsecondi (16000 secondi) ossia quasi 4 ore e mezza.

6.7.4) Rilascio della connessione

Il rilascio della connessione 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 `fin`;
- quando esso viene confermato, la connessione in uscita viene rilasciata;
- quando anche l'altra parte completa lo stesso procedimento e rilascia la connessione nell'altra direzione, la connessione full-duplex termina.

Per evitare il problema dei 3 eserciti si usano i timer, impostati al doppio della vita massima di un pacchetto.

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.

6.7.5) Politica di trasmissione

L'idea di fondo è la seguente: la dimensione delle finestre scorrevoli non è strettamente legata agli ack (come invece di solito avviene), 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.

Nell'esempio che segue, le peer entity si sono preventivamente accordate su un buffer di 4K a destinazione.

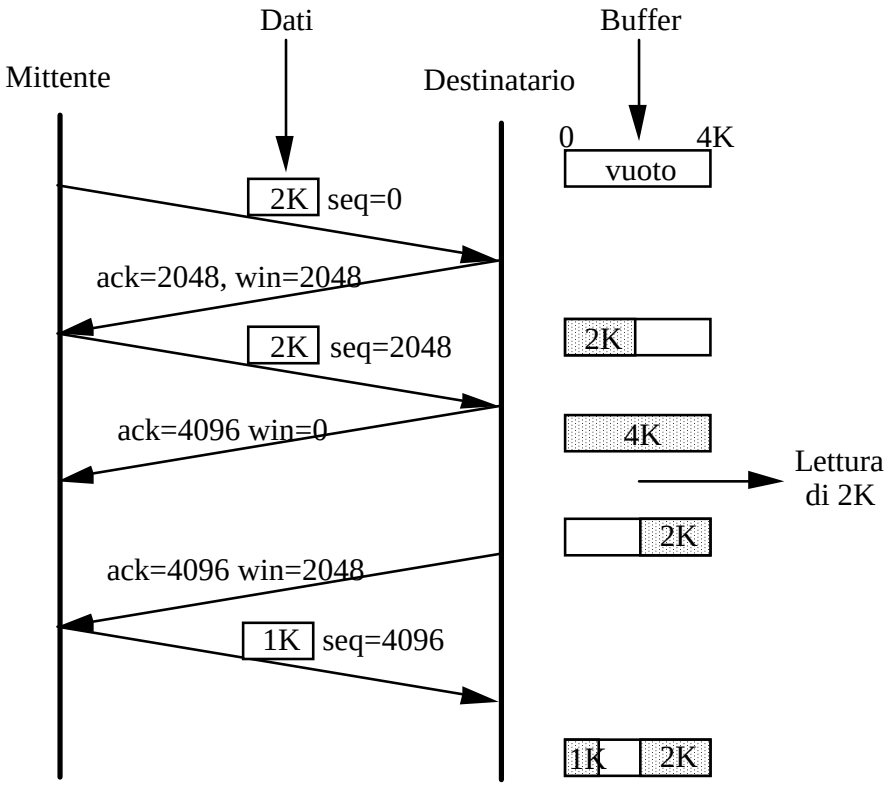

Figura 6-16: Esempio di controllo del flusso TCP

Anche se riceve win=0, il mittente può comunque inviare:
- dati urgenti;
- richieste di reinvio dell'ultimo ack spedito (per evitare il deadlock se esso si è perso).

6.7.6) Controllo congestione

Il protocollo TCP assume che, se gli ack non tornano in tempo, ciò sia dovuto a congestione della subnet piuttosto che a errori di trasmissione (dato che le moderne linee di trasmissione sono molto affidabili).

Dunque, TCP è preparato ad affrontare due tipi di problemi:
- scarsità di buffer a destinazione;
- congestione della subnet.

Ciascuno dei problemi viene gestito da una specifica finestra mantenuta dal mittente:
- la finestra del buffer del ricevitore (quella di cui all'esempio precedente);
- la ***congestion window***, che rappresenta quanto si può spedire senza causare congestione.

Il mittente si regola sulla più piccola delle due.

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 la finestra si raddoppia, fino a un valore ***threshold***, inizialmente pari a 64 Kbyte, dopodiché aumenta linearmente di 1 segmento alla volta;
- quando si verifica un timeout per un segmento:
    - il valore di threshold viene impostato alla metà della dimensione della congestion window;
    - la dimensione della congestion window viene impostata alla dimensione del massimo segmento usato nella connessione.

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

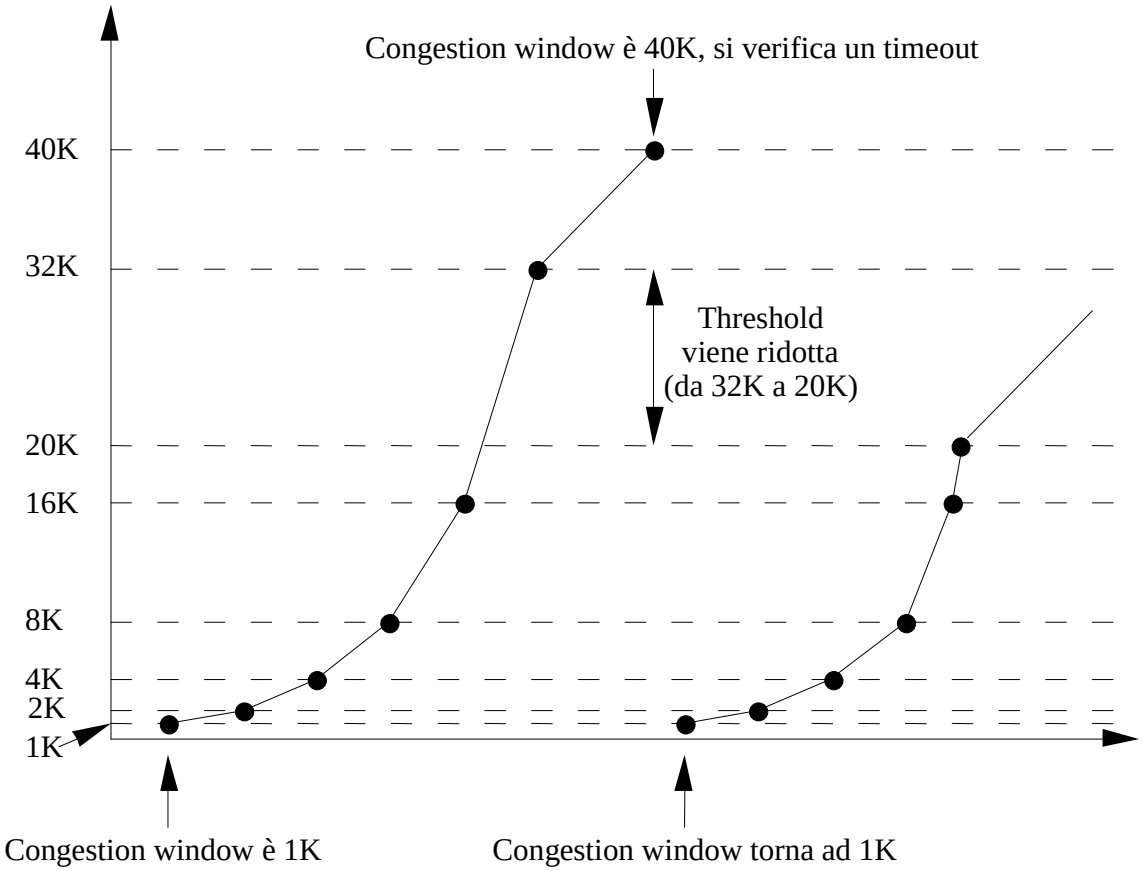

Figura 6-17: Esempio di controllo della congestione TCP

6.7.8) Ulteriori accorgimenti incorporati in TCP

Il protocollo TCP incorpora alcuni ulteriori accorgimenti per migliorare le prestazioni complessive:

- ***Delayed piggybacking***: il destinatario può ritardare l'invio di un ack, al fine di sfruttare il piggybacking, fino a un massimo di 200 msec; ciò però è consentito solo per gli ack di segmenti giunti nel giusto ordine, mentre un ack relativo a un segmento fuori ordine deve essere inviato immediatamente (naturalmente si tratterà di un ack col valore dell'ultimo byte ricevuto nell'ordine corretto);
- ***Fast retransmit***: il mittente, se riceve per la terza volta un ack con identico valore, passa immediatamente a trasmettere il segmento successivo a quello confermato dai tre ack identici, senza attendere che il relativo time-out si verifichi.

6.7.7) Il protocollo UDP

Il livello transport fornisce anche un protocollo non connesso e non affidabile, utile per inviare dati senza stabilire connessioni (ad esempio per applicazioni client-server).

Lo header di un segmento UDP è molto semplice:

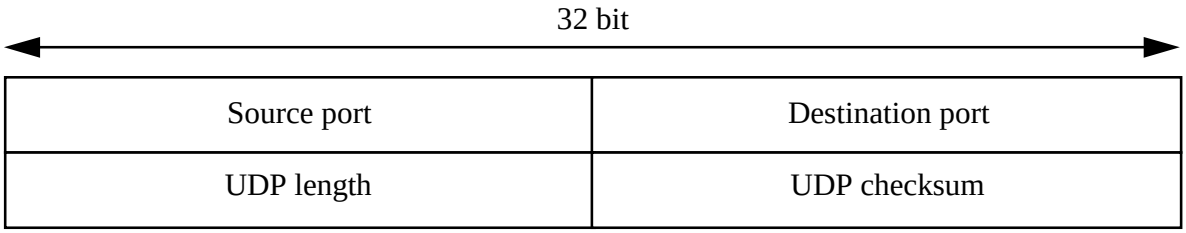

Figura 6-18: Formato dello header UDP

La funzione di calcolo del 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.