CAPITOLO 3: Livello di trasporto

**3.1 Introduzione**

Tra il livello applicativo e di rete risiede il livello di transporto che è il nucleo dell'architettura di rete. Il livello di trasporto ha la responsabilità di fornire servizi di comunicazione direttamente ai processi delle applicazioni in esecuzione sui differenti host sfruttando i l servizio offerto dal livello di rete.

Il problema principale di questo livello è capire come si effettua la connessione affidabile su un mezzo che potrebbe perdere o corrompere i dati. Inoltre, è importante capire come controllare il rate di trasmissione del livello di trasporto per evitare le congestioni all'interno della rete.

**3.2 Servizio del livello di trasporto**

Il protocollo del livello di trasporto fornisce comunicazione logica tra i processi delle applicazioni sui differenti host. In particolare, si intende che dalla prospettiva delle applicazioni, è come se l'host fosse connesso direttamente col processo in esecuzione. L'applicazione utilizza la comunicazione logica dal livello di trasporto per inviare i messaggi tra di loro. I protocolli del livello di trasporto, inoltre, sono implementati negli end-system ma non nei routers di rete.

Dalla parte del mittente, il livello di trasporto converte i messaggi del livello applicativo in pacchetti a livello di trasporto, detti segmenti, ciascuno dei quali contiene informazioni dell’utente. A questo punto, il livello di trasporto invia i segmenti sfruttando il servizio offerto dal livello di rete e dall’altra parte vengono ricevuti i segmenti che, una volta riassemblati, vengono resi disponibili all'applicazione. Quindi ho la possibilità di scambiare messaggi in modo affidabile.

**3.3 Relazione tra livello di trasporto e di rete**

Ricordiamo che il livello di trasporto giace appena sopra il livello di rete nello stack di protocollo. Mentre un protocollo di livello di trasporto fornisce comunicazione logica tra i processi in esecuzione sui differenti host. D’altra parte, un protocollo di rete fornisce comunicazione logica tra gli host.

**3.4 Protocolli di livello di trasporto in Internet**

Internet fornisce due protocolli di livello di trasporto al livello di applicazione.

Uno di questi protocolli è detto UDP (User Datagram Protocol) il quale fornisce un servizio inaffidabile, senza connessione alle applicazioni che lo invocano e senza rispettare l’ordine di invio dei file. Il secondo protocollo è il protocollo TCP, il quale fornisce un servizio affidabile, orientato alle connessioni alle applicazioni invocanti che rispetta l’ordine di invio dei file.

TCP: offre servizi aggiuntivi come il controllo della congestione della rete, il controllo della saturazione del buffer di sezione e il controllo delle connessioni.

UDP: non garantisce l’ordine dell’invio dei file a differenza del TCP

Il protocollo di livello di rete è detto Internet Protocol IP; esso è un servizio di consegna best-effort. Questo significa che IP rende il suo miglior sforzo per consegnare i segmenti tra host comunicanti, ma non lo garantisce. In particolare, oltre a non garantire la consegna dei messaggi, non garantisce la consegna ordinata dei segmenti.

Per queste ragioni il servizio IP non è affidabile.

**3.5 Multiplexing e Demultiplexing**

Le estensioni offerte dal livello di trasporto sono il multiplexing e il demultiplexing che estendono il servizio di consegna host-to-host fornito dal livello di rete in un servizio di consegna process-to-process per le applicazioni in esecuzione sugli host.

In particolare, nell'host di destinazione, il livello di trasporto riceve i segmenti dal livello di rete sottostante ed ha il compito di consegnare il dato contenuto in questi segmenti al giusto processo d'applicazione in esecuzione nell'host. Il livello di trasporto non trasporta il segmento direttamente al processo, ma al socket intermediario. Inoltre, poiché in qualsiasi istante di tempo vi possono essere più di un socket nell'host ricevente, ogni socket ha un identificativo unico.

A questo punto quello che vogliamo capire è come un host ricevente direziona un segmento alla giusta socket. A tale scopo, il segmento è dotato di campi utili al compimento di questa funzione ed il livello di trasporto controlla questi campi per identificare la socket ricevente e la direzione su di essa. Questo processo viene detto DEMULTIPLEXING.

Al contrario, come faccio a far si che pacchetti uscenti dalla stessa macchina abbiano informazioni che permettono di capire quale processo agirà su quei pacchetti, è detto MULTIPLEXING.

**Demultiplexing senza connessione (protocollo UDP)**

Quando una socket UDP viene creata, contiene l’indirizzo IP locale e un numero di porta non attualmente in uso nell'host. A questo punto supponiamo che un processo nell'host A con un suo numero di porta UDP, vuole inviare una porzione di un dato di un'applicazione ad un processo nell'host B con un certo numero di porta.

L'host A per fare ciò deve effettuare la multiplazione: crea un segmento di livello di trasporto che include i dati dell'applicazione, il numero di porta sorgente, il numero di porta destinatario e due altri valori. Il livello di trasporto a questo punto invia il risultante segmento al livello di rete che lo incapsula in un datagramma IP ed attua un tentativo “best-effort” (come tipico dal protocollo IP) per consegnare il segmento all'host ricevente. Appena il segmento arriva all'host, quest'ultimo lo demultipla alla corretta socket esaminando il numero di porta destinazione.

Se due segmenti UDP hanno differente indirizzo IP sorgente, ma hanno lo stesso indirizzo IP destinazione, allora i due segmenti saranno inviati allo stesso processo destinazione attraverso la stessa socket.

**Demultiplexing con connessione (protocollo TCP)**

Quando una socket TCP viene creata, contiene l’IP sorgente, l’IP destinazione, il numero di porta sergente e il numero di porta destinazione. Inoltre, quando un segmento TCP arriva dalla rete ad un host, l'host utilizza tutti e quattro i valori per demultiplare il segmento alla socket corretta. Al contrario di UDP, due segmenti con differenti indirizzi IP sorgenti o numeri di porta sorgenti saranno demultiplati in due differenti socket.

**Connectionless Transport: UDP**

UDP fa poco più di quello che farebbe un protocollo di transporto e, ad eccezione della multiplazione/demultiplazione e del controllo degli errori, non aggiunge nulla al protocollo IP (best effort: può perdere pacchetti e possono non arrivare in ordine). In UDP non vie nessun handshaking tra il livello di trasporto mittente e destinatario prima di inviare il segmento. Per questo motivo UDP viene detto connectionless. Ha dei vantaggi importanti:

1. Non richiedendo connessione, posso inviare subito un messaggio senza aspettare un certo tempo
2. È semplice poiché non ha una connessione in cui devo memorizzare informazioni
3. Ha un’intestazione piccola
4. Non ha il controllo di congestione, quindi può inviare dati a qualsiasi velocità

Un segmento UDP ha la seguente struttura:

• Il campo data del segmento UDP è occupato dai dati dell'applicazione;

• Il campo length specifica il numero di bytest nel segmento UDP (viene contato anche l'header). Questo valore è necessario poiché la dimensione del campo dati potrebbe cambiare da un segmento UDP ed un altro;

• Il campo checksum è utilizzato dall'host ricevente per controllare qualora vi siano errori nel segmento. In un segmento UDP, il campo checksum fornisce il controllo degli errori. Questo campo viene utilizzato qualora i bit all'interno del segmento UDP siano stati alterati nel passaggio fra sorgente e destinazione.

In particolare, dal lato del mittente effettua una somma in complemento ad 1 di tutti i 16-bit nel segmento con nessun overflow; il risultato viene messo nel campo checksum del segmento UDP. Dal lato destinatario si calcola il checksum del segmento e se è uguale a quello del campo checksum allora non vi sono errori.

**Principi di trasferimento dati affidabile**

Affrontiamo il problema del trasferimento di dati affidabile. Proseguiamo per gradi: il caso più semplice è quello di un canale già affidabile. Con un canale affidabile, nessun bit trasferito è corrotto o perso e tutti i segmenti sono consegnati nell'ordine in cui sono stati inviati. Questo è il servizio offerto da TCP. Inoltre, è responsabilità del protocollo di trasferimento dati affidabile di implementare l'astrazione del servizio di un canale affidabile tramite il quale si possono trasferire dati. Questo è un problema non banale poiché il livello sottostante potrebbe essere inaffidabile.

• rdt: protocollo di trasferimento dati affidabile;

• udt: protocollo di trasferimento dati non affidabile;

• \_send: indica il lato mittente di rdt

• \_rcv: indica il lato destinatario di rdt

Analizziamo una serie di protocolli “fittizi” sempre più complessi per ottenere un protocollo impeccabile di trasferimento dati affidabile:

1. **Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0**

Questo è il caso più semplice in cui voglio garantire comunicazione affidabile con il canale sottostante completamente affidabile, ossia non ci sono errori di bit e non perde dati. Utilizziamo una macchina a stati finiti per rappresentare il funzionamento di rdt1.0. Il mittente e il destinatario sono formati da un singolo stato:

• Le frecce indicano la transizione del protocollo da uno stato all'altro;

• L'evento che causa la transizione è mostrato al di sopra della linea orizzontale;

• L'azione presa quando l'evento si verifica è mostrato al di sotto della linea orizzontale. Dalla parte del mittente, si accetta il dato dal livello superiore tramite rdt\_send(data) (in pratica una chiamata dal livello applicativo), si crea il pacchetto contenente il dato, tramite make\_pkt(data), si invia il pacchetto nel canale tramite l'invocazione di udt\_send(packet) e si rimette in attesa di una nuova chiamata.

Dal lato del destinatario, si riceve un pacchetto dal canale sottostante tramite rdt\_rcv(packet), si estrae il dato dal pacchetto, tramite la chiamata extract(packet,data), lo si invia al livello superiore invocando deliver\_data(data) e si rimette in attesa di nuovi dati.

1. **Reliable Data Transfer over a Channel with Bit Errors: rdt2.0**

Questo è un modello in cui il canale sottostante è uno in cui i bit di un pacchetto potrebbero essere corrotti. Le cause sono molteplici come errori di propagazione, trasmissione o bufferizzazione. Questo modello utilizza sia ACK positivi che negativi permettendo al mittente di sapere cosa è stato ricevuto correttamente e cosa presenta degli errori e in caso il ricevente può richiedere di ripetere la trasmissione dei dati.

Per fare ciò vengono aggiunte delle funzionalità:

-Rilevamento di errore: è necessario permettere al destinatario di rilevare quando un bit presenta un errore (si utilizza un bit aggiuntivo);

-Ricezione del Feedback: è necessario che il receiver invii messaggi di risposta di controllo al mittente ossia degli ACK e NACK;

-Ritrasmissione: un pacchetto che è ricevuto con errore al ricevente sarà ritrasmesso dal mittente.

Il destinatario possiede un singolo stato. All'arrivo del pacchetto, risponde con un ACK o un NACK in base al fatto se il pacchetto è corrotto o meno.

Il mittente, invece, possiede 2 stati:

1. nel primo si aspetta il dato che venga passato dal livello superiore. Quando ciò avviene, tramite rdt\_send(data), si crea il pacchetto sndpkt contenente il dato da inviar e con un pacchetto di checksum udt\_send(sndpkt);
2. Nel secondo stato il mittente sta aspettando un ACK o un NAK dal destinatario:

-se il pacchetto viene ricevuto, il destinatario invia un ACK e il mittente saprà che il pacchetto è stato ricevuto correttamente e inoltre il protocollo ritorna nello stato di attesa per il dato dal livello superiore;

-se il pacchetto non viene ricevuto, il destinatario invia un NACK e il protocollo ritrasmette l'ultimo pacchetto e aspetta un ACK o NACK che deve ritornare dal destinatario.

Quando il mittente aspetta un ACK o un NACK, non può prendere altri dati dal livello superiore e per questo rdt2.0 viene detto protocollo Stop&Wait.

Il difetto del protocollo rdt2.0 è che non abbiamo pensato se l'ACK o il NACK è corrotto. Per risolvere questa problematica, occorre aggiungere un campo checksum per rilevare questi errori.

Dobbiamo capire come recuperare agli errori negli ACK/NACK:

• Aggiungere abbastanza bits di checksum per permette al mittente non solo di rilevare, ma anche di recuperare dagli errori di bit;

• Il mittente invia il pacchetto corrente quando riceve un ACK o NACK modificato. Questo approccio introduce pacchetti duplicati nel canale mittente-destinatario. Non è ottimale poiché il destinatario non sa se l'ACK/NACK precedente è stato già inviato.

**Reliable Data Transfer over a Channel with Bit and ACK Errors: rdt2.1**

Una soluzione al problema è quella di aggiungere un nuovo campo al pacchetto dati ed avere il numero del pacchetto del mittente in aggiungendo una sequenza di numeri (0-1). Il destinatario a questo punto devo solo controllare questa sequenza di numeri per determinare se il pacchetto ricevuto è una ritrasmissione o meno.

In un protocollo stop-&-wait, una sequenza di 1 bit è sufficiente, dato che permetterà al destinatario di sapere se il pacchetto era una ritrasmissione o uno nuovo.

Il mittente si basa sul concetto del “nel dubbio rinvio”: se ricevo un ACK/NACK alterato, il mittente a prescindere rinvia il pacchetto precedente e a quel punto se era un ACK il destinatario scarta il pacchetto perché è un duplicato altrimenti se era un NACK il destinatario si salva il pacchetto e manda un ACK (il fatto di essere un duplicato o no lo capisco dal dato 0-1 aggiunto al pacchetto).

**Reliable Data Transfer, Bit and ACK Errors, NACK-free: rdt2.2**

Un'alternativa è quella di inviare un ACK, al posto di un NACK, dando sempre il feedback sull'ultimo pacchetto correttamente ricevuto: un mittente che riceve 2 ACK per lo stesso pacchetto sa che il destinatario non ha ricevuto correttamente il pacchetto seguente a quello per cui ha ricevuto il doppio ACK. I protocolli di questo genere vengono detti rdt2.2. In particolare, si aggiunge una sequenza di numeri dei pacchetti che vengono riconosciuti con un messaggio ACK e il mittente ora deve controllare il numero di pacchetto controllando l'ACK.

**Reliable Data Transfer over a Channel with Bit Errors: rdt3.0**

Supponiamo ora che oltre alla corruzione di bit, il canale sottostante può perdere i pacchetti. Quindi, dobbiamo sapere come rilevare la perdita di pacchetti e cosa fare quando ciò accade.

Ci sono vari approcci per gestire la perdita di pacchetti. Supponiamo che il mittente trasmette i dati di un pacchetto e quel pacchetto o il suo ACK viene perso. In entrambi i casi, al mittente non perviene nessuna risposta dal destinatario.

Nella pratica si sceglie un tempo prestabilito per cui la perdita del pacchetto non è mai avvenuto: se non si riceve un ACK in questo arco di tempo, il pacchetto viene ritrasmesso.

Il mittente potrebbe decidere di inviare di nuovo il pacchetto anche in caso di assenza di corruzioni, introducendo la possibilità di duplicati, ma che vengono gestiti da rdt2.2.

Le prestazioni di questo protocollo non sono buone perché la maggior parte del tempo lo impiego ad attendere una risposta (ACK) dal reciever e dunque il protocollo limita l’uso delle risorse.

**3.6 Pipelined Protocol**

Il protocollo rdt3.0 è un protocollo corretto funzionalmente, ma non nelle sue perfomance. Il problema è che è un protocollo stop-and-wait.

Consideriamo il caso in cui due host si devono trasmettere dei pacchetti. Supponiamo di conoscere il ritardo dovuto alla propagazione della luce tra i due end system RTT, la dimensione dei pacchetti L (compresi di campi di intestazione e dati) e la velocità del mezzo trasmissivo. Il tempo di trasmissione del pacchetto nel link vale:



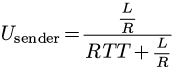
Dalla parte del ricevente quando gli è arrivato l'ultimo bit del pacchetto è passato un tempo pari a:

Assumiamo, inoltre, per semplicità che i pacchetti ACK sono estremamente piccoli e che il ricevitore può mandare un ACK appena l'ultimo bit del pacchetto è ricevuto, il tempo impiegato da questo pacchetto da destinatario a sorgente è:



A questo punto il mittente può trasmettere il prossimo messaggio.

Definiamo l'utilizzazione del mittente come la frazione di tempo in cui il mittente è attualmente impegnato ad inviare bits nel canale:



Nelle nostre assunzioni abbiamo evitato di introdurre i ritardi di accodamento durante il tragitto fra gli host, il ritardo di processamento tra mittente e destinatari. Di conseguenza si riscontra un aumento del tempo per inviare un pacchetto. La soluzione a questo problema introdotto dai protocolli di tipo stop-and-wait viene risolto permettendo al mittente di inviare molteplici pacchetti senza aspettare i rispettivi ACK, così facendo si aumenta notevolmente l'utilizzazione del canale.

Dato che i pacchetti in transito possono essere visti in coda uno con l'altro, si definisce questa tecnica come **pipelining**. Le conseguenze di questa tecnica sono le seguenti:

• Il range dei numeri di sequenza deve essere aumentato, dato che in ogni pacchetto in transito ci deve essere un unico numero di sequenza;

• Il mittente e il destinatario del protocollo potrebbero avere la necessità di bufferizzare più di un pacchetto.

• Il range dei numeri di sequenza necessari e i requisiti di bufferizzazione dipenderanno sul modo in cui il protocollo di trasferimento dati risponde alle perdite, corruzioni e ai ritardi dei pacchetti.

I due approcci principali verso la tecnica del pipelining sono: GO-Back-N(GBN) e Selective Repeat (SR).

**Confronto generale GBN e SR**

GBN:

1. Nel protocollo GBN, al mittente è concesso trasmettere molteplici pacchetti senza aspettare per un ACK, ma è vincolato ad avare non più di qualche numero N di pacchetto non riscontrati da un ACK in pipeline.
2. Il reciever invia ACK di tipo cumulativo ossia ho ricevuto correttamente tutti i pacchetti fino a quello dell’ACK
3. Il sender ha un solo timer legato al pacchetto più vecchio non ancora riscontrato: se il timer termina allora ritrasmette tutti i pacchetti non riscontrati

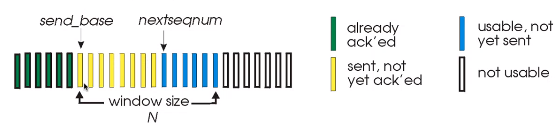
SR:

1. Nel protocollo SR, come nel GBN, al mittente è concesso trasmettere molteplici pacchetti senza aspettare per un ACK, ma è vincolato ad avare non più di qualche numero N di pacchetto non riscontrati da un ACK in pipeline.
2. Il reciever invia ACK di tipo individuale ossia ogni ACK corrisponde ad un solo pacchetto
3. Il sender ha un timer per ogni pacchetto: se il timer termina allora ritrasmette quel pacchetto

**Go-Back-N(GBN)**

Nel protocollo GBN posso trasmettere fino a N pacchetti senza aspettare un ACK e indico con K il numero di bit usato per numerare i pacchetti.

Se definiamo “base” il numero di sequenza del più vecchio pacchetto non riconosciuto e “nextseqnum” come il più piccolo numero di sequenza non usato, allora possiamo identificare quattro intervalli di numeri di sequenza:

I numeri di sequenza nell'intervallo [0,base-1] corrispondono ai pacchetti che sono già stati inviati e riconosciuti. L'intervallo [base, nextseqnum-1] corrisponde ai pacchetti che sono già stati inviati ma non ancora riconosciuti. La sequenza nell'intervallo [nextseqnum,base+N-1] può essere usato per i pacchetti che possono essere inviati immediatamente, i dati dovrebbero arrivare dal livello superiore. Infine, i numeri di sequenza più grandi o uguali a base+N non possono essere usati finché un pacchetto non riconosciuto è ancora nella pipeline.

L'intervallo dei numeri ammissibili di sequenza per i pacchetti trasmessi ma non ancora riconosciuti può esser visto come una finestra di dimensioni N. Quando il protocollo opera, questa finestra scorre in avanti sullo spazio dei numeri di sequenza e per questa ragione GBN viene detto protocollo sliding-window.

Il mittente deve rispondere a tre eventi:

• Invocazione dall'alto: quando rdt\_send() è chiamato dal livello superiore, il mittente controlla se la finestra è piena. Se la finestra non è piena, si crea un pacchetto e lo si invia e le altre variabili vengono aggiornate. Se la finestra è piena, il mittente ritorna il dato al livello superiore con un'indicazione che la finestra è piena. A questo punto il livello superiore dovrà riprovare in seguito;

• Ricezione di un ACK: in GBN, un ACK per un pacchetto con numero di sequenza n sarà un ACK cumulativo per indicare che tutti i pacchetti con numero di sequenza fino ad n (incluso) sono stati correttamente ricevuti dal destinatario;

• Eventi di timeout: si utilizza un timer per recuperare la perdita di dati o di ACK. Se scade il tempo, il mittente invia nuovamente tutti i pacchetti che sono stati precedentemente inviati.

Nel caso in cui si riceve un ACK ma ci sono ancora pacchetti trasmessi ma non riconosciuti, il timer viene resettato. Se non ci sono pacchetti non riconosciuti, il timer si ferma.

Il ricevente deve rispondere al seguente evento:

* se un pacchetto con numero di sequenza n viene ricevuto correttamente ed è in ordine, il destinatario invia un ACK per il pacchetto n e consegna la porzione di dato del pacchetto al livello superiore; in tutti gli altri casi, il destinatario scarta il pacchetto ed invia una ACK per il pacchetto più recente che ha ricevuto in ordine.

Ricordiamo che il ricevitore deve consegnare il dato in ordine al livello superiore quindi non ha bisogno di memorizzarsi in un buffer i pacchetti fuori sequenza (per esempio si perde il pacchetto n e ricevo l’n+1) perché tanto verranno ritrasmessi entrambi successivamente.

**Selective Repeat (SR)**

Il protocollo GBN permette al mittente di riempire la pipeline con i pacchetti, inoltre permette di evitare i problemi di utilizzazione del canale. Tuttavia, questo protocollo soffre di problemi di performance: quando il prodotto della dimensione della finestra e del ritardo di banda diventa grande, ci potrebbero essere molti pacchetti in pipeline. Un errore di un singolo pacchetto potrebbe causare la ritrasmissione di un numero grande di pacchetti, molti di questi non necessari; all'aumentare della probabilità degli errori di canale, la pipeline si riempie di ritrasmissioni non necessarie.

Il protocollo selective-repeat evita ritrasmissioni non necessarie tramite il mittente che ritrasmette solo quei pacchetti che sono stati ricevuti erroneamente. Questa individualità richiede che il destinatario riconosca individualmente i pacchetti ricevuti. Si avrà quindi una finestra di N per limitare i pacchetti non riconosciuti nella pipeline e il mittente avrà ricevuto ACK per alcuni pacchetti in questa finestra.

Il ricevitore, inoltre, riconoscerà un pacchetto correttamente ricevuto sia se in ordine o meno. Nel caso in cui non è in ordine, il pacchetto viene bufferizzato fino a che qualsiasi altro pacchetto venga ricevuto e poi inviato al livello superiore.

Il mittente deve rispondere a 3 tipi di eventi:

• Dati ricevuti dall'alto: quando un dato è ricevuto dall'alto, il mittente controlla il prossimo numero di sequenza disponibile per il pacchetto. Se la sequenza è all'interno della finestra del mittente, il dato viene pacchettizzato ed inviato; altrimenti viene bufferizzato e rispedito al livello superiore;

• Timeout: I timer vengono usati per proteggere la perdita di dati. Ogni pacchetto deve avere il proprio timer logico dado che solo un pacchetto sarà trasmesso al timeout;

• ACK ricevuto: Se si riceve un ACK, il mittente lo segna come ricevuto. Se il numero di sequenza del pacchetto è uguale a send\_base, la finestra viene mossa in avanti al pacchetto non riconosciuto con il più piccolo numero di sequenza. Se la finestra si muove e ci sono pacchetti non trasmessi con numeri di sequenza che ora cadono all'interno della finestra, vengono trasmessi.

Il destinatario deve risponde anche lui a 3 tipi di eventi:

• Pacchetti con numero di sequenza in [rcv\_base,rcv\_base+N-1] correttamente ricevuti: In questo caso, il pacchetto ricevuto cade all'interno della finestra del ricevente e si ritorna un ACK al mittente. Se il pacchetto non era precedentemente ricevuto, viene bufferizzato. Se il pacchetto ha un numero di sequenza uguale al numero di base della finestra di ricezione, allora questo pacchetto e tutti quelli precedenti sono bufferizzati e numerati consecutivamente ed inviati al livello superiore. A questo punto la finestra di ricezione viene mossa in avanti dal numero dei pacchetti consegnati al livello superiore;

• Pacchetti con numero di sequenza in [rcv\_base-N,rcv\_base-1] correttamente ricevuto: In questo caso, si genera un ACK, anche se il pacchetto è stato riconosciuto precedentemente;

• Altrimenti: il pacchetto viene ignorato.

Il ricevente riconosce nuovamente i pacchetti già ricevuti con un certo numero di sequenza inferiore al numero base della finestra corrente. Questo perché, se non ci fosse ACK per il pacchetto send\_base che si propaga dal ricevente al destinatario, il mittente potrebbe inviare il pacchetto send\_base anche se il destinatario l'ha già ricevuto.

Se il ricevente non riconoscesse questo pacchetto, la finestra del mittente non scorrerebbe mai in avanti. Il mittente e il destinatario non hanno necessariamente la stessa visione di quello che sta accadendo.

**3.7** **Connessione TCP**

-TCP è un protocollo punto-punto ossia per la comunicazione tra un mittente ed un receiver.

-TCP è pensato per una consegna in ordine di uno stream di byte ossia permette di inviare una sequenza di byte tra mittente e destinatario ossia non esiste il concetto di messaggio ma solo di byte.

-TCP è pipelined ossia ragiona su una finestra con dimensione che dipende dall’algoritmo di controllo di flusso e congestione.

-TCP si basa su buffer sia per il sender che per il receiver.

-TCP è detto connection-oriented poiché prima che un processo di un'applicazione può iniziare ad inviare i dati da un altro processo di un'altra applicazione, entrambi i processi devono prima effettuare un handshake inviandosi preliminarmente segmenti tra loro per stabilire una connessione affidabile.

-Ricordiamo, inoltre, che il protocollo è in esecuzione solo negli end-systems e a non negli elementi di rete intermediari e quindi non possiede il concetto di stato.

-TCP fornisce un servizio full-duplex: se vi è una connessine tra due processi, allora i dati delle applicazioni possono essere inviati da un processo all'altro e viceversa con la stessa connessione. Inoltre, esiste una Maximum Segment Size (MSS) ossia la dimensione massima di un segmento.

-TCP ha il controllo di flusso ossia posso inviare dati a qualsiasi velocità senza però sovraccaricare il buffer del receiver.

Trasmissione dati:

Una volta ricevuti i dati dal livello superiore, si crea un segmento con il numero di sequenze in base al numero di byte stream associato al primo byte del segmento e si fa partire un timer per quel segmento. Una volta scaduto il timeout si ritrasmette il pacchetto e si fa ripartire il timer. Una volta ricevuti degli ACK cumulativi, faccio scorrere la finestra.

**Struttura Segmento TCP**

Il segmento TCP è formato dai campi di intestazione e i campi dati, che contiene una porzione dei dati dell'applicazione. La struttura di un segmento TCP è la seguente:

• Due intestazioni da 16 bit contenenti i numeri di porta di sorgente e destinazione utilizzato per multiplazione e demultiplazione;

• campo checksum;

• Un campo di numero di sequenza di 32-bit e un campo di numero di ack da 32-bit usati dal mittente e dal destinatario TCP per realizzare un servizio di trasferimento affidabile di dati;

• Un campo di finestra di ricezione di 16 bit usato per il controllo di flusso;

• Un campo di lunghezza header di 4-bit che specifica la lunghezza di una intestazione TCP. Questo poiché l'intestazione TCP può avere una lunghezza variabile in base alle opzioni che si selezionano.

• Un campo opzioni di lunghezza variabile e facoltativo usato quando un mittente e un destinatario negoziano sulla grandezza massimo del segmento o per motivi di scalabiltà di rete;

• Un campo flag di 6-bit:

1. Il bit ACK è usato per indicare che il valore nel campo ACK è valido, cioè che il segmento contiene un ACK per un segmento correttamente ricevuto;
2. RTS, SYN, FIN sono usati per il setup della connessione e la sua chiusura.
3. CWR e ECE sono usati per la notifica esplicita di congestione della rete;
4. PSH indica che il destinatario dovrebbe passare il dato al livello superiore immediatamente;
5. URG è usato per indicare che il dato all'interno del segmento è stato marcato come urgente;

• Un campo Urgent Data Pointer indica la posizione dell'ultimo byte del dato segnato come urgente.

**Numeri di sequenza e numeri di ACK**

I campi più importanti di un segmento TCP sono il campo di numero di sequenza e il campo di numero acknowledgment e sono fondamentali per la realizzazione di servizio di trasporto affidabile di dati.

TCP vede i dati come uno stream di byte non strutturati, ma ordinati. Il numero di sequenza per un segmento è dunque il numero nel flusso di byte del primo byte del segmento.

Ricordiamo, inoltre, che TCP è full-duplex, quindi A potrebbe ricevere dati da B mentre sta inviando dati a B. Ognuno di questi segmenti posseggono un numero di sequenza per il dato che si sta trasmettendo. Il numero di ACK che A mette nel suo segmento è il numero di sequenza del prossimo byte che sta aspettando A da B.

Poiché TCP riconosce byte solo fino al primo byte mancante nello stream, TCP fornisce ACK cumulativi e in particolare ACK-n vuol dire ho ricevuto tutti i byte fino a quello n-1.

Nel caso in cui si ricevano segmenti fuori ordine, si possono prendere due strade:

• Il destinatario scarta i segmenti fuori ordine;

• Il destinatario mantiene i byte fuori ordine e aspetta i byte mancanti per colmare i buchi.

**Stima RTT e Timeout**

TCP utilizza un meccanismo di timeout/ritrasmissione per recuperare la perdita dei segmenti. Il timeout dovrebbe essere grande quanto l'RTT da quando il segmento viene inviato finché non viene riconosciuto (ACK). Se il timer è troppo piccolo (più piccolo del RTT) c’è rischio di non aver ricevuto un riscontro e quindi potrei rinviare un segmento anche se non c’è bisogno. Se il timer è troppo grande causa una reazione molto lenta quando viene perso un pacchetto.

Iniziamo lo studio della gestione dei timer TCP considerando come TCP stima l'RTT.

Il round-trip-time SampleRTT per un segmento è il tempo che intercorre dall'invio del segmento alla ricezione dell'ACK. Invece di misurare un SampleRTT per ogni segmento trasmesso, lo si calcola istantaneamente per un pacchetto a cui non corrisponde ancora un ACK. Questo valore varia in base al segmento a causa della congestione nei router e dal carico variabile degli end-system, dunque, si vuole considerare la media di questo valore detto EstimatedRTT. Si ottiene nel seguente modo (filtro passa basso):



Questo valore è una media pesata di SampleRTT, in cui si mette maggiore peso sui campioni recenti piuttosto che quelli vecchi. Questo perché i campioni recenti rappresentano maggiormente la congestione attuale della rete (alpha=0.125).

Inoltre, è utile anche introdurre un parametro che considera la variabilità dell'RTT detto DevRTT. Questo valore, in particolare, rappresenta di quando SampleRTT si discosta da EstimatedRTT:



Ossia quanto dista in valore assoluto il valore osservato rispetto alla stima della sua media (beta=0.25).

Avendo ora una stima del valore medio (EstimatedRTT) e una stima di quanto sia la variabilità dei ritardi (DevRTT), TCP imposta come valore del timeout:



**Trasferimento Affidabile di Dati**

Ricordiamo che il servizio del livello di rete di Internet è inaffidabile: IP non garantisce la consegna del datagramma, non garantisce la consegna in ordine dei datagrammi e non garantisce l'integrità del dato nei datagrammi.

TCP crea un servizio affidabile di dati al di sopra del protocollo IP, assicurandosi che il flusso dei dati che fuoriesce da TCP non è corrotto, non presenta mancanze, non vi siano duplicazione e che sia in sequenza: in poche parole il flusso di byte in uscita deve essere uguale a quello in entrata dall'altra parte della connessione.

TCP risponde a tre tipi di eventi:

1.Quando TCP riceve un dato dall'applicazione, lo incapsula in un segmento e lo passa al protocollo IP. Se il Timer non è in esecuzione per qualche altro segmento, TCP inizializza il timer quando lo passa ad IP. La scadenza del timer è il TimeoutInterval.

2.Quando TCP risponde ad un evento di timeout, il protocollo ritrasmette il segmento e riavvia il timer;

3.Quando TCP riceve un ACK dal destinatario, TCP confronta il valore dell'ACK y con SendBase. La variabile di stato TCP SendBase è il numero di sequenza del più vecchio byte non riconosciuto. Poiché gli ACK in TCP sono cumulativi, y riconosce come ricevuti tutti i segmenti prima del numero di byte y. In caso contrario i segmenti non sono stati riconosciuti.

A questo punto si aggiorna la variabile SendBase e si riavvia il timer se ci sono segmenti non ancora riconosciuti.

**Flow Control**

Ricordiamo che l'host in ogni lato della connessione TCP possiede un buffer per la connessione. Quando la connessione TCP riceve dei byter che sono corretti e in sequenza, li posiziona nel buffer di ricezione. Poiché il buffer è di dimensione finita, se l'applicazione è lenta a leggere i dati, il mittente può facilmente portare il buffer del destinatario in overflow. A questo scopo TCP fornisce un servizio di controllo di flusso alla sua applicazione per eliminare la possibilità del mittente di colmare il buffer del destinatario. Nella pratica consiste nel far combaciare la velocità con cui si inviano i dati a quella a cui vengono letti.

Quindi, TCP fornisce flow control assegnando al mittente una variabile detta receiver window: viene usata per dare al mittente un'idea di quando spazio libero è disponibile al destinatario. Dato che TCP è full-duplex, il mittente ad ogni lato della connessione mantiene un distinto receiver window. In questo modo si evita che il mittente invii dati al ricevente quando non ha posto in cui metterli.

Supponiamo che un host A sta inviando un file di grandi dimensioni ad un Host B su una connessione TCP. L'host B alloca un buffer di ricezione nella sua connessione, definendo la dimensione RcvBuffer. In qualche istante di tempo, il processo dell'applicazione nel destinatario legge dal buffer. Definiamo le seguenti variabili:

• LastByteRead: è il numero dell'ultimo byte nel flusso dei dati letti dal buffer da parte del processo dell'applicazione B;

• LastByteRcvd:è il numero dell'ultimo byte nel flusso dati che è arrivata dalla rete ed è stata salvata nel buffer di ricezione;

Dato che TCP non permette l'overflow del buffer allocato, dobbiamo avere:



Definiamo la receive window come la dimensione rimanente nel buffer:

Questa variabile è dinamica nel tempo. L'host B dice all'host A quanta spare room possiede nel buffer della connessione salvando in ogni segmento la variabile rwnd.

Inizialmente, l'host B imposta:

rwnd=RcvBuffer

HostA segue due variabili, LastByteSent e LastByteAcked. Mantendo la quantità dei dati non ancora riconosciuti minore di rwnd, l'host A si assicora che non sta “overflowando” il buffer di destinazione nell'host B. In particolare, imposta il suo thorughput come segue:



**Fast Retrasmit**

Uno dei problemi delle ritrasmissioni è che il periodo di timeout potrebbe essere molto lungo. Quando un segmento è perso, questo lungo periodo di timeout forza il mittente a ritardare il reinvio del pacchetto perso, aumentando conseguentemente il ritardo. Il mittente spesso può rilevare la perdita del pacchetto prima che l'evento di timeout avviene tramite gli ACK duplicati.

L'ACK duplicato è un ACK di un segmento che è stato già ricevuto dal mittente.

Quando il destinatario riceve un segmento con un numero di sequenza che è maggiore del prossimo numero di sequenza atteso in ordine, rileva una mancanza nel flusso dei dati: rileva il segmento mancante. Questa mancanza potrebbe essere il risultato di una perdita o un riordino dei segmenti all'interno della rete. Inoltre, ricordiamo che TCP non utilizza NACK, il destinatario non può inviare un NACK esplicito al mittente; tuttavia, manda un ACK duplicato che corrisponde all'ultimo byte ricevuto in ordine.

Poiché il mittente invia spesso un grande numero di segmenti, se uno di questi viene perso, ci saranno molti ACK duplicati. Se il mittente TCP riceve 3 ACK duplicati per lo stesso dato, considera come se il segmento seguente a quello di cui abbiamo riscontrato 3 ACK duplicato è perso. In questo casto, il mittente TCP effettua il cosiddetto fast retrasmit: ritrasmette il segmento prima che il timer scada.

**TCP Connection Management**

La gestione della connessione TCP può aumentare significatamene i ritardi percepiti. Supponiamo che un processo in esecuzione in un host vuole iniziare una connessione con un altor processo di un altro host. Il processo dell'applicazione client inizialmente informa il client TCP che vuole stabilire una connessione verso un processo nel server. TCP client allora inizia a stabile una connessione TCP con il server TCP seguendo una procedura detta handshake a tre vie:

1. La parte client TCP inizia inviando un segmento speciale TCP al lato server. Questo segmento non contiene alcun dato del livello applicazione, ma un flag nell'intestazione del segmento SYN=1. Quindi il client sceglie casualmente un numero di sequenza e lo inserisce nel campo numero di sequenza del segmento SYN. Questo segmento viene incapsulato all'interno di un datagramma IP ed inviato al server;
2. Una volta che il datagramma IP contenente il segmento TCP SYN arriva al server, quest'ultimo lo estrae dal datagramma e alloca il buffer TCP e le variabili della connessione. A questo punto invia un segmento di connessione approvata al client TCP, privo di dati del livello applicativo, che contiene nell’intestazione tre informazioni importanti: il bit SYN impostato ad 1, il campo ACK è impostato a client\_isn+1 ed il proprio numero di sequenza lo mette nel campo di intestazione opportuno nel segmento. Questo segmento è detto SYNACK segment.
3. Alla ricezione del segmento del passo 2, il client alloca i buffer e le variabili alla connessione. Il client, infine, invia al server un altro segmento con SYN = 0, dato che la connessione è stata stabilita. Questo terzo stato potrebbe contenere dati di livello applicativo. Al conseguimento di questi tre passi, il client e il server possono inviare i segmenti contenenti dati tra di loro. Il flag SYN è impostato a 0.

Durante la vita della connessione TCP, il protocollo TCP in esecuzione in ogni host effettua transizioni attraverso vari stati TCP.

Il client:

• Inizia nello stato CLOSED.

-L'applicazione del lato client inizia una nuova connessione TCP. Questo causa l'invio di un segmento SYN al server.

• Il client arriva nello stato SYN\_SENT.

-In questo stato il client aspetta un segmento dal server TCP che include un ACK per il segmento precedenti e che possegga un SYN bit =1. Una volta ricevuto cambia stato.

• Il client arriva nello stato ESTABILISHED.

-Il TCP client ora può inviare e ricevere segmenti contenenti dati del livello applicativo.

-Il client può decidere a questo punto di chiudere la connessione. In questo caso il client TCP invia un segmento con bit FIN=1

•Il client TCP entra nello stato FIN\_WAIT\_1 e attende dal server un ACK e una volta ricevuto entra nel FIN\_WAIT\_2 e attende un altro segmento con bit FIN=1. Quando lo riceve manda un ACK e cambia stato:

•Il client TCP entra nello stato TIME\_WAIT in cui invia l’ultimo ACK e rimane in wait (dai 30 sec ai 2 minuti). A questo punto la connessione viene chiusa.

**Principi di controllo di congestione**

Il problema del controllo della congestione consiste nell’avere troppi pacchetti nella rete che non riesco a gestire ossia ho un overflow dei buffer nei router quando il traffico in rete diventa eccessivo. Inoltre, la ritrasmissione dei pacchetti persi contribuisce ancora di più ad aumentare la congestione causando così il collasso della rete (nel controllo di flusso mi preoccupa che il noto receiver abbia buffer a disposizione mentre qui mi preoccupa dei buffer dei router). Per evitare di far congestionare la rete esistono diversi approcci che differiscono per il ruolo che ha la rete nel supportare il controllo di congestione:

1. Controllo di congestione assistito dalla rete:

Il controllo è assistito o gestito dalla rete ossia sono i router forniscono informazioni per poter garantire che gli utenti inviino una quantità di traffico compatibile con la capacità della rete di inviare traffico in quel momento. Queste informazioni possono essere un singolo bit che dice c’è o non c’è congestione (TCP/IP ECN, ATM) oppure direttamente la frequenza di trasmissione che il router supporta in quel momento (ATM ABR):

ATM ABR è un servizio di tipo elastico per il trasferimento dati ossia se la rete è sottoutilizzata allora ABR si avvantaggia usando tutta la larghezza di banda disponibile mentre se è congestionata ABR fa diminuire il traffico trasmissivo diminuendo la velocità di trasmissione.

Vediamo come viene implementato nel contesto del funzionamento delle reti ATM:

Un mittente invia dati verso un destinatario sotto forma di celle, dove quelle bianche contengono dati mentre quelle rosse (RM) sono generiche per la gestione delle risorse. Il destinatario rinvia indietro le celle rosse mettendoci all’interno delle informazioni, che possono essere inserite anche dagli switch in modo che la sorgente capisce la velocità a cui bisogna inviare dati. Vediamo tre meccanismi per segnalare al destinatario informazioni relative alla congestione della rete:

-Bit EFCI: nelle celle c’è un bit che viene impostato ad 1 da uno switch se lo switch è congestionato. Arrivato a destinazione, se una cella dati che precede una cella RM ha il bit impostato ad 1, allora la destinazione rinvia indietro una cella impostando ad 1 un bit per dire che c’è congestione.

-Due bit NI/CI: uso due bit nelle celle RM che se posti uguali a 1 indicano: NI che lo switch è leggermente congestionato quindi non incrementare la velocità, CI che c’è congestione quindi riduci la velocità.

-Due byte ER: uso due byte nelle celle RM in cui lo switch indica la velocità massima a cui lui permetteva di inviare traffico.

2.Controllo di congestione end-to-end:

le entità di trasporto di livello 4 agli estremi della rete devono gestire il controllo da estremo ad estremo, usando il fatto che la rete non ha meccanismi espliciti per indicare la congestione. Poiché quindi la rete non mi dice quando c’è congestione, capisco che c’è congestione quando perdo segmenti e di conseguenza TCP diminuisce l’ampiezza della propria finestra.

**Controllo di congestione TCP end-to-end**

Il problema di limitare la velocità a cui invio i dati, TCP lo risolve manipolando la dimensione della receiver window. Deve sempre valere la seguente disuguaglianza:

LastByteSent-LastByteAcked<=min(CongWin,RcvWindow)

Ossia il numero di byte che posso avere in rete inviati ma non ancora riscontrati sia minore o uguale del minimo tra la finestra di ricezione e la finestra di congestione, ossia la quantità di dati che posso avere in rete inviati ma non riscontrati in base all’algoritmo di controllo di congestione. Ricordiamo che la velocità di invio di dati inviati è pari a:

Rate=

Ossia la velocità è proporzionale alla dimensione della finestra di congestione diviso il RTT. In TCP il CongWin non è costante ma è funzione della percezione del livello di congestione della rete.

Per capire se c’è congestione vedo se ci sono stati perdite di segmenti tramite o un timeout o 3 ACK duplicati: in questo caso il TCP riduce la velocità di trasmissione (rate) riducendo la dimensione della finestra di congestione. L’algoritmo per gestire la congestione presenta 3 fasi:

1. Slow Start: quando una connessione TCP comincia, la dimensione della finestra di congestione è minima, pari a 1MSS (pari ad un segmento). Avere una finestra così piccola vuol dire avere un rate bassissimo. Inizia molto lentamente ma aumentare il rate velocemente (esponenziale) e questo approccio conservativo è dato dal fatto che una nuova connessione perturba lo stato della rete quindi meglio iniziare a trasmettere lentamente. Per aumentare in maniera esponenziale il rate, l’idea è la seguente:

ogni volta che invio un segmento e ricevo il suo ACK, aumento la dimensione della finestra di 1, dunque la finestra raddoppia ogni RTT (invio un segmento, ricevo 1 ACK, aumento di 1 la dimensione della finestra -> ora invio 2 segmenti, ricevo 2 ACK, aumento di 2 la dimensione della finestra-> ora invio 4 segmenti, ricevo 4 ACK, aumento di 4 la dimensione della finestra).

1. AIMD: caratterizza lo stato di una connessione TCP a regime. Dopo una fase iniziale, TCP ha un andamento della finestra che cresce linearmente nel tempo: la dimensione aumento di 1 ogni RTT se tutto va bene ma quando TCP si accorge di una perdita tramite 3 ACK duplicati allora dimezza la dimensione della finestra (comportamento a dente di sega). TCP ha un comportamento conservativo, invece, quando la perdita viene rilevata con un timeout che scade. TCP prende la finestra e la pone uguale a 1 e ricomincia con la fase di Slow Start. Questo perché ricevere 3 ACK duplicati vuol dire che ho perso qualcosa ma i segmenti successivi sono arrivati mentre se c’è un timeout vuol dire che non ci sono i 3 ACK e quindi ho perso tanti segmenti.

Il passaggio dalla velocità esponenziale alla velocità lineare è indicato dalla variabile di Threshold. All’inizio, quando non ho ancora perso niente, il Threshold è impostato dal protocollo, in generale il Threshold è pari alla metà della dimensione della finestra prima della perdita precedente.

1. Fast recovery: se la perdita è tramite 3 ACK, in realtà TCP pone la dimensione della finestra pari a Threshold+3 ed entro nella fase Fast recovery per recuperare velocemente e tornare nella fase di Congestion-Avoidance.

Quindi quando CongWin è minore del Threshold, il mittente è nella fase di Slow Start e la finestra cresce esponenzialmente.

Quindi quando CongWin è maggiore del Threshold, il mittente è nella fase di Congestion-Avoidance e la finestra cresce linearmente.

Se ho delle perdite tramite 3 ACK, il Threshold è impostato a CongWin/2 e la finestra impostata al valore del Threshold+3, se ho delle perdite tramite un timeout, il Threshold è impostato a CongWin/2 e la finestra impostata a 1.

**Versioni TCP**

TCP ha avuto una forte evoluzione nel tempo. Inizialmente nella versione TCP Tahoe, non c’era il meccanismo di ritrasmissione veloce e le perdite veniva apprese solo tramite il timeout. Nella versione successiva, vengono implementati i meccanismi appena descritti.

**Throughput**

Il throughput ossia la quantità di dati che invia per un’unità di tempo (la portata) è proporzionale alla dimensione della finestra di congestione e inversamente proporzionale al RTT. Suppongo una condizione ideale in cui ci siano sempre dati da inviare e che non c’è la fase di slow start, allora la dimensione della finestra è periodica e assume la forma del “dente di sega”. La relazione tra la dimensione della finestra, il suo valore medio e il RTT corrisponde al throughput medio che è pari a:

TCP throughput=

**Fairness**

Consideriamo K connessioni TCP, ognuna dei quali con un differente path end-to-end, ma che attraversano un link di collo di bottiglia con rate di trasmissione R bps. Supponiamo che ogni connessione stia inviando un grande file e non vi è traffico UDP attraverso il link del collo di bottiglia. Un meccanismo di controllo di congestione è detto fair se la media della velocità di trasmissione di ogni connessione è approssimativamente R/K cioè che ogni connessione condividere la stessa capacità di banda. TCP tende ad essere fair cioè condivide in modo uguale la capacità di banda disponibile tra le diverse sessioni e questo è vero solo se il RTT delle diverse sessioni sono simili.

Consideriamo il caso più semplice in cui due connessioni TCP condividono un singolo link con velocità di trasmissione R. Assumiamo che due connessioni hanno lo stesso MSS e RTT, che hanno un grande quantitativo di dati da inviare e che non vi sono altre connessioni TCP o datagrammi UDP che attraversano il link condiviso. Infine, ignoriamo lo stato slow-start di TCP e assumiamo che le connessioni TCP stiano operando in AIMD tutte le volte.

Se TCP condivide la capacità di banda equamente tra le due connessioni allora il throughput dovrebbe corrispondere a una freccia a 45 gradi partendo dall'origine. Idealmente, la somma di tutti i throughput è uguale a R. L'obiettivo da raggiungere è quello di avere i throughput che cadano da qualche parte vicino l'intersezione della capacità di banda della linea condivisa e l'utilizzazione totale della linea.

Supponiamo che la dimensione della finestra è tale che in ogni istante di tempo, le connessioni 1 e 2 realizzano il nostro obiettivo. Dato che l'ammontare della capacità di banda consumata dalle due connessioni è minore di R, non vi saranno perdite, entrambe le connessioni aumenteranno la propria finestra di congestione di 1 MSS per RTT. Si procederà quindi su una linea a 45 gradi partendo dall'origine.

Nel caso in cui, la capacità di banda del link consumata è maggiore di R, vi saranno perdite di pacchetti e quindi si diminuirà di un fattore 2 e così via.

Quindi la capacità di banda realizzata dalle due connessioni fluttua lungo la capacità di banda della linea condivisa e si avrà convergenza a prescindere dall'ipotesi di uno spazio bidimensionale. Da cui si può evincere il perché in TCP si condivide la stessa quantità di banda.