Il livello trasporto ed il protocollo TCP
Indice

- **servizi del livello trasporto**
- **multiplexing/demultiplexing**
- **trasporto senza connesione: UDP**
- **principi del trasferimento dati affidabile**
- **trasporto orientato alla connessione: TCP**
  - **trasporto affidabile**
  - **controllo del flusso**
  - **gestione della connessione**
- **principi di controllo della congestione**
- **controllo della congestione TCP**

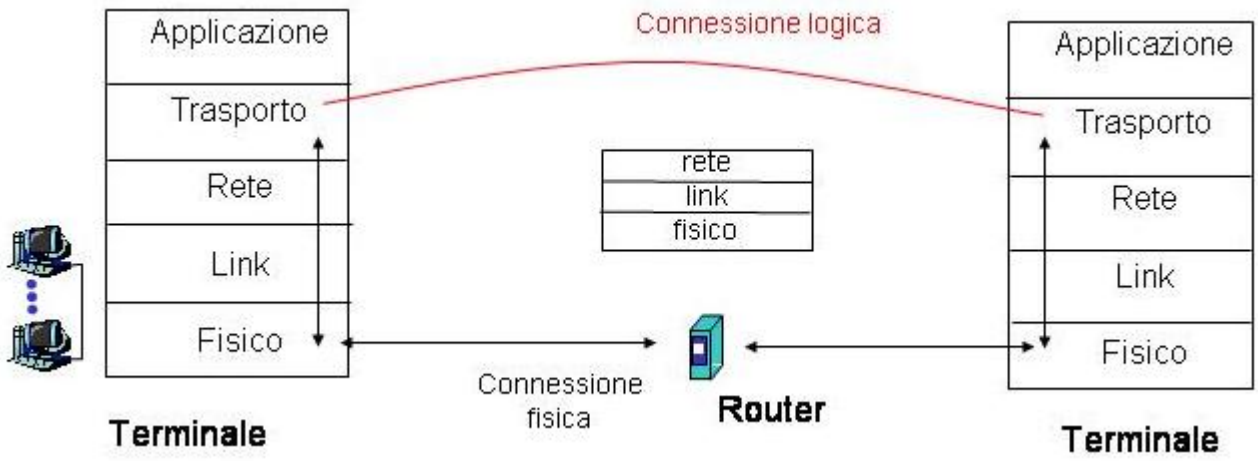
Cosa è il livello trasporto

Il *livello trasporto* fornisce una *comunicazione logica fra processi applicativi* che girano su terminali diversi. I processi del livello applicazione per spedire dei dati li passano al livello trasporto.

I protocolli di trasporto girano solo nei terminali della rete (*end systems*) e non nei *router*.

Confronto servizi trasporto/rete:

- livello rete: realizza il trasferimento dati fra terminali usando le funzionalità del livello link
- livello trasporto: realizza il trasferimento dati fra processi: si basa sui (ed estende i) servizi di livello rete.



Servizi e protocolli del livello trasporto

Servizi di trasporto in *Internet*: il livello trasporto realizza due diversi tipi di servizio e quindi prevede due protocolli principali:

- Servizio orientato alla connessione:
  - realizza il trasferimento di dati fra *end-systems* (terminali).
  - prevede un fase iniziale di *handshaking* nella quale viene preparato lo stato dei due terminali che devono comunicare
  - TCP (*Transmission Control Protocol*) è il protocollo che realizza il servizio orientato alla connessione di Internet
- Servizio senza connessione:

  - realizza il trasferimento di dati fra *end-systems* (terminali).
  - trasferimento dati inaffidabile
  - no controllo flusso
  - no controllo congestione
  - **UDP** (*User Datagram Protocol*) realizza il servizio senza connessione di Internet
- Non sono disponibili al livello trasporto i seguenti servizi:
  - *real-time*
  - banda minima garantita
  - multicast affidabile

Segmento del livello trasporto

Segmento: l'unità di dati trasferita fra entità del livello trasporto è detta segmento (o anche TPDU *transport protocol data unit*)

In prima approssimazione il segmento del livello trasporto (sia *TCP* che **UDP**) ha il formato illustrato qui sotto

I numeri di porta (sorgente e destinazione) servono per il *multiplexing* ed il *demultiplexing*

Multiplexing/demultiplexing 1

Il *Multiplexing-demultiplexing* permette una comunicazione fra processi di tipo molti a molti, ovvero processi diversi che girano su uno stesso *host* (con un unico numero di *IP*) possono inviare messaggi a processi diversi che girano su un altro *host* (con un altro numero di *IP*). La corrispondenza fra processo mittente e processo destinatario è possibile grazie ai numeri di porta che identificano i vari processi.

Multiplexing: raccolta di dati da più programmi applicativi e aggiunta ad essi di opportune intestazioni (che serviranno dopo per identificare il destinatario nella fase di *demultiplexing*). Basato su:

- indirizzo *IP* (presente a livello di pacchetti *IP*);
- numeri porta sorgente e porta destinazione presenti in ogni segmento.

Demultiplexing: consegna dei segmenti ricevuti ai corretti processi destinatari del livello applicazione.

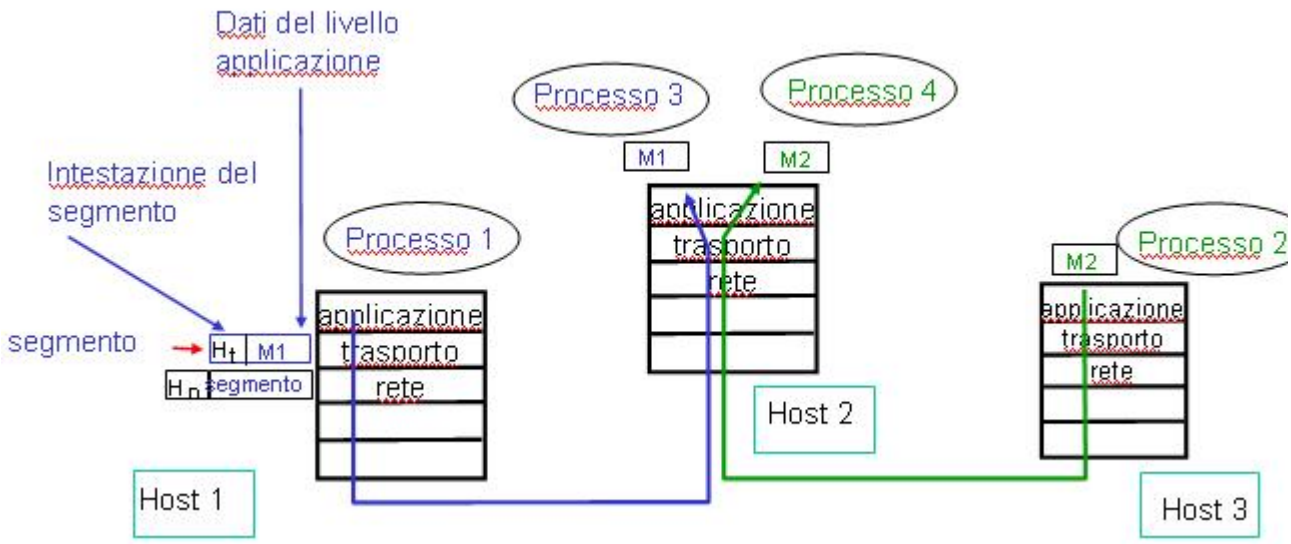
Nota: alcuni numeri di porta sono riservati per applicazioni specifiche quali, ad esempio, posta elettronica (protocollo SMTP), trasferimento *file* (FTP), *Web* (HTTP).

Multiplexing/demultiplexing 2

La spedizione di una messaggio comporta le seguenti fasi (vedi figura nella **pagina successiva**):

- Nell'host mittente il messaggio M viene spedito dal livello applicazione al livello rete:il messaggio:
  - viene passato al livello trasporto e qui diviso in tanti segmenti;
  - ad ogni segmento sono aggiunte le intestazioni (numeri di porta) necessari per il *multiplexing*;
  - il segmento viene passato al livello rete;
  - il livello rete del mittente scompone ogni segmento in pacchetti.
- La rete quindi fà transitare i pacchetti dall'host sorgente all'host destinazione (usando l'indirizzo *IP* del destinatario).
- Nell'host destinatario il messaggio M viene spedito dal livello rete al livello applicazione:
  - il livello rete del destinatario ricompone i pacchetti per ricostruire i segmenti;
  - da ogni segmento viene estratta la parte dati e inviata al processo opportuno del livello applicazione;
  - il processo è identificato dal numero di porta destinazione presente nell'intestazione del segmento.
  - Al livello applicazione il messaggio è ricostruito mediante i segmenti che lo compongono.

Multiplexing/demultiplexing 3



UDP: User Datagram Protocol [RFC 768]

UDP è un protocollo del livello trasporto *Internet* molto semplice.

Offre un servizio *best effort* che in pratica non garantisce nulla: i segmenti **UDP** possono essere:

- persi;
- consegnati in ordine errato alle applicazioni.

Offre un servizio senza connessione:

- non c'è *handshacking* iniziale fra il mittente (*sender*) **UDP** *sender* ed il ricevente (*receiver*), ovvero non esiste una fase iniziale nella quale il mittente ed il ricevente stabiliscono una connessione.
- Ogni segmento **UDP** è gestito indipendentemente dagli altri.

Non c'è controllo della congestione e non c'è controllo del flusso: i dati possono essere spediti dal *sender* alla velocità che vuole, senza tenere conto dello stato della rete e senza che il *receiver* possa rallentare la velocità di spedizione.

Trasmissione sia *unicast* che *multicast*.

Usare UDP o TCP?

Perchè usare **UDP**?

- Non viene stabilita una connessione (che può aggiungere fattori di ritardo);
- è semplice: non c'è stato della connessione;
- piccola intestazione di ogni segmento;
- nessun controllo della congestione: **UDP** può mandare i segmenti alla velocità che vuole;
- quindi maggiore efficienza (minore ritardo e maggiore banda) al prezzo di mancanza di affidabilità.

UDP è quindi usato da quelle applicazioni che possono tollerare perdita di dati ma che hanno bisogno della massima banda possibile e di ritardi più bassi possibile:

- tipiche applicazioni che hanno queste caratteristiche e usano **UDP** sono le seguenti: applicazioni multimediali, teleconferenze, giochi interattivi telefonia via *Internet*. Da notare che il DNS (servizio del livello applicazione per la traduzione dei nomi in indirizzi *IP*) usa **UDP**.

Viceversa, *TCP* è usato da quelle applicazioni per le quali la banda ed i ritardi non sono essenziali, mentre è importante l'affidabilità della trasmissione:

- Applicazioni che hanno queste caratteristiche e usano *TCP*: HTTP (WWW), FTP (*file transfer*), Telnet (*remote login*), SMTP (*email*).

Requisiti dei servizi di trasporto per alcune applicazioni comuni

Applicazione	**Perdita di dati**	**Banda elastica**	**Rilevanza ritardo**
Trasferimento *file*	no	si	no
Posta elettronica	no	si	no
Documenti *Web*	si	si	no
audio/video *real time*	si	no (audio: 5kb-1Mb, *video* 10Kb-5Mb)	si 100msec.
audio/*video* no *real time*	si	no (audio: 5kb-1Mb, *video* 10Kb-5Mb)	si pochi sec.
giochi interattivi	si	> alcuni Kbps	si 100msec.
applicazioni finanziarie	no	elastica	si e no

Perdita dati: no -> non tollerata; si -> tollerabile.

Banda elastica: si -> l'applicazione funziona con qualsiasi banda; no -> è indicata la banda minima necessaria per il funzionamento.

Rilevanza ritardo: no -> l'applicazione funziona con qualsiasi ritardo; si -> è indicato il ritardo massimo che permette il funzionamento corretto dell'applicazione.

Applicazioni di Internet ed i relativi protocolli

Applicazione	**Protocollo del livello applicazione**	**Protocollo del livello trasporto usato**
Posta Elettronica	smtp [RFC 821]	TCP
Accesso terminale remoto	telnet [RFC 854]	TCP
Web	HTTP [RFC 2068]	TCP
Trasferimento *file*	FTP [RFC 959]	TCP
Streaming multimedia	proprietario	tipicamente **UDP**
Telefono via Internet	proprietario	tipicamente **UDP**

Formato del segmento UDP

UDP è usato tipicamente per applicazioni multimediali che possono tollerare la perdita di dati ma che invece sono sensibili ai ritardi ed alla banda di trasmissione. Il segmento **UDP**, dato che non deve memorizzare informazioni necessarie per realizzare meccanismi di trasporto affidabile, risulta particolarmente snello: infatti contiene soltanto i seguenti campi

- dati: questo campo contiene i dati inviati dall'applicazione;
- porta mittente e porta destinatario: questi due campi, come visto, contengono i numeri di porta (mittente e destinazione) e servono per il *multiplexing*;
- lunghezza: indica la lunghezza in *bytes* del segmento inclusa l'intestazione;
- *checksum*: permette un controllo (parziale) sulla correttezza dei dati trasmessi.

Trasferimento affidabile su **UDP**: è possibile soltanto gestendo al livello applicazione i meccanismi che controllano l'affidabilità della trasmissione: deve quindi essere realizzato un meccanismo di gestione dell'errore specifico per ogni applicazione.

UDP checksum

A che serve: il campo *checksum* contiene un valore di controllo che serve per il rilevamento di eventuali errori nel segmento. Tale controllo avviene secondo le seguenti modalità che coinvolgono sia il mittente che il destinatario del segmento.

- *Mittente*: il mittente considera il contenuto del segmento come una sequenza di interi a 16-*bit* esegue le seguenti operazioni:
  - esegue la somma del contenuto del segmento (visto come sequenza di interi);
  - esegue il complemento a 1 della somma calcolata (il complemento a 1 si ottiene cambiando gli 0 in 1 e gli 1 in 0) il valore così ottenuto è messo nel campo *checksum*.
- *Destinatario*: il destinatario controlla la correttezza del segmento ricevuto mediante le seguenti operazioni:
  - calcola la somma del contenuto del segmento ricevuto (sempre visto come sequenza di interi);
  - somma la somma calcolata al valore contenuta nel campo *checksum*:
    - risultato diverso da sequenza di 1 ---> rilevamento errore (infatti questo significa che la somma fatta dal mittente è diversa dallasomma fatta dal destinatario);
    - risultato costituito da sequenza di 1 ---> nessun errore rilevato.

N.B. Il fatto che non si siano rilevati errori non garantisce l'assenza di errori: la *checksum* permette sono un controllo di correttezza parziale.

Servizio orientato alla connessione: TCP

TCP offre le seguenti funzionalità [RFC 793]:

- trasmissione *unicast* (ovvero punto a punto);
- trasferimento di *stream* di *byte* affidabile e ordinato: è garantito che i dati spediti arrivino tutti a destinazione e nell'ordine in cui sono spediti. Vengono usati meccanismi di riscontro (*ack*) e di *time-out* per controllare la perdita di dati;
- controllo della congestione: sono implementati opportuni meccanismi che evitano il sovraccarico (congestione) della rete.: se la rete è congestionata il *sender* rallenta la velocità di spedizione dei segmenti;
- controllo del flusso: il destinatario può obbligare il mittente a non superare una *data* velocità (variabile dinamicamente) di trasmissione per evitare la saturazione del *buffer* di ricezione;
- *setup* della connessione: prima della trasmissione dei dati c'è una fase di inizializzazione della connessione.

TCP in prima approssimazione fà le seguenti cose:

- prende dati dal livello applicazione;
- aggiunge informazioni per indirizzamento e affidabilità;
- forma dei segmenti;
- spedisce segmenti ad un pari su di un altro terminale;
- aspetta la ricevuta (*ack*).

Principi del trasferimento dati affidabile

Il livello rete offre una trasferimento dati che non garantisce in alcun modo l'affidabilità. Il livello trasporto deve quindi implementare dei meccanismi che controllino la ricezione corretta dei segmenti spediti.

Le caratteristiche del canale non affidabile determineranno la complessità del protocollo per il

trasferimento affidabile (rdt).

Assumendo che il canale possa:

- perdere dati;
- introdurre errori;
- alterare l'ordine di arrivo dei dati rispetto a quello di spedizione.

L'idea di base del trasferimento affidabile è comunque la seguente per tutti i protocolli: il mittente per ogni segmento spedito si attende che il destinatario gli invii un messaggio di riscontro (detto *ack*) che attesti la corretta ricezione dei dati (l'analogo della ricevuta di ritorno delle raccomandate). Se tale riscontro non arriva entro un dato tempo limite, il mittente assume che i dati siano andati perduti e quindi li rispedisce. Inoltre il mittente rispedisce i dati anche nel caso in cui il destinatario gli abbia mandato un riscontro negativo nel quale lo avverte che i dati sono arrivati con degli errori (questo è controllabile usando la *checksum*, come visto). L'identificazione dei singoli segmenti avviene usando opportuni numeri di sequenza associati ai segmenti. Si deve cercare di evitare che due pacchetti siano identificati dallo stesso numero di sequenza.

Tipi di trasferimento affidabile

Si possono individuare due categorie di protocolli per il trasferimento affidabile:

- protocolli *stop and wait*: per ogni segmento spedito si aspetta l'*ack* prima di inviare il segmento successivo. L'efficienza dei protocolli *stop and wait* è molto scarsa perchè il canale non viene utilizzato al meglio: dopo aver spedito un segmento per tutto il tempo necessario a questo per raggiungere la destinazione e all'*ack* per tornare indietro nessun altro dato è spedito sul canale che quindi è inutilizzato per la maggior parte del tempo.
- Protocolli *pipeline*: vengono inviati più segmenti senza aspettare l'*ack* dei precedenti. Tali segmenti poi possono poi essere riscontrati individualmente o in modo cumulativo. I protocolli reali del livello trasporto sono di tipo pipeline. Si distinguono due modelli di protocolli pipeline:
  - *GO-BACK* n: i segmenti sono riscontrati in modo cumulativo.
  - Ripetizione Selettiva: i segmenti sono riscontrati singolarmente.

TCP usa un protocollo di tipo *pipeline* con riscontri gestiti mediante una combinazione delle tecniche go-*back* n e ripetizione selettiva

TCP: Introduzione

TCP è descritto in dettaglio nelle RFC 793, 1122, 1323, 2018, 2581. Come accennato in precedenza le principali caratteristiche di *TCP* sono le seguenti:

- protocollo punto a punto ovvero si ha un solo mittente ed un solo destinatario *receiver*;
- trasferimento affidabile, ordinato di *byte steam*: non vi sono confini dei messaggi: il livello applicazione dopo aver aperto una connessione *TCP* vede l'invio e l'arrivo di un flusso (*stream*) di *byte*;
- *pipeline*: *TCP* è di tipo *pipeline*. Il numero di segmenti che possono essere spediti senza che vi sia riscontro è modificabile dinamicamente dai meccanismi di controllo del flusso e della congestione;
- *send & receive buffers*: sono presenti dei *buffer* sia nel mittente che nel destinatario;
- comunicazione *full-duplex*: nella stessa connessione si ha un flusso di dati in entrambe le direzioni;
- orientato alla connessione: all'inizio della connessione vengono scambiati dei messaggi di controllo fra mittente e destinatario per inizialiizare lo stato della connessione (*handshaking*);

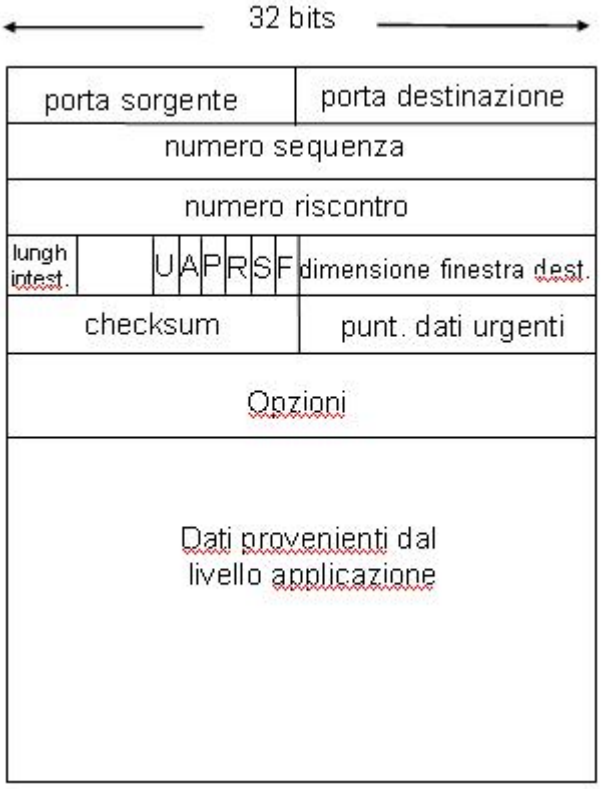
- controllo del flusso e della congestione.

TCP: spedizione dei dati

I dati sono passati dal livello applicazione al livello trasporto attraverso i *socket*.

Il *socket* (introdotto in *Unix*, 1981) è un'interfaccia locale fra livello applicazione e livello trasporto creata, usata e rilasciata esplicitamente dalle applicazioni e controllata dal sistema operativo. Il *socket* è associato ad un indirizzo *IP* ed ad un numero di porta che permettono di individuare il processo applicativo al quale devono arrrivarare i dati. Processi applicativi possono spedire messaggi a un processo (remoto o locale) scrivendo nel *socket* ricevere messaggi da un processo leggendo dal *socket*.

Struttura del segmento TCP



Componenti del segmento TCP

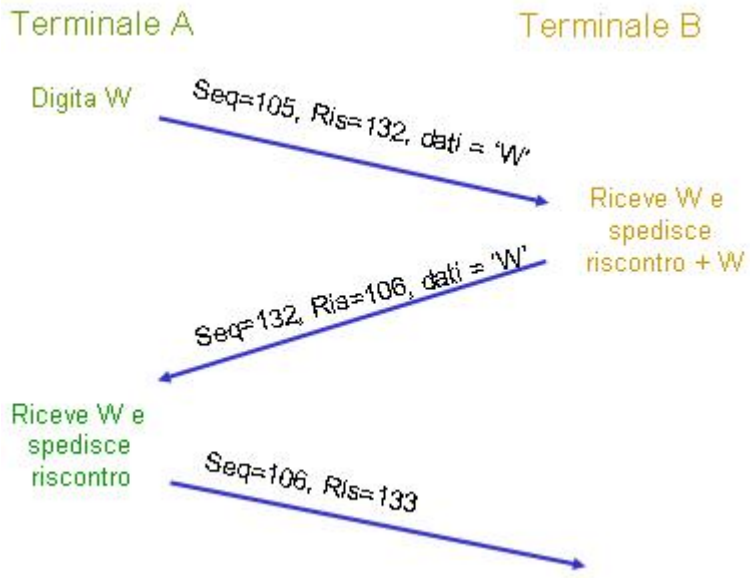
- Porta sorgente e destinazione: numeri di porta sorgente e destinazione per *multiplexing*;
- numero sequenza: numero che identifica ogni segmento e che serve per realizzare il

trasferimento affidabile. Il numero indica il numero del primo *byte* nella parte dati del segmento, dove i *byte* sono numerati a partire dall'inizio del flusso di *byte* delle connessione *TCP*. Ad esempio, se nel primo segemento sono spediti 500 *bytes* di dati, numerati da 0 a 499, il numero di sequenza del secondo segmento sarà 500;

- numero riscontro: questo èil numero che indica il segmento per il quale il destinatario ha inviato un riscontro (*ack*): il numero indica il numero del prossimo segmento che il destinatario si aspetta. Ad esempio, se un segmento con numero di sequenza 1000 contenente 450 *bytes* di dati è ricevuto, il destinatario invierà come riscontro un segmento contenente nel campo riscontro il valore 1451 (il numero del prossimo *byte* che ci aspetta);
- lungh. intestazione: lunghezza del campo intestazione;
- U (URG): *bit* che indica dati urgenti (usato raramente);
- A (ACK): *bit* che indica se si tratta di un segmento che contiene un riscontro (ACK);
- R (RST), S (SYN), F (FIN): *bit* usati nella fase di inizializzazione della connessione;
- dimensione finestra destinatario: campo (aggiornabile dinamicamente dal destinatario) che contiene la dimensione della parte libera del *buffer* destinatario e che è usata per il controllo del flusso. Il mittente controllerà che il numero di *byte* spediti e non ancora riscontrati sia inferiore al valore contenutoin questo campo;
- *checksum*: campo che contiene il valore della *checksum* per il controllo di errore;
- punt. dati urgenti: campo che contiene l'indirizzo degli eventuali dati urgenti (raramente usato).

Un esempio di uso dei numeri di sequenza: una sessione telnet

- Seq= **Numero sequenza** (numero del primo *byte* di dati nel segmento)
- Ris = Numero riscontro: (umero sequenza del prossimo *byte* che ci si aspetta dall'altro lato)
- Nell'esempio in figura, l'utente sul terminale A apre una sessione telnet. Ricordiamo che telnet è un protocollo di livello applicazione per il collegamento remoto che usa *TCP* come protocollo di livello trasporto. Dopo aver aperto la sessione telnet e quindi aver iniziato la connessione *TCP* avviene quanto segue:
  - L'utente sul terminale A digita la lettera W
  - B riceve W, spedisce il riscontro per (Ris = 106) e rispedisce anche W (per l'eco sul terminale)
  - A riceve W e rispedisce il riscontro



Trasferimento affidabile TCP: lato mittente 1

Il seguente pseudo-codice descrive il funzionamento di un *sender* che realizza il trasferimento

affidabile di *TCP* (:= indica il comando di assegnamento).

```
base_send := numero di sequenza iniziale:
prossimo_num_seq := numero di sequenza iniziale
ripeti {
se vengono ricevuti dati dall'applicazione:
```

- crea segmento con numero di sequenza prossimo_num_seq;
- fai partire il *timer* per il segmento con numero di sequenza prossimo_num_seq;
- passa il segmento al livello rete;
- prossimo_num_seq : = prossimo_num_seq + lunghezza dati spediti;

```
se si verifica un timeout per il segemento con numero di sequenza x:
```

- ristrametti il segmento con numero di sequenza x;
- calcola il nuovo valore da assegnare al *timer* del segmento x;
- fai ripartire il *timer* per il segemento con numero di sequenza x;

```
se si riceve un riscontro con valore y nel campo numeri riscontro:
se X> base_send;
cancella tutti i timer per i segmenti con numeri di sequenza < Y;
/* riscontro cumulativo*/
base_send : = y;
se X <= base_send
incrementa il numero dei ricontri duplicati ricevuti per il segmento Y;
se numero dei ricontri duplicati = 3;
ristrametti il segmento con numero di sequenza x;
fai ripartire il timer per il segmento con numero di sequenza x;
}
```

Trasferimento affidabile TCP: lato mittente 2

Come si può notare nello pseudo codice visto in precedenza in *TCP* si ha:

- **Riscontro** Cumulativo: il riscontro per un certo numero di sequenza vale anche per tutti i numeri di sequenza minori.
- *Timer*: al momento della spedizione di un segmento viene fatto partire un orologio (*timer*); quando tale orologio raggiunge una certa soglia, ovvero quando si verifica un *time*-out, se non si è ricevuto un riscontro per un segmento si assume che il segmento sia andato perduto. Il valore della soglia è modificabile dinamicamente in base a criteri statistici che tengono conto del tempo necessario ad un segmento per raggiungere il destinatario e quindi tornare indietro (detto RTT ovvero *Round Trip Time*).
- Ritrasmissione selettiva: solo un segmento viene rispedito al verificarsi di un *time out* e non tutti i segmenti che ancora non sono stati riscontrati.
- Ritrasmissione veloce: dato che *TCP* non usa riscontri negativi, l'unico modo che ha il destinatario per avvertire il mittente che un segemento non è stato ricevuto è quello di ritrasmettere un riscontro per il precedente segmento ricevuto correttamente. Quando il mittente riceve tre riscontri duplicati per un segmento rispedisce il segmento successivo (anche se il suo *timer* non è ancora arrivato alla soglia). Il mittente si accorge che un segmento non è arrivato ricevendo un segmento con numero di sequenza maggiore di quello che si sarebbe aspettato.

TCP lato destinatario

Il seguente pseudo-codice descrive il funzionamento di un destinatario che realizza il trasferimento affidabile di TCP.

```
ripeti {
Se arriva un segmento in ordine, non ci sono buchi e tutti gli altri segmenti
arrivati sono già stati riscontrati
aspetta fino a 500 millisecondi. Se non arriva nessun altro segmento spedisci un
riscontro per il segmento arrivato
Se arriva un segmento in ordine, non ci sono buchi e e c'e un riscontro sospeso
spedisci immediatamente un ack cumulativo
Se arriva un segmento non in ordine, con numero di sequenza maggiore di quello
atteso (ovvero è rilevato un buco)
spedisci un riscontro duplicato che indichi il prossimo numero di sequenze
atteso
Se arriva un segmento che riempie parzialmente o totalmente un buco
spedisci immediatamente un Ack
}
```

Controllo del flusso in TCP

Controllo del flusso: fa sì che il *buffer* del destinatario non venga saturato con i dati dal mittente. Da non confondersi con il controllo di congestione. Viene realizzato dal mittente facendo si che il numero di *byte* trasmessi e non ancora riscontrati sia minore del valore contenuto nel campo dimensione finestra destinatario dell'ultimo semento ricevuto dal mittente (valore inserito dal destinatario). Tale valore indica lo spazio libero nel *buffer* del destinatario ed è inserito nel segmento dal destinatario.

Gestione della connessione TCP

La connessione *TCP* passa attraverso tre fasi:

- inizializzazione;
- trasferimento dati;
- chiusura.

Nella fase di inizializzazione il mittente ed il destinatario *TCP* si scambiano alcuni segmenti che servono solo per stabilire alcuni valori di controllo quali:

- numero di sequenza;
- dimensioni dei *buffer*;
- dimensioni del campo dimensione finestra destinatario per il controllo del flusso.

Nella fase di trasferimento dati vengono inviati i dati ricevuti dal livello applicazione.

Nella fase di chiusura vengono scambiati dei segmenti per avvertire delle chiusura della connessione il partner.

Fase di apertura della connessione TCP (handshaking)

La connessione è iniziata da un processo applicativo che chiede al livello trasporto i tabire un collegamento *TCP* con un altro processo applicativo. Il processo che richiede la connessione è detto *client* mentre l'altro è detto *server*. La connessione è dunque iniziata dal *client* ad esempio con la seguente istruzione Java

```
Socket client = new Socket(indirizzo IP,numero porta);
```

Il *server* dovrà essere pronto ad accettare la richiesta di connessione, ad esempio usando la seguente istruzione Java

```
Socket connessione = Iniziale.accept()
```

dove Iniziale è un oggetto della classe Server**Socket**

La connessione viene stabilita seguendo i seguenti tre passi:

1. Il *client* invia una richiesta di connessione: il *client*:
    - sceglie un numero di sequenza;
    - invia al *server* un segmento (con il numero di sequenza scelto) che non contiene dati e contiene il valore 1 per il *bit* SYN (per questo tale segmento è detto segmento SYN).
2. Quando il *server* riceve la richiesta esso:
    - alloca i *buffer* necessari;
    - sceglie un proprio numero di sequenza per il primo segmento che dovrà spedire;
    - spedisce al *client* un segmento (con il numero di sequenza scelto) che contiene un riscontro per il segmento SYN ricevuto.
3. Quando il *client* riceve il segmento contenente il riscontro per il segmento SYN:
    - alloca i *buffer* necessari;
    - invia al *server* un segmento con il *bit* SYN a 0 che contiene un riscontro per il segmento ricevuto dal *server*;
    - inserisce nella parte dati i dati del livello applicazione che vuole spedire.

La connessione a questo punto è stabilita.

Chiusura della connessione TCP

La connessione può essere chiusa sia dal *client* che dal *server*. Ad esempio, il *client* la può chiudere facendo

```
client.close();
```

dove *client* è il *socket* visto in precedenza. Al verificarsi di una tale richiesta di chiusura della connessione:

1. Il *client* manda un segmento che contiene il *bit* FIN posto a 1 (segmento FIN);
2. Il *server* riceve il segmento FIN e risponde con segmento che contiene il riscontro per tale segmento FIN;

3. Il *server* invia al *client* un altro segmento che contiene il *bit* FIN a 1;
4. Il *client* riceve questo segmento FIN dal *server* e invia un segmento di 1 di riscontro per esso;
5. Il *client* entra quindi in una fase di attesa (della durata tipica di 30 secondi) durante la quale se necessario può rispedire l'ultimo segmento di riscontro.

Terminata la fase di attesa la connessione è chiusa e tutte le risorse necessarie per la comunicazione sono de-allocate.

Principi di controllo della congestione

Congestione: informalmente, si ha congestione in una rete quando ci sono troppi mittenti che spediscono più dati di quanti la rete riesca a gestirne. Gli effetti della congestione vanno dai lunghi ritardi di arrivo dei dati alla perdita di pacchetti. Difatti i *router* e i nodi intermedi della rete contengono dei *buffer* sia di ingresso che di uscita nei quali sono accodati i pacchetti in attesa di essere smistati sui collegamenti opportuni. Quando il numero di tali pacchetti aumenta oltre le capacità di gestione del *router* le *code* nei *buffer* si allungano con conseguente aumento dei ritardi e, quando si oltrepassa la capacità dei *buffer*, perdita di pacchetti.

Controllo della congestione: è uno dei problemi più rilevanti nello studio delle reti attuale. Vi sono due tecniche principali:

- Controllo assistito dalla rete: i *router* ed i nodi intermedi forniscono informazioni ai terminali sullo stato di congestione della rete e possono anche imporre ai terminali di limitare la velocità di spedizione. Questo tipo di controllo è usato nell reti ATM.
- Controllo senza assistenza della rete: in questo caso lo stato di congestione della rete è osservato dai terminali in base ai ritardi e all'eventuale perdita dei pacchetti. Non c'è alcuna informazione da parte dei *router*. Questo è l'approccio usato da *TCP*.

Controllo della congestione in TCP

In *TCP* in controllo della congestione avviene come segue: il mittente mantiene in una opportuna variabile (detta finestra di congestione) il numero di segmenti massimo che possono essere spediti senza riscontro. Il valore della finestra di congestione è aumentato dinamicamente dal mittente fino a che non si verifica una perdita di un segmento; quando si verifica una perdita si modifica il valore della finestra riportandolo ad un valore prefissato (più basso) e quindi si inizia di nuovo a spedire segmenti aumentando il valore della finestra.

Idealmente quindi *TCP* cerca di sfruttare al massimo la banda disponibile.

Diverse implementazioni di *TCP* possono usare varianti dell'algoritmo di controllo della congestione qui accennato e descritto più in dettaglio nella pagina seguente (tale algoritmo è di solito detto *Tahoe*).

Controllo della congestione in TCP: algoritmo Tahoe

Questo algoritmo usa due variabili:

- FinCong = finestra di congestione, contiene il numero massimo di segmenti che possono essere spediti senza riscontro;
- Soglia = contiene un valore che varia dinamicamente che modifica il modo in cui cresce FinCong.

e consiste delle seguenti fasi:

- Partenza Lenta: FinCong viene inzializzata a 1. Fino a quando FinCong è minore di Soglia vengono ripetute le seguenti operazioni:
    - spedisci un numero di segmenti pari a FinCong; aspetta il riscontro (cumulativo) per tali segmenti; raddoppia il valore di FinCong.
- Annullamento della congestione: quando FinCong diventa maggiore di Soglia viene ripetuto il seguente ciclo fino al momento in cui si verifica una perdita di un segmento:
    - spedisci un numero di segmenti pari a FinCong; aspetta il riscontro (cumulativo) per tali segmenti; aumenta di 1 il valore di FinCong.
- Perdita: al verificarsi di una perdita (segnalata da un *timeout* per un segmento) vengono eseguite le seguenti operazioni:
    - Il valore di Soglia diventa quello quello di FinCong/2; si ritorna alla fase di partenza lenta.

L'idea dell'algoritmo è quindi quella di aumentare esponenzialmente il numero di segmenti che vengono spediti (partendo da 1) senza aspettare il riscontro fino ad arrivare al valore della soglia; da tale valore in poi tale numero viene aumentato di 1 e infine, quando si verifica una perdita, si ritorna al valore 1, ponendo allo stesso tempo come nuovo valore di soglia quello dell'ultimo numero di segmenti spediti (ovvero quello contenuto in FinCong) diviso 2.

Fairness di TCP

Il meccanismo di controllo della congestione visto con l'algoritmo *Tahoe* è detto AIMD (aumento addittivo, decremento moltiplicativo) dato che, a parte la fase dalla partenza lenta, il numero di segmenti che possono essere spediti aumenta di 1 ad ogni passo (ovvero ad ogni RTT) mentre si dimezza ad ogni passo quando la rete è congestionata.

Questo meccanismo AIMD fa si che *TCP* sia *fair* (equo), nel senso che se ci sono più connessioni attive contemporaneamente su uno stesso canale, dopo un fase iniziale a regime ogni connessione userà la stessa percentuale della banda disponibile. Intuitivamente questo avviene perchè la connessione che usa più banda al momento del decremento (moltiplicativo) viene penalizzata più di quella che usa meno banda, mentre l'aumento (addittivo) è lo stesso per entrambe.