RDT 3.0 - continuo

Calcoliamo il throughput nel canale

Usiamo l'esercizio visto in precedenza per calcolare il throughput della trasmissione:

Ricordiamo che il throughput è il valore dato dalla quantità di dati trasmessi in un'unità di tempo; nell'esempio venivano trasmessi 8000 byte con un ritardo di propagazione di 15ms; il ritardo complessivo ammonta quindi al RTT (15*2) più il tempo di trasmissione, che è dato da L/R, quindi otteniamo un totale di `30.008 ms`.

Di conseguenza il throughput complessivo è `8000 / 30.008 = 266.6 Kb/s`

Visto che il massimo throughput del canale corrisponde ad 1Gbps capiamo che questo protocollo riduce sensibilmente l'utilizzo ottimale del canale.

Operazioni pipelined nel protocollo

Siccome il problema principale è caratterizzato da un RTT elevato, l'idea è di mantenere il canale più a lungo possibile; usando il pipelining osserveremo degli effetti collaterali all'interno della rete.

Pipelining

Poichè le connessioni sono tante, per via del pipelining, una macchina deve memorizzare un enorme numero di risorse (per attendere i loro riscontri). Se ad esempio il massimo numero di segmenti inviati in pipeline è 4, nulla ci vieta di inviare altri segmenti dopo l'invio del quarto; l'unica cosa che si deve attendere è il **riscontro del primo segmento**. Infatti, con l'arrivo del primo segmento, è come se si liberasse lo spazio in finestra, quindi abbiamo la possibilità di spedire un ulteriore pacchetto.

Questo tipo di comportamento viene chiamato **a finestra scorrevole**: assumo che la finestra possa liberare nuovi numeri di sequenza per la spedizione man mano che arrivano i riscontri precedenti.

Go-Back-N: sender

Nel momento in cui prevediamo che il mittente possa spedire più segmenti in successione, non è più sufficiente una **coppia di numeri di sequenza** (fino all'rdt3.0 abbiamo considerato i numeri di sequenza 0 ed 1).

Se per due numeri di sequenza è sufficiente un bit, un insieme di numeri di sequenza richiederanno l'uso di **16 bit**. La dimensione del campo dipende quindi da quanti numeri di sequenza vogliamo gestire.

Queste "tacchette" rappresentano i numeri di sequenza che sono attribuiti di volta in volta ai segmenti che vogliamo spedire. I diversi colori indicano che i numeri di sequenza corrispondenti alle barrette verdi, sono relativi a segmenti già spediti, e di cui abbiamo già ricevuto un riscontro ACK. Questo vuol dire che questi vengono rimossi dai buffer in modo da liberare memoria.

Dal primo segmento giallo inizia la **finestra** di dimensione N; la variabile **send_base** tiene traccia del primo numero di sequenza nella variabile, mentre con la variabile **nextsequnum** tiene traccia del numero di sequenza che posso assegnare al pacchetto da spedire.

I segmenti gialli sono già stati spediti, ma non abbiamo ancora ricevuto i riscontri. Quando ci sarà da spedire un nuovo segmento, questo prenderà il numero di sequenza indicato dalla variabile **nextseqnum**. Se il mittente produce un numero tale di segmenti tale da saturare la finestra, non sarà possibile spedire ulteriori pacchetti.

Questo protocollo prevede l'utilizzo degli **ACK comulativi**: non è un riscontro che fa riferimento ad un singolo frammento, ma ha un numero di sequenza; questo numero conferma la ricezione di quel segmento e di tutti i numeri di sequenza inferiori. Conferma quindi la ricezione di diversi pacchetti.

Viene utilizzato **un solo timer**, che viene associato al segmento più vecchio. Quando si verifica il timeout di quel segmento, questo viene rispedito. Il go-back-n non è pero in grado di gestire pacchetti che arrivano in ricezione **fuori ordine**; non sarà quindi spedito solo il segmento per il quale si è verificato il timeout, ma anche tutti i segmenti successivi.

Go-Back-N: ricevente

Il ricevenete non utilizza alcuna finestra, ed impiega solo una variabile **rcv_base** che costringe a ricevere in ordine i segmenti spediti dal mittente. Questa variabil tiene traccia del numero di sequenza atteso. Se viene ricevuto quel segmento (con quel num di seq) allora il pacchetto viene memorizzato (passato ai livelli applicativi).

Se arriva un segmento con un num di seq maggiore rispetto a quello presente in rcv_base, allora abbiamo un **fuori ordine**, quindi il segmento precedente o si è perso o sarà ricevuto in ritardo. Il go back n (puro) scarta il pacchetto (perchè non è quello atteso), mentre altre varianti consentono di ospitare i pacchetti fuori ordine.

Se viene scartato, il pacchetto dovrà essere rispedito, nel momento in cui avverrà un timeout al lato del mittente.

Dal ricevente è presente una singola variabile **rcv_base** che tiene traccia del numero di sequenza atteso, se arriva il segmento con quel numero viene accettato, altrimenti viene scartato.

Go-Back-N in azione

In questo esempio assumiamo che la finestra del mittente è di 4 elementi; inizialmente abbiamo la possibiltà di spedire i pacchetti con i numeri di sequenza `0-1-2-3`.

Dopo aver spedito il pacchetto con num di seq 3, essendo 4 fuori dalla finestra, il mittente è costretto ad attendere.

Il pacchetto "numero 2" viene perso. Nella versione pura il pacchetto 3 viene quindi scartato perchè è un fuori ordine, proprio perchè il ricevente si aspetta il pacchetto 2. Il ricevente invia un riscontro per l'ultimo pacchetto che ha ricevuto correttamente, quindi invia un ACK0 ed ACK1:

Ricevuti i riscontri, la finestra lato mittente avanza quando viene ricevuto un riscontro ACK0, quindi la finestra diventa `1–2–3–4`, e possiamo quindi spedire il pacchetto con numero di seq 4.

Viene ricevuto un altro riscontro e viene quindi liberato un altro num di seq all'interno della finestra:

Viene quindi spedito anche il pacchetto 5. Ad un certo punto avviene il timeout del pacchetto 2. Quando si verifica il pacchetto viene rispedito. **Il problema** è che vengono rispediti anche i pacchetti 3-4-5 (pacchetti spediti dopo il 2), anche se questi erano stati spediti correttamente, con tanto di ACK.

Questo protocollo risulta essere si efficiente per quanto riguarda la spedizione di più pacchetti in successione, ma poco efficiente nel momento in cui è presente un errore.

Selective Repeat

Un miglioramento dal punto di vista dell'efficienza nel momento in cui il canale ha un'alta probabilità di errore, è il **protocollo selective repeat**. In questo protocollo i pacchetti sono riscontrati in maniera selettiva individuale: non usiamo il meccanismo degli ACK comulativi visti in precedenza.

Ogni riscontro riscontra **un singolo pacchetto**. Come implicazione abbiamo la scelta dal lato ricevente di **memorizzare i pacchetti fuori ordine**, e vengono riscontrati in modo selettivo.

Nel momento in cui prevediamo che ciascun pacchetto possa essere riscontrato in modo indipendente, anche il timer deve essere assegnato indipendentemente (uno per ogni pacchetto). Quando si verifica l'evento di timeout sarà rispedito un unico singolo pacchetto, e non altri.

Sender:

Il sender funziona in modo simile all'implementazione precedente; abbiamo una finestra, all'interno della quale vengono spediti dei pacchetti. Per far avanzare la finestra i pacchetti devono ricevere un ACK individuale.

Ricevente

Nel ricevente viene usata solo una variabile per tenere traccia del segmento atteso, ma usiamo anche una seconda finestra che indica quanti pacchetti è possibile ricevere **fuori ordine**; questo perchè è possibile che il ricevente riceva un pacchetto fuoriordine che lo memorizzi e che lo riscontri in maniera separata (individualmente).

Il fatto di usare una finestra dal lato ricevente, implica un diverso uso delle risorse: dobbiamo infatti usare più risorse lato ricevente. Questo perchè i pacchetti ricevuti fuori ordine devono essere temporaneamente memorizzati in attesa che i pacchetti restanti arrivino.

Se guardiamo l'immagine, notiamo come dei pacchetti in viola siano stati ricevuti **fuori ordine** e sono all'interno di un buffer in attesa che altri pacchetti (in grigio) arrivino per essere ordinati correttamente.

Demo

Prima a seguito di una perdita di pacchetto inviavamo un ACK che testimoniava l'arrivo di tutti i pacchetti arrivati correttamente fino ad un certo punto, mentre ora inviamo gli ACK individualmente per ogni pacchetto.

Come prima si verificherà un timeout per un pacchetto perso per strada, e l'operazione di recovery è specifica, infatti viene rispedito **solo il pacchetto 2**!

La finestra lato ricevente viene fatta avanzare non più di una unità ma in modo da portare fuori tutti i pacchetti ricevuti in ordine, insieme al pacchetto 2.

Quando arriva un ACK2 al mittente, cosa succede? La finestra lato mittente scorre in avanti di tutte e quattro le posizione inviate correttamente.

TCP

Il protocollo è di tipo **point-to-point** perchè usiamo le connessioni; è inoltre un protocollo affidabile e che offre la comunicazione a stream. Grazie all'utilizzo dello stream, i messaggi non hanno limiti di grandezza.

Vengono usato gli ACK comulativi, infatti ai fini di affidabilità il TCP usa un protocollo molto simile al **go-back-n**, e sfrutta il pipelining. Infine il TCP consente di controllare il flusso **rispetto alle risorse** sia del ricevente che della rete.

🏁 1:14

Come viene realizzato tutto cio? - L'header

Più meccanismo abbiamo, più grande diventa l'intestazione (per via delle informazioni di controllo) più overhead abbiamo. Con il TCP il throughput è **sicuramente più basso rispetto a UDP**; questo è il motivo per cui. UDP viene preferito per comunicazioni real time, come ad esempio streaming video.

Abbiamo due campi da 16 bit, quindi ogni porta è compresa tra 0 e 65335, quindi i numeri di porta non sono infiniti. Questi due campi da 16 bit sono dedicati alla porta source e la porta destinazione.

Successivamente abbiamo un **sequence number** utilizzato sia per l'affidabilità sia perchè vogliamo gestire la connessione a stream, su un canale di comunicazione FIFO. Per realizzare un canale FIFO è necessario ordinare i pacchetti in arrivo. Infatti per capire a che segmento il riscontro fa riferimento, abbiamo bisogno del **numero di riscontro**.

Abbiamo quindi un numero di sequenza usato per etichettare i segmenti che contengono i dati ed un numero di riscontro per etichettare i segmenti che contengono i riscontri.
I campi previsti nell'intestazione dei pacchetti TCP, sono usati sia per i segmenti contenenti dati, sia per segmenti non contenenti dati.

Per migliorare l'efficienza, il TCP evita di inviare segmenti contenenti solo riscontri; usa infatti una tecnica chiamata **piggy backing** che sovrappone i riscontri ai dati: se A ha inviato dei dati a B, B invia un riscontro, ma potrebbe aggiungere anche dei dati per ottimizzare il tutto.

Per indicare che il campo **sequence number** è un campo significativo (visto che a volte posso inviare un segmento di soli dati, senza quindi un riscontro), viene usato un bit flag: **bit A**. Questo flag non è l'unico, ma ne vengono usati anche altri.

Abbiamo inoltre un campo **checksum**, sempre usato per verificare la validità dei pacchetti, e delle **opzioni**, di dimensione variabile.

Siccome l'intestazione ha **dimensione variabile**, dobbiamo fissare un massimo: il massimo numero di byte di un'intestazione TCP è di **60 byte**, mentre il minimo è **20 byte**; in altre parole varia da **5 word a 15 word di 32 bit / 32 byte.**

Per quanto riguarda la lunghezza dell'intestazione, abbiamo un campo **di 4 bit** che viene usato per rappresentare dei numeri da 0 a 15; questo numero ci dice **quanto è grande l'intestazione**.

Numeri di sequenza ACKs in TCP

Il numero di sequenza che abbiamo nel segmento è il numero di sequenza che viene recuperato dalla sequenza di cui parlavamo prima; questo dice qual è il numero di sequenza libero che può essere assegnato ad un numero di sequenza da spedire. Viene quindi prelevato un numero di seq libero e viene assegnato al segmento.

Quando viene ricevuto un riscontro, questo ha il **flag ACK ad 1**, quindi il campo ACK number è significativo (se non c'è il flag ad uno il campo non è significativo). Questo riscontro farà riferimento ad uno dei pacchetti spediti, se è uno dei num di seq più bassi la finestra scorre.

🏁 1:30

MSS - Maximum Segment Size

Non indica la dimensione massima del segmento! Indica invece **la massima dimesione dell'area dati di un segmento TCP**: non di tutto il segmento ma solo **dell'area dati**.

Il TCP aggiunge un ulteriore vincolo nelle dimensioni dell'area dati. L'MSS riflette **la dimensione dell'area dati del frame** in cui il datagram viene incapsulato. Segue un altro parametro chiamato **MTU - Maximum Transmission Unit**; questo è un parametro che se violato non consente il corretto funzionamento della rete.

Nel caso di una rete locale, questo parametro vale **1500**, quindi l'area dati di un frame Data-Link è di **1500 byte**. Se all'interno di un frame vogliamo incapsulare un segmento TCP (a sua volta incapsulato in un datagram ip), dobbiamo togliere dai 1500 byte, 20 byte per l'intestazione TCP e 20 byte per l'intestazione IP (nel caso di intestazione minima).

Per comunicazioni su reti locali, l'MSS è **tipicamente 1460 byte**.

🏁 1:39 - da questo momento in poi vengono usate le slides della professoressa Ronaldo.

Quando il processo client invia tramite una socket i dati, questi vengono inseriti in un buffer del software TCP, chiamato **send buffer**. Di conseguenza, l'entita **mittente** del TCP estrae periodicamente un certo numero di byte dal buffer, e **costruisce i segmenti TCP**. Come abbiamo visto il numero dei byte estratti è **limitato dall'MSS**;

Per capire il valore tipico dell'MSS dobbiamo considerare il **lunghezza massima del frame più grande** che può essere inviato sul canale dati **inviato dall'host**; questo valore viene detto **MTU - Maximum Transmission Unit**. Scegliamo quindi un MSS in modo che il segmento TCP possa essere incapsulato **in un singolo frame a livello data link**.

Come abbiamo visto in **reti LAN il valore dell'MSS è di 1460**, ma possiamo avere anche valori pari a **536 e 512** byte.

A che serve l'MSS?

Lo scopo dell'MSS è quello di evitare che il **livello IP frammenti i segmenti TCP** per inviare i datagram sulla rete. Quando l'entita del TCP riceve un segmento, lo **inserisce nel buffer di ricezione**; infatti **ogni lato** della connessione è provvisto di un **buffer di invio e di ricezione**.

Struttura dei segmenti TCP

La lunghezza dell'intestazione di un segmento TCP è data da un'informazion (flag) presente nell'intestazione stessa: usiamo 4 bit per rappresentare un valore **da 0 a 15**, che ci indica la lunghezza dell'intestazione. E' presente una **Internet Checksum**, come in UDP. Sono infine presenti delle Opzioni di lunghezza variabile:

Opzioni

Il campo delle opzioni è facoltativo, e viene usato per fornire funzionalità aggiuntive, che non vengono fornite mediante i campi regolari dell'intestazione. L'opzione più importante è quella che permette ad un mittente ed un ricevente di **negoziare la dimensione amssima del segmento MSS**.

Puntatore urgente

Il TCP permette la trasmissione **fuori banda** di dati ad alta priorità: ad esempio, quando il TCP è usato per unas essione di accesso remoto, l'utente può decidere di inviare una sequenza da tastiera che interrompe o termina il processo dall'altro lato; quando ad esempio siamo collegati in remoto ad una macchina e stiamo eseguendo un programma, ci basta digitare la combinazione **CTRL+C** per

terminare sia il processo che la trasmissione.

Questi devono essere consegnati al ricevente il prima possibile, indipendentemente dalla posizione all'interno dello stream. E' possibile capire a quale **posizione del segmento** i dati urgenti terminano perchè in questo campo è presente un puntatore proprio a quella posizione.

Quando il ricevente riceve dei **dati urgenti**, deve avvisare il processo applicativo di entrare in modalità urgente.

Checksum

La checksum è un campo di **16 bit** che contiene un valore intero utilizzato dal TCP (e anche da UDP) della **macchina host di destinazione** per verificare l'integrità dei dati e la correttezza dell'intestazione.

Per il calcolo della checksum il TCP ha bisogno di aggiungere una **pseudointestazione** al segmento, per effettuare così un controllo anche sugli indirizzi IP di destinazione e provenienza (anche loro potrebbero essere corrotti).

Bit di Flag

- **URG** indica la presenza di dati urgenti: se il bit è posto ad 1 il campo puntatore ai dati urgenti conterrà la posizione dell'ultimo byte dei dati urgenti.
- **ACK** indica se il campo **numero di riscontro** è significativo, ovvero se il segmento contiene un segmento o meno
- **PSH** se posto ad 1 indica che il destinatario dovrebbe immediatamente inviare i dati al libello applicativo e non bufferizzarli (poco usato).
- **RST** re-inizializza una connessione diventata instabile; inoltre rifiuta un segmento non valido o dopo l'apertura di una connessione

- **SYN** usato per creare connessioni
- **FIN** usato per chiudere la connessione; il mittente non ha più dati da inviare.

Numeri di sequenza e di ACK

Numero di sequenza

Un numero di sequenza indica la posizione relativa all'origine dello stream **del primo byte** nei dati di un segmento.

Numero di ACK

Il numero di ACK è rappresentato dal **numero di sequenza del successivo byte atteso da chi riscontra** (sender). Il TCP usa **ACK comulativi**.

Per quanto riguarda i segmenti non in ordine questi possono essere o scartati e ritrasmessi oppure (più comunemente) **bufferizzati** per efficienza in termini di banda (abbiamo visto come la ritrasmissione possa limitare l'utilizzo del canale).

Gestione di numeri di sequenza e riscontro

Con la tecnica del **piggybacking** quando il ricevente invia il riscontro di un frammento, invia anche dei nuovi dati (eventuale risposta), in modo da ottimizzare la comunicazione.

Timeout nel TCP

Il timeout nel TCP è un **valore dinamico**, solitamente più grande dell'RTT (round trip time), anche se l'RTT varia. Se non scegliamo bene il valore del timeout, potremmo ottenere dei **timeout prematuri**, dove il timer scade prima della ricezione dell'ACK e quindi avremmo delle **ritrasmissioni non necessarie**.

Se invece il timeout è troppo lungo, la reazione sarebbe troppo lenta rispetto alla perdita di segmenti (ci accorgeremmo della perdita di un segmento con troppo ritardo).

In che modo possiamo stimare il RTT?

sampleRTT: E' il tempo misurato dalla trasmissione di un segmento e dalla successiva ricezione dell'ACK. Con questo valore vengono ignorate le ritrasmissioni.

sampleRTT varia tra una spedizione e la successiva; abbiamo quindi bisogno di un modo per stimare il successivo RTT, visto che cambierà con la prossima spedizione. Possiamo quindi usare **una media sulle misure recenti**.

Raddoppio dell'intervallo di timeout

Lo scadere di un timer viene probabilmente causato dalla congestione della rete: infatti nei periodi di congestione, se i mittenti continuano a trasmettere (o ritrasmettere nel caso di errori o timout) dei pacchetti, la congestione va a peggiorare.

Nel TCP è possibile prevedere un primo meccanismo di **controllo della congestione** forzando il mittente a ritrasmettere ad intervalli sempre più lunghi finchè non si riesce a trasmettere il segmento senza timeout (quindi nei primi tentativi il tiemout è corto mente all'ultimo tentativo è posto ad infinito). Dopo lo scadere del timer, la maggior parte delle implementazioni del TCP reimpostano il

valore del timeout a:

`timeout = A*timeout`

- Il valore tipico di A è 2, quindi abbiamo il **doppio del valore precedente**
- Gli intervalli crescono ad ogni ritrasmissione (o tentativo)
- viene usato L'algoritmo di **Karn** con tecnica del **backoff del timer**.

Per l'invio di nuovi segmenti e quando viene ricevuto un ACK per un segmento senza ritrasmissione il TCP aggiorna la stima dell'RTT e aggiorna il timeout.

Controllo del flusso in TCP

Ricevente

Il ricevente comunica esplicitamente e dinamicamente l'ammontare di **spazio libero nel buffer**; questo valore corrisponde al **campo finestra di ricezione**.

Mittente

Il mittente regola la quantità di byte trasmessi e di cui non si è ricevuto riscontro, in modo che deve essere inferiore all'ultima **finestra di ricezione ricevuta** (da parte del ricevente di cui abbiamo appena parlato).

`nextseqnum - sendbase <= RcvWindow`

In questo modo **il mittente non sovraccarica il ricevente**, trasmettendo quindi in maniera abbastanza lenta da permettere lo svuotamento del buffer del ricevente.

Eccesso di piccoli pacchetti

Cosa succede se le operazioni di lettura da parte del processo ricevente e quelle di scrittura da parte del mittente **sono lente**?

Abbiamo come conseguenza che l'applicazione mittente è lenta, perchè invia i segmenti con pochi byte di dati.
Anche l'applicazione ricevente è lenta, perchè riceve pochi byte alla volta. Se consideriamo una connessione **telnet**, nel caso peggiore, appena l'utente digita un carattere, il TCP invia un segmento di `20+20+1 = 41 byte` e così via, ed il ricevente invia un riscontro **per ogni byte (singolo!)** ricevuto.

Capiamo quindi che inviando dei pacchetti troppo piccoli andiamo ad utilizzare inutilmente la rete, dove la quantità di byte di intestazione è maggiore rispetto ai dati effettivamente trasmessi.

Possiamo risolvere il problema grazie a due algoritmi:

- **Mittente** algoritmo di Nagle
- **Ricevente** algoritmo di clark o riscontro ritardato

Soluzione lato mittente - Algoritmo di Nagle

Il mittente invia subito il primo byte di dati disponibile, quindi accumula i dati successivi nel buffer in modo da inviarli insieme in un unico segmento TCP. Il primo byte di dati viene inviato per instaurare una connessione TCP, visto che questo tipo di protocollo richiede la connessione.

I dati inviati in segmenti TCP vengono incapsulati nei datagram fino alla dimensione massima di un MSS, o metà della finestra del ricevente. Questo valore viene calcolato sul minore dei due: `min(MSS,rcv_window)`.

Questo algoritmo garantisce che una connessione tCP non può presentare più di **un solo segmento piccolo** di cui non si è ricevuto un riscontro; per **segmento piccolo** si intende una dimensione **minore dell'MSS**.

Soluzione lato Ricevente - Algoritmo di Clarck

La situazione che vogliamo evitare è quella in cui il buffer del ricevente **è pieno** e l'applicazione legge un byte alla volta, liberando dal buffer un byte alla volta. Il mittente, per via di questo comportamento del ricevente, è costretto ad inviare anch'esso un byte alla volta.

Con la soluzione di Clark il riscontro inviato dal ricevente contiene un valore zero per **rcvWindow** finchè lo spazio libero nel buffer sarà minore di **min(MSS, RecvBuffer/2)**. Di conseguenza il mittente continua ad inviare byte.

Soluzione mediante il ritardo del riscontro

Il ricevente non invia un riscontro per ogni segmento ricevuto; di conseguenza ritarda la spedizione del riscontro fino a che **nel buffer non vi è uno spazio ragionevole**. Questa tecnica permette anche di **ridurre il traffico sulla rete**: la soluzinoe di Clark potrebbe richiedere due segmenti (uno di riscontro ed uno per aggiornare la dimensione della finestra).

Questa soluzione sicuramente aumenta il RTT, perchè ritardando la spedizione del riscontro, il tempo che passa dalla spedizione di un pacchetto ed il suo ACK aumenta.

Cosa accade se rcvWindow = zero?

Cosa accade se la finestra di ricezione vale zero? Il ricevente aggiorna la finestra quando l'applicazione elgge i dati dal buffer: se il buffer ricevente non ha nulla di inviare al mittente, questo non verrà informato che è presente dello spazio libero nel buffer. Abbiamo inoltre un altro problema: cosa accade se il segmento contenente l'aggiornamento viene perso?

Timer di persistenza

Questo timer viene avviato ogni volta che la finestra di ricezione è pari a zero; allo scadere del timer di persistenza il mittente invia periodicamente pacchetti di dimensione 1 byte. Il ricevente risponde con un ACK (con un valore nullo di rcvWindows) anche se non può memorizzare i pacchetti, ed ad un certo punto il buffer inizierà a svuotarsi ed il mittente verrà avvisato ricevento un valore non nullo di **rcvWindow**.

🏁 Slide 25 tcp.pdf

Dalle lezioni del prof 🏁 1:55

Gestione della connessione TCP

In java abbiamo detto che dopo la creazione della socket di benvenuto, di tipo ServerSocekt, abbiamo una socket già in gradio di ricevere richieste di connessione, che noi recuperiamo invocando accept().

Lato client viene costruita una socket di tipo Socket, che ci permette di creare una socket connessa con quella remota. Creare una connessione significa inizializzare una serie di variabile, tra i **numeri di sequenza iniziali**. Vengono anche creati i **buffer**, sia lato client che server.

Quando invochiamo la listen() (nativa) e quindi abbiamo creato una socket ti dipo server, lo stato in cui si pone il server è listen. Lo stato in cui è l'altra (prima della creazione della socket) è **closed**. Quando lato client invochiamo la connect() dallo stato **closed** si va nello stato **SYNSENT**: il TCP del client, invia un **segmento al TCP server**; questo segmento è un **segmento di controllo che non contiene dati**. Uno dei bit di flag (S) è posto ad 1, infatti questo bit viene usato per caratterizzare un segmento di controllo usato per chiedere l'attivazione della connessione.

Questo segmento ha anche un **numero di sequenza** che viene scelto dal client (da chi promuove l'attivazione della connessione); questo numero è iniziale (perchè poi si continuerà a partire da questo numero):

Il server può accettare o rifiutare la richiesta di connessione: se accetta passa nello stato **SYN RCVD** inviando un segmento contenente il bit **SYNbit** posto ad 1, che attiva la connessione. Questo segmento inviato come risposta dal server, contiene anche un **riscontro per il precedente**. Il numero di riscontro è **il numero di sequenza assegnato al segmento da parte del client + 1**.

Capiamo che questo segmento, anche se non ha dati veri e propri, ha di dimensione x+1 byte. Questo serve a **non confondere questo segme** con altri segmenti.