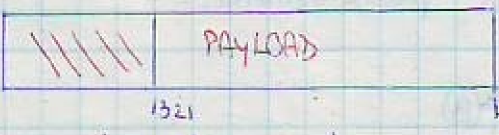
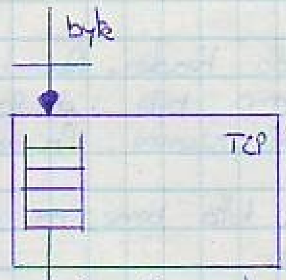
Si noti che la comunicazione è monodirezionale. Il TCP regola, attraverso il buffer, il modo di invio dei dati all'IP. Questa cosa accade in entrambe le parti, e avviene attraverso un insieme di regole. Ma quasi sempre la comunicazione è bidirezionale. Quindi il buffer di ricezione e trasmissione è in entrambe le parti. Esiste anche la comunicazione full-duplex in cui due TCP possono ricevere e trasmettere contemporaneamente.

Si noti che per garantire la correttezza delle informazioni c'è bisogno di un sistema di numerazione e riscontro. Il TCP numera i byte. Ogni byte ha un numero di sequenza. Attenzione che il TCP non numera i segmenti. Per esempio:

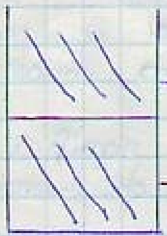


esiste un campo nell'header denominato sequence number contenente l'indirizzo del 1° byte presente nel PAYLOAD (1321). Vediamo ora come il TCP garantisce la trasmissione dei dati in sequenza e senza buchi.



• il TCP prende tot byte e forma il segmento

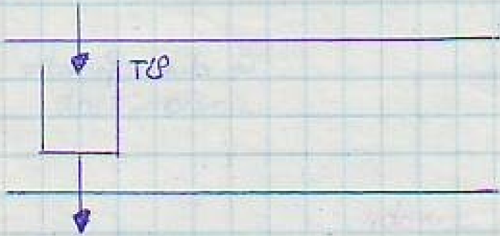
struttura del buffer di trasmissione:



→ Byte non ancora trasmessi

→ sezione dei byte trasmessi ma non ancora riscontrati dal ricevitore. Vengono conservati per eventuali ritrasmissioni.

I byte vengono eliminati dal buffer di trasmissione dopo un acknowledge. Vediamo ora la formazione dei segmenti:

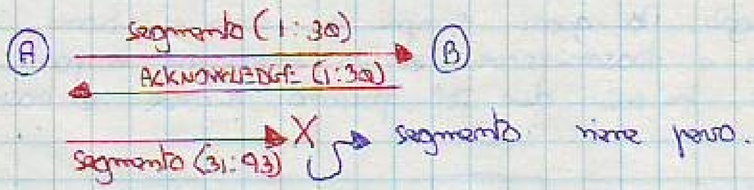


Criterio: in generale il TCP aspetta di avere almeno un MSS byte nel buffer di trasmissione prima di formare il segmento. Si fa ciò per avere una maggior efficienza.

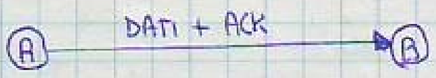
Si noti che se il PAYLOAD è piccolo abbiamo un'elevata percentuale di byte di overhead e una bassa percentuale di dati. Quindi il TCP cerca di massimizzare la lunghezza dei segmenti. Non sempre si creano però segmenti di lunghezza massima. Può accadere che l'applicazione comunichi al TCP di forzare la trasmissione del segmento anche se non ha raggiunto la sua lunghezza massima. Questo può essere fatto attraverso la primitiva PUSH. Analizziamo ora il processo di trasmissione:

A manda dei byte alla macchina B. Ipotizziamo che A invii un segmento con byte da 1 a 30 a B.

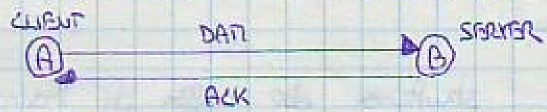
Ricordandosi che la comunicazione TCP è bidirezionale si ha:



Per ogni segmento inviato dal TCP viene attivato un timer. Se ricevo un acknowledge prima che il tempo sia scaduto allora va tutto bene, altrimenti il TCP ritrasmette il segmento. Questa strategia va sotto il nome di ACKNOWLEDGE POSITIVO. L'header TCP ha, come vedremo, vari campi tra cui il sequence number e l'acknowledge number. Nel sequence number c'è il numero di sequenza del primo byte del PAYLOAD, mentre nell'acknowledge number ci sono informazioni che uso per inviare l'acknowledge dei dati mandati precedentemente.



Chiaramente se la trasmissione è monodirezionale si ha:



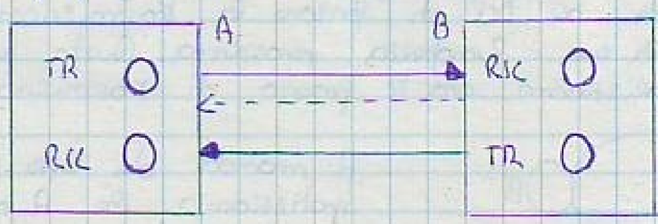
Si noti che gli ACKNOWLEDGE sono segmenti formati dal solo header. Quindi l'acknowledge number indica il numero di sequenza del prossimo byte che la macchina la quale ha inviato l'acknowledge si aspetta di ricevere. Per esempio:

ACK (10000) significa che fino al byte 9999 va tutto bene; mi aspetto di ricevere i byte a partire da 10000.

Quindi gli acknowledge sono di gruppo e non di singolo byte. Tutto ciò viene fatto sempre per questioni di efficienza. Il meccanismo di numerazione e riscontro è un meccanismo a finestra.

Le due finestre sono sincronizzate.

All'apertura della connessione TCP ogni macchina è responsabile della dichiarazione della propria receive window.



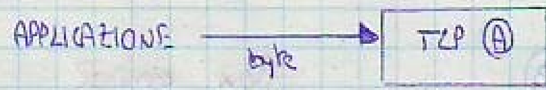
Se B ha una receive window di 10.000 byte, allora A dovrà avere una send window di 10.000 byte. In pratica la send window viene adattata in base alla receive window.

Bisogna ricordarsi poi di dichiarare la dimensione della receive window in byte. La receive window contiene i byte ricevuti ma non ancora elaborati. Comunque ogni byte di dati è numerato in entrambe le finestre. Consideriamo per esempio:

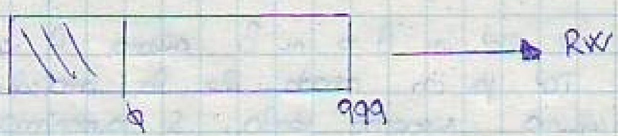
Supponiamo che:

$$\begin{cases} 1^{\circ}\text{ byte} \rightarrow 0 \\ 2^{\circ}\text{ byte} \rightarrow 2999 \end{cases}$$ (in realtà non è così).

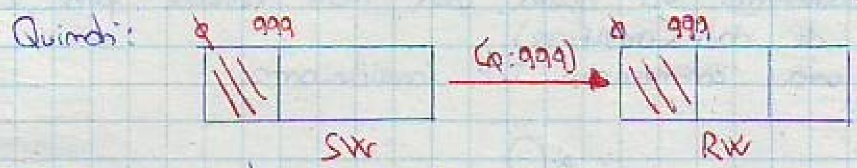
Ipotizziamo di avere segmenti da 1000 byte. La SW è quel pezzo di buffer contenente i byte trasmessi ma non ancora riscontrati.



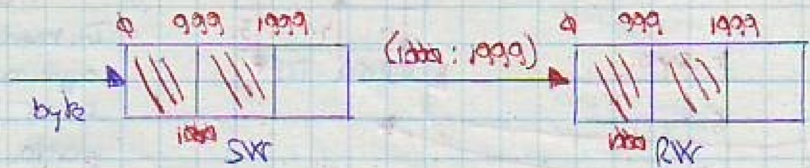
Il TCP forma il seguente segmento:

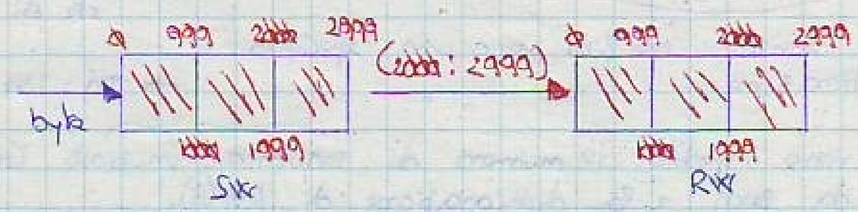


Quindi:

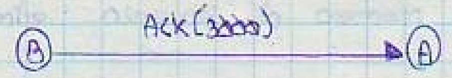


Il TCP ricevente non fa ancora nessun acknowledge perché sia RW che SW hanno ancora dello spazio libero.



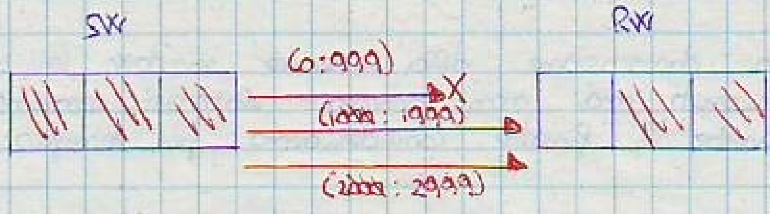


A questo punto 'B' invia l'acknowledge ad 'A'. Poi 'B' passa i dati all'applicazione. Supponiamo ora che 'B' spedisca 3000 byte all'applicazione. Così facendo RW si pulisce e in più passa da 3000 a 5999. Il TCP trasmettitore questo però non lo sa. Quindi:

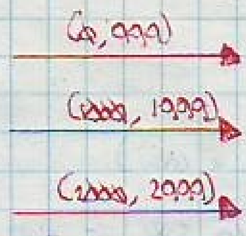


e quindi il TCP trasmittente pulisce il SW.

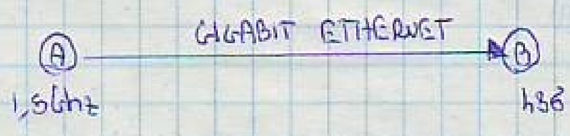
Ma se le cose vanno male che accade!



Il ricevente a questo punto si blocca. Non si può inviare nessun acknowledge. Scatterà il timer legato al primo blocco di dati, e quindi il TCP dovrà rispedire tutta la SW.

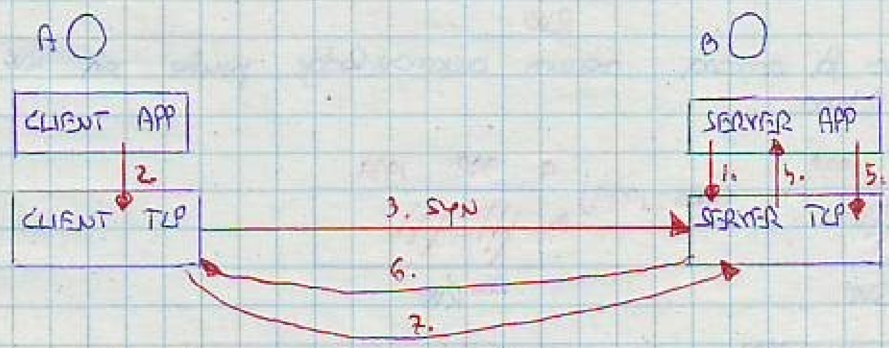


Con SW ed RW si può anche implementare il controllo di flusso. Si consideri:



flow control

Se si usa il TCP in A e in B allora A capisce che l'RW di B è piena e si ferma. Quindi il TCP fa in modo che la macchina trasmittente veloce si adatti alla velocità della macchina ricevente lenta. Se avessimo UDP questa cosa non verrebbe fatta, e quindi perderei la maggior parte dei dati (OVERFLOW).
Vediamo ora come viene attivata una connessione TCP. Consideriamo:



1. richiesta di effettuare la PASSIVE OPEN. In breve la server application dice al server TCP che è pronta a fare richieste di servizio.

2. ACTIVE OPEN = richiesta di un servizio specifico verso uno specifico socket di destinazione.

→ seq = 700

3. SYN = SYNCRONIZE in cui viene spedito il numero di sequenza iniziale (non sempre si parte da zero) e la dichiarazione di RW(A).

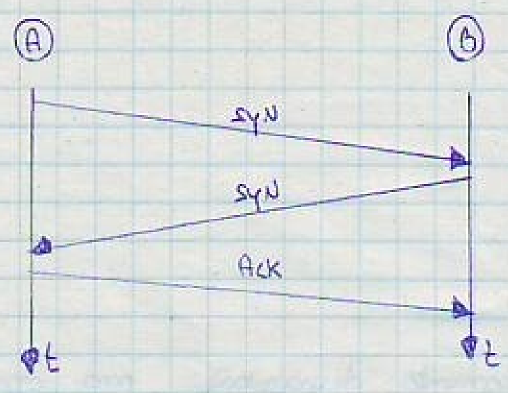
4. Interrogazione dell'applicazione.

5. Risposta.

6. Ritorna un pacchetto SYN che contiene un numero di sequenza estratto a sorte, e un Ack number (ACK 701) e in più RW di B.

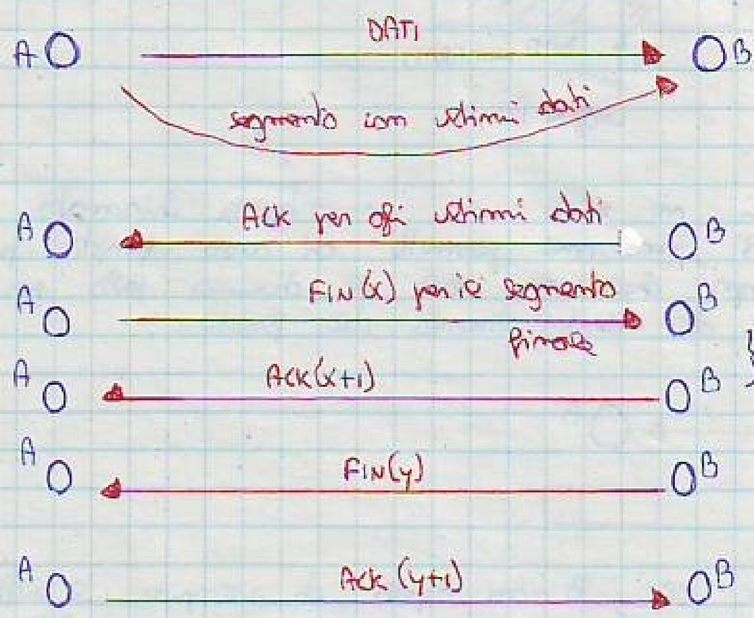
7. Ack (401).

Infine entrambe le applicazioni vengono avvisate che la connessione TCP è aperta.



Se il ROUNDTRIPTIME non è trascurabile, la sola apertura della connessione ci fa perdere tempo.

Le connessioni TCP vanno anche chiuse. Per esempio:



} si informa l'applicazione della macchina B

→ FIN chiude la connessione.

Vediamo ora l'header del segmento TCP:

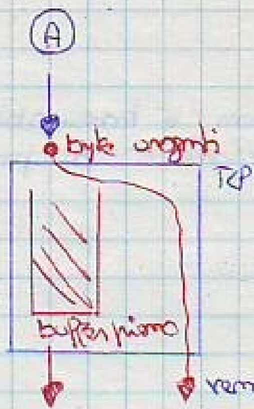
SOURCE PORT			DESTINATION PORT
SEQUENCE NUMBER			
ACK NUMBER			
HLEN	—	FLAG	WINDOW
CHECKSUM			URGENT POINTER
eventuali opzioni			

32 bit

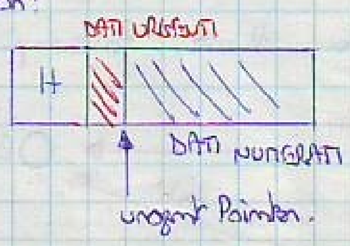
20 byte

Nel campo flag ci sono dei flag per dire se un pacchetto è SYN o è normale. C'è un flag speciale chiamato flag urgent che ci dice se ci sono dati urgenti.

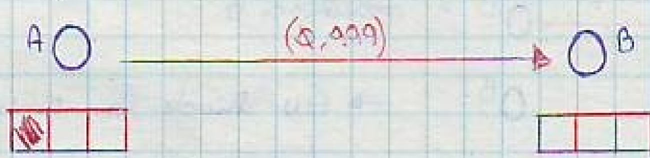
Nel campo WINDOW dichiaro la mia RWr, mentre il campo URGENT POINTER punta alla fine del blocco di dati urgenti. Per esempio:



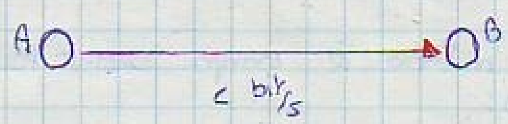
vengono piazzati nel primo segmento disponibile, ma non vengono numerati. Quindi:



Per quanto riguarda le opzioni ne esiste una importante chiamata LARGE WINDOW OPTION che aumenta l'overhead, ma mi permette di avere finestre più grandi. Infine anche per questo campo checksum vale il discorso fatto per lo pseudoheader. Vediamo un altro esempio del funzionamento delle finestre:

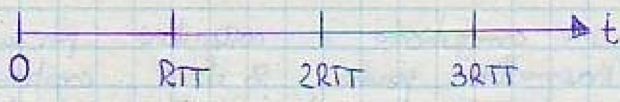


Normalmente se c'è abbastanza CPU, B non aspetta di riempire il buffer. Semplicemente passa subito il segmento all'applicazione, invia un acknowledge e slitta di 1000 byte la sua finestra. Questo è lo schema di funzionamento più tipico. Normalmente la RW contiene un numero intero di segmenti di lunghezza massima. Di solito viene dimensionata su un multiplo della massima dimensione del segmento. Il colloquio tra l'applicazione ricevente e la RW è proprietario e dipendente dall'implementazione e non è legato ai segmenti. L'applicazione può richiedere quanti byte vuole. Il TCP ricevente comunque aspetta di avere uno spazio libero sufficiente a contenere almeno un segmento prima di inviare un acknowledge. Questo viene fatto perché altrimenti verrebbero trasmessi segmenti molto piccoli e quindi con poca efficienza. Mediante la numerazione e il riscontro il TCP garantisce una trasmissione corretta dei dati evita il fenomeno dei fuori sequenza, cioè il ricevente sa sempre come ordinare i segmenti che riceve. Inoltre il TCP evita le duplicazioni ed effettua il flow control. Purtroppo le prestazioni sono difficilmente prevedibili; consideriamo:



Il caso è dedicato e ha una capacità C.

Inoltre c'è un certo RTT (Round Trip Time) che in questo caso è legato alla velocità dell'impulso sul cavo. Supponiamo che B abbia una RW di W bit come dimensionamento. Qual è la velocità effettiva del collegamento?



Qual'è il tempo di trasmissione di una finestra di bit? W/C.
Ipotizziamo che:

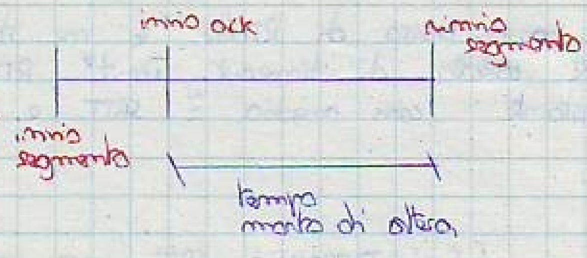
a) $W/C < RTT \Rightarrow V = \frac{W}{RTT}$ (V è la velocità di trasmissione del TCP). Così facendo sfrutto solo parzialmente il link.

b) $W/C \geq RTT$ e $V = C \Rightarrow$ sfrutto tutta la banda del link.

Quindi si ha:

$$V = \min\left\{\frac{W}{RTT}; C\right\} \quad e \quad \begin{cases} a) \ W < C \cdot RTT \Rightarrow V = W/RTT \\ b) \ W \geq C \cdot RTT \Rightarrow V = C \end{cases}$$

Il prodotto $C \cdot RTT$ si chiama **prodotto banda ritardo**, e mi dice quanti bit stanno transitando dal ricevitore al trasmettitore. La finestra quindi dovrebbe essere maggiore del prodotto banda - ritardo. Se ho un link molto lungo e veloce allora $RTT \cdot C > 65535$ byte che è la dimensione massima della finestra. Ecco che in questo caso è utile la TCP LARGE WINDOW OPTION che consente di aumentare il numero di bit per la codifica della finestra. Questa opzione viene usata per i collegamenti speciali. Anche i link satellitari sono speciali e veloci. Anche il TIMEOUT dei segmenti è importante. Ma come si dimensiona il timeout o meglio il timer? Questo è un problema del TCP. Ipotizziamo di avere un collegamento su una rete molto veloce. In questo contesto il timeout dovrebbe essere piccolo. Questo si ha perché così è difficile che mi scatti il timeout in maniera spuria. Se metto un timeout lungo, può accadere che se invio un segmento e subito dopo il ricevente del segmento invia un acknowledge che però va perso, mi tocca aspettare molto prima di capire che devo ritrasmettere l'intero segmento.



Se invece il collegamento è molto lento, se metto un timeout corto quest'ultimo scatta spesso e a sproposito. Quindi il timeout dovrebbe essere più lungo.