

Appunti corso di Reti

1 Reti di Calcolatori e Internet

1.1 Internet

Internet è una rete di calcolatori che connette miliardi di dispositivi in tutto il mondo. Questi sono chiamati **end systems**.

Gli end system sono collegati tra loro da una rete di **link** di comunicazione e packet switches. Quando un sistema deve inviare dati ad un altro sistema, il mittente segmenta i dati e aggiunge dei byte di header ad ogni segmento: questi sono chiamati packets. Un **packet switch** riceve un packet in arrivo su uno dei suoi link di comunicazione in entrata e lo inoltra su uno dei suoi link di comunicazione in uscita. La sequenza di collegamenti attraversati da un pacchetto è detto **route**.

Gli end system accedono ad Internet tramite gli Internet Service Providers (**ISPs**):

- Operatori locali, regionali : connessione “casalinga” (Tim, Vodafone, ...)
- Operatori nazionali, globali : connessione globale (Spark, ...)

Il canale su cui viaggiano i pacchetti determina la banda (velocità) di comunicazione.

Le reti sono composte da 2 parti:

- **Edge** : host e applicazioni che cambiano nel tempo
- **Core** : parte centrale della rete che non cambia spesso nel tempo

1.1.1 Protocolli

I sistemi implementano **protocolli** che controllano l'invio e la ricezione delle informazioni all'interno di Internet: i più importanti sono il Transmission Control Protocol (**TCP**) e Internet Protocol (**IP**). I protocolli definiscono il formato e l'ordine dei messaggi inviati e ricevuti e le azioni da prendere sulla trasmissione del messaggio.

Un ente che stabilisce gli standard è **IETF**, questo può approvare o no un **RFC** (richiesta per stabilire un protocollo).

Il SO fornisce delle interfacce di programmazione, queste servono alle applicazioni per non preoccuparsi di cosa succede ai livelli più bassi.

1.2 Il bordo della rete

Il bordo della rete è composto dagli end systems, divisi in due categorie: **clients** (richiede il servizio) e **servers** (fornisce il servizio).

Data center : luogo in cui sono ospitati i server. Le tecnologie del data center sono molto diverse da quelle domestiche ma vengono denominate entrambe edge

1.2.1 Reti d'accesso

La **rete d'accesso** è ciò che collega fisicamente un end system al **primo router** (edge router) che connette l'edge al core della rete. I due tipi di accesso a banda larga più diffusi sono:

Cable-based access:

- Viene introdotto un dispositivo **CMTS** alla fine del cavo (della rete televisiva) che comprende cosa è video e cos'è dati per mandare il video sul canale TV e i dati su internet.
- In **FDM** (frequency division multiplexing) il video viene trasmesso a certe frequenze più basse, e i dati internet a frequenze più alte.
- Con **HFC** (hybrid fiber coax) il cavo dallo splitter al CMTS è un ibrido tra fibra e coassiale, asimmetrico perché la velocità è diversa tra le due direzioni (Download 40Mbps-1.2Gbps, Upload 30-100Mbps) perché le richieste del client sono molto corte, invece le risposte dei

server possono essere molto lunghe. Questo canale è **condiviso**, tutte le case condividono un caso coassiale per entrare nel CMTS e la banda è divisa per il numero di utenti.

Digital subscriber line:

- Di solito l'accesso **DSL** è ottenuto dalla stessa compagnia telefonica che fornisce accesso telefonico. Ogni modem DSL usa la linea telefonica esistente per scambiare dati con il DSL multiplexer (DSLAM) locato nell'ufficio centrale della compagnia telefonica. Il modem casalingo converte dati digitali in toni analogici ad alta frequenza e li trasmette sulla linea telefonica. La velocità di trasmissione in downstream è più elevata di quella in upstream: si dice che l'accesso è **asimmetrico (ADSL)**. La velocità massima è limitata dalla compagnia telefonica e dalla distanza dell'ufficio centrale. Il mezzo non è condiviso, la velocità è garantita.

Un altro tipo di connessione è il fiber to the home (**FTTH**), che fornisce una connessione via fibra direttamente all'end user. Solitamente, la fibra in uscita dall'ufficio centrale è divisa relativamente vicino alle case. Ci sono due architetture per la distribuzione di fibra ottica: active optical networks (**AON**) e passive optical networks (**PON**).

Esiste inoltre il 5G fixed wireless che permette ad un modem di collegarsi senza cavo. Di solito, viene utilizzata una local area network (LAN) per collegare un end system all'edge router. La tecnologia d'accesso più utilizzata è **Ethernet**: gli utenti usano un doppino per connettersi ad un Ethernet switch, che a sua volta si connette ad Internet. Con la tecnologia wireless, gli utenti si collegano ad un access point.

Dispositivi mobili possono connettersi a Internet tramite wide-area wireless (3G, 4G, 5G).

1.2.2 Connessioni fisiche

Per ogni coppia mittente-ricevitore, i bit sono propagati via onde elettromagnetiche o impulsi ottici attraverso un mezzo fisico. Questi ricadono in due categorie: guidati (cavi) e non guidati (wireless). Il mezzo più popolare è il doppino in rame: i cavi sono intrecciati per ridurre interferenze. Sono di solito usati nelle LAN. I doppietti possono essere di categorie diverse, le quali garantiscono velocità di connessioni diverse.

Il cavo coassiale è composto da due conduttori in rame disposti in maniera concentrica. Più end systems possono essere connessi allo stesso cavo coassiale direttamente.

La fibra ottica è un mezzo sottile e flessibile che permette di raggiungere alte velocità di trasmissione, ma è ancora considerata troppo costosa per essere usata in connessioni locali.

I canali radio terrestri trasportano segnali nello spettro elettromagnetico e quindi non richiedono cavi, ma sono molto suscettibili alle interferenze.

I canali radio satellitari utilizzano satelliti in orbita per trasmettere segnali su microonde.

WiFi è un protocollo che stabilisce come utilizzare l'aria, invece Ethernet è un protocollo su come utilizzare il cavo fisico.

WiFi = standard 802.11 b/g/n (tipologia), per reti locali WLAN (wireless local area network)

Wide-area cellular access networks = 4/5g, per distanze lunghe (dipende dalla frequenza utilizzata)

1.3 Centro della rete

1.3.1 Packet switching

Gli end system scambiano messaggi tra di loro. Tra origine e destinazione ogni pacchetto viaggia attraverso i link di comunicazione e i packet switches. Un pacchetto di L bit su un link con velocità

di trasmissione R viene trasmesso in $\frac{L}{R}$ secondi.

La maggior parte dei packet switch usa trasmissione store-and-forward: il packet switch deve ricevere l'intero pacchetto prima di poter iniziare a trasmettere il primo bit del pacchetto su un collegamento in uscita (ritardo end to end = $n * \frac{L}{R}$).

Esempio

$L=2$, $R=10\text{Mbps}$, Pacchetti da 10Kbps

Ritardo nell'invio = $\frac{20\text{Kbps}}{10\text{Mbps}} = 2\text{ms}$

Ogni packet switch ha multipli collegamenti ad esso collegati. Per ogni connessione, un packet switch possiede un **buffer** di output che contiene i pacchetti che il router sta per inviare su quel collegamento. I pacchetti quindi soffrono di ritardo in accodamento: questi ritardi sono variabili e dipendono dalla congestione della rete. Nel caso il buffer sia pieno all'arrivo di un ulteriore pacchetto, quel pacchetto verrà perso (**packet loss**).

Ogni router ha una **tabella di inoltro** che mappa gli indirizzi di destinazione ai collegamenti in uscita di quel router: quando un pacchetto giunge ad un router, questo esamina l'indirizzo del pacchetto e cerca la connessione appropriata nella tabella di inoltro. Le tabelle sono automaticamente impostate tramite protocolli di routing. Il **forwarding** (switching) è un'azione locale, dove ogni router riceve il pacchetto dal link, legge la destinazione e invia al link d'uscita per proseguire il percorso del pacchetto. L'informazione viene salvata in una tabella:

Local forwarding table	
From/To	Output Link

Vantaggi: buono quando abbiamo un traffico non continuo

- Risorse condivise
- Semplice, nessun setup iniziale

Svantaggi: ritardo e perdita dei pacchetti dovuto all'overflow del buffer

- Protocolli necessari per l'affidabilità

1.3.2 Circuit switching

Un altro modo per muovere dati nella rete è quello del circuit switching (tipico della telefonia nel passato). Nelle reti a commutazione di circuito le risorse necessarie per la comunicazione lungo il percorso sono riservate per la durata della comunicazione tra end systems (nella commutazione di pacchetto non sono riservate). Un circuito in un collegamento è implementato tramite frequency-division multiplexing (FDM) oppure pure tramite time-division multiplexing (TDM). Con **FDM**, lo **spettro della frequenza** di un collegamento è diviso tra le connessioni stabilite. Con **TDM**, il **tempo** è diviso in frame di una certa durata, ed ogni frame è diviso in un certo numero di slot: quando la rete stabilisce una connessione attraverso ad un collegamento, la rete dedica uno slot in ogni frame a questa connessione, tutti utilizzano il massimo di capacità del link ma per un tempo limitato.

All'inizio della comunicazione avremo un delay dovuto alla fase di creazione del circuito

1.3.3 Rete di reti

Gli **ISP** d'accesso sono **interconnessi**, creando una rete di reti. Gli ISP sono disposti a livelli, con pochi ISP di primo livello che interconnettono centinaia di ISP di livello inferiore. Gli ISP di livello inferiore possono fare peering, ovvero connettersi direttamente tra loro senza dover passare ad

un ISP di livello superiore, tramite un Internet Exchange Point (IXP). Inoltre, i content-provider networks hanno le loro reti che portano servizi e contenuti vicino agli utenti finali.

Tipi di collegamento:

- **Full mesh:** ogni punto è connesso con tutti gli altri. $O(n^2)$ troppo costoso
- **Global ISP:** connettere tutti gli ISP ad un ISP globale. Ottimo per scalabilità e costi, ma il problema è che tutti vorrebbero fare da ISP globale, quindi se ne creerebbero tanti
- **Nuvola di ISP:** ISP più piccoli connessi tra di loro
 - Peering link : più ISP simili si uniscono e permettono il passaggio senza far pagare il traffico a chi vi transita.
 - Transit link: link dove si deve pagare per transitare.
 - Internet exchange point: punto di interconnessione tra ISP, gestito da operatori terzi che gestiscono gli IXP.

Per ridurre i tempi di attesa, i server vengono duplicati e posizionati nei pressi dei consumatori finali. **Content provider network**

1.4 Ritardo, perdite e throughput

1.4.1 Ritardo nelle reti a commutazione di pacchetto

Il tempo richiesto per esaminare l'header di un pacchetto e determinare dove esso debba essere spedito si dice ritardo di processamento.

Quando un pacchetto è in attesa di essere spedito, subisce un ritardo di accodamento, che dipende da quanto il buffer di accodamento è pieno.

Il ritardo di trasmissione è il tempo richiesto per trasmettere tutti i bit di un pacchetto su un collegamento.

Il tempo richiesto da un bit per propagarsi da un router ad un altro è detto ritardo di propagazione e dipende dalla velocità di propagazione del collegamento, che dipende dal mezzo fisico utilizzato.

Con d = distanza tra router e v = velocità di propagazione, il ritardo di propagazione è pari a :

$$\frac{d}{v}$$

Il ritardo totale è :

$$d_{total} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

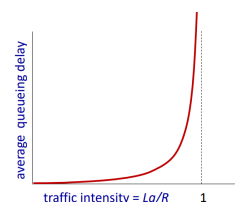
1.4.2 Ritardo di accodamento e perdita di pacchetti

Con a = velocità media di arrivo dei pacchetti

$$\text{intensità del traffico} = \frac{L \cdot a}{R}$$

$$\frac{L \cdot a}{R} :$$

- ~ 0 delay molto piccolo
- $\rightarrow 1$ delay molto grande
- > 1 più lavoro di quanto ne posso servire, packet loss



Data la capacità limitata della coda, se essa è piena e un pacchetto cerca di

accodarsi, quel pacchetto verrà perso.

1.4.3 Ritardo end-to-end

Il delay totale tra due router è :

$$d_{end-end} = N * (d_{proc} + d_{trans} + d_{prop})$$

Oltre a questi, si possono verificare ritardi dovuti a protocolli o applicazioni.

1.4.4 Throughput

Il throughput istantaneo è la velocità con cui un router riceve i dati in un determinato istante. Se un file è composto da F bit e impiega T secondi per essere trasferito, il throughput medio è $\frac{F}{T}$ bit al secondo. Il throughput è pari a quello minore tra tutti i collegamenti, ovvero pari a quello del bottleneck link.

1.5 Livelli protocollari

ISO ha deciso di applicare il sistema protocollare a 7 livelli:

1. Applicativo: interazione con l'utente, genera i messaggi da trasmettere
2. Presentazione: applica nuove informazioni
3. Sessione: sincronizzazione, recovery dei dati persi
4. Trasporto: trasferimento dei dati tra processi, identifica il processo (TCP, UDP)
5. Rete: routing dei dati (IP)
6. Collegamento: identificare l'elemento successivo per la comunicazione (router successivo)
7. Fisico: come i bit devono essere trasmessi nel mezzo fisico

Nella pratica i protocolli sono organizzati su più livelli, che offrono diversi servizi ai livelli sovrastanti. Lo stack dei protocolli consiste di 5 strati: applicazione, trasporto, rete, collegamento e fisico. Il livello applicazione è dove le applicazioni di rete e i loro protocolli risiedono: tali protocolli sono HTTP, SMTP, FTP... Il pacchetto nel livello applicazione è detto messaggio. Il livello di trasporto trasporta i messaggi del livello applicazione tra gli endpoint applicativi. Esistono due protocolli di trasporto, TCP e UDP. Il pacchetto a livello di trasporto è detto segmento. Il livello di rete trasporta pacchetti chiamati datagrammi, instradandoli tra i vari router. Include il protocollo IP e protocolli di routing.

Per trasferire un pacchetto tra un nodo e l'altro nel percorso, il livello di rete utilizza i servizi del livello di collegamento. I servizi offerti dal livello di collegamento dipendono dallo specifico protocollo implementato nel collegamento. I pacchetti a questo livello si chiamano frame. Il livello fisico si occupa di trasferire i bit in un frame da un nodo all'altro.

1.6 Sicurezza

Originariamente internet è stato progettato senza considerare la sicurezza. I protocolli sono nati per mettere una "pezza", il problema è nato quando internet è diventato accessibile a tutti.

Varie minacce presenti sulla rete:

- **Malware:** software indesiderato che compie azioni indesiderate
- **Denial of Service (DoS):** attacco mirato a rendere inutilizzabile un'infrastruttura che viene riempita di richieste. Il target all'inizio prova a rispondere alle richieste ma dopo un po' non ce la fa più e smette di funzionare
- **Packet sniffing:** l'attaccante intercetta il pacchetto e vede cosa è contenuto

al suo interno. Avviene quando l'attaccante ha accesso allo stesso canale fisico della vittima.

- **IP spoofing:** l'attaccante si finge un'altra persona, facendo provenire la richiesta a nome di un'altra persona. Oppure l'attaccante si finge il server.

Difesa:

- **Autenticazione:** controllare che l'utente sia veramente chi dice di essere
- **Confidentiality:** crittazione
- **Check d'integrità:** controllare che il messaggio sia stato compromesso

2 Livello applicazione

2.1 I principi delle applicazioni di rete

Un'applicazione di rete deve poter essere eseguita su sistemi diversi che comunicano tra di loro sulla rete.

2.1.1 Architetture

L'architettura di un'applicazione è progettata da uno sviluppatore e detta come l'applicazione sarà strutturata nei vari end system.

Nell'architettura **client-server** esiste un host sempre attivo, detto server, che serve richieste di altri host, detti client (Eg. HTTP, IMAP, FTP). Nell'architettura **peer-to-peer** l'applicazione sfrutta la comunicazione diretta tra host interconnessi, chiamati peer, ogni processo si comporta sia come client che come server. Una delle caratteristiche principali di questa architettura è l'**auto scalabilità** (Bittorrent).

2.1.2 Comunicazione tra processi

Nel contesto di una sessione di comunicazione, il processo che inizia la comunicazione è detto client; il processo che attende la comunicazione per iniziare la sessione è detto server. Anche i processi P2P utilizzano questa struttura.

Un processo invia messaggi, e li riceve da un'interfaccia detta **socket**, posta tra il livello applicazione e il livello trasporto. Viene anche chiamata Application Programming Interface (**API**) tra l'applicazione e la rete, dato che la socket è l'interfaccia con cui le applicazioni di rete sono sviluppate. Per far sì che un processo in esecuzione su un host riesca a spedire messaggi ad un altro processo in esecuzione su un altro host, il processo ricevente deve avere un indirizzo: l'host è identificato con un **indirizzo IP**. Inoltre, il processo mittente deve identificare il processo ricevente: cioè realizzato tramite un **numero di porta** (Eg. porta 80 HTTP).

Il livello applicazione fornisce:

- **Tipo** di messaggio scambiato: richiesta, risposta
- **Sintassi** del messaggio: quali campi del messaggio e come questi campi sono delineati
- **Semantica** del messaggio: il significato delle informazioni nei campi
- **Regole:** come e quando avvengono le risposte e le richieste

Tipi di protocolli:

- **Open** protocols: definito in un RFC, visibile da tutti (HTTP)
- **Proprietary** protocols: scelte private, funzionerà solo con certe applicazioni (Skype, Zoom)

2.1.3 Servizi di trasporto disponibili alle applicazioni

Le garanzie per il trasporto possono essere:

- **Data integrity:** recupero dei dati quando vengono persi
- **Timing:** ritardo massimo nella comunicazione (latenza minima)
- **Throughput:** alcune applicazioni necessitano di un throughput importante bandwidth-sensitive, altre accettano quello che ricevono
- **Security:** crittazione, autenticazione e integrità dei dati

2.1.4 Servizi di trasporto forniti da Internet

Internet fornisce alle applicazioni due protocolli di trasporto, **TCP** e **UDP**.

Il modello di servizio TCP include un servizio orientato a connessione e trasferimento dati **affidabile**. Il protocollo TCP assicura una connessione tra client e server prima dell'invio di messaggi (connessione TCP) e meccanismo di controllo della congestione di rete e controllo di flusso, per impedire che il ricevente venga sovraccaricato di messaggi.

UDP è un protocollo di **trasporto minimale**, che **non fornisce garanzie** di trasporto.

UDP è più veloce di TCP perché deve fare meno operazioni.

Sicurezza in TCP:

1. Lo sviluppatore rende sicuri i dati prima di inviarli
2. Sicurezza esterna nel livello sessione (**TLS**)

2.2 Il Web e HTTP

2.2.1 HTTP

L'HyperText Transfer Protocol (**HTTP**) è il protocollo di livello applicazione del **Web**. HTTP è implementato in due processi, il processo client e il processo server, che si scambiano messaggi HTTP. Il protocollo definisce come i **Web client** richiedono pagine Web ai **Web server** e come i server trasferiscono le pagine ai client.

HTTP usa TCP come protocollo di trasporto: il client HTTP inizializza una connessione TCP con il server e poi accede a TCP tramite le loro interfacce socket.

HTTP è un **protocollo stateless**, ovvero il server non mantiene informazioni sullo stato del client.

2.2.2 Connessioni persistenti e non

Client e server comunicano tra di loro per un periodo esteso di tempo, quindi è compito dello sviluppatore di applicazioni decidere se ogni richiesta debba avvenire su una connessione TCP separata (connessione **non persistente**) o se tutte le richieste debbano avvenire su una sola connessione TCP (connessione **persistente**).

HTTP non persistente (**HTTP 1.0** default):

1. Per ogni oggetto si apre una connessione TCP
2. Al massimo un oggetto viene inoltrato
3. Viene chiusa la connessione TCP

HTTP persistente (**HTTP 1.1** default):

1. Dopo l'apertura ci potranno essere più richieste e risposte
 2. Il client o server chiude la connessione TCP.
- Avviene una singola connessione

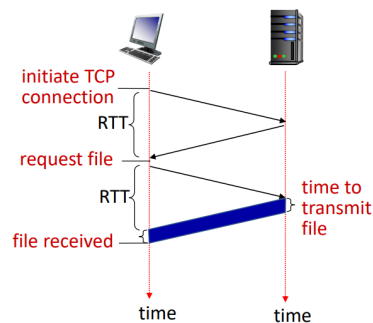
Una connessione viene chiusa quando non serve più per ridurre la complessità e aumentare la sicurezza.

Il round-trip time (**RTT**) è il tempo impiegato da un piccolo pacchetto per essere trasportato da client a server e di nuovo al client. Questo include ritardi di propagazione, accodamento e

processamento.

HTTP non persistente response time = $(2 * \text{RTT} + \text{file transfer time}) * n$

HTTP persistente response time = $n * \text{RTT} + \text{file transfer time}$



2.2.3 Il formato dei messaggi HTTP

Esistono due tipi di messaggi HTTP, messaggi di richiesta e messaggi di risposta. La prima riga del **messaggio di richiesta** è detta riga di richiesta, le seguenti sono dette righe di header. La **riga di richiesta** ha tre campi: **metodo**, **URL** e **versione HTTP**.

Il campo metodo può avere vari valori, di solito GET, usato quando un browser richiede un oggetto. Nelle righe di header sono specificati:

- l'**host** dove risiede l'oggetto richiesto,
- il **tipo di connessione** (persistente o non) e
- l'**User Agent** (tipo di browser che sta facendo la richiesta).

Il **corpo di un messaggio HTTP è usato con il metodo POST**, che permette di creare un nuovo oggetto web, i dati sono presenti nel body.

Il metodo **HEAD** è simile a GET ma non include l'oggetto richiesto, verifica se qualche oggetto è presente nel path.

PUT permette di caricare un oggetto su una directory specifica del server e aggiornare un oggetto già esistente.

DELETE permette di eliminare un oggetto.

Il **messaggio di risposta** è strutturato diversamente. Ha tre sezioni:

- **riga di status**,
 - versione di protocollo HTTP utilizzata,
 - codice di status
 - messaggio di status corrispondente
- **righe di header**
 - informazioni sulla persistenza della connessione
 - data di risposta
 - server dalla quale proviene
 - User Agent
 - data di modifica (critica nel caching degli oggetti)
 - lunghezza
 - tipo del contenuto
- **corpo**.

2.2.4 Cookies

I siti Web utilizzano cookies per tracciare gli utenti. L'implementazione ha quattro componenti:

1. una riga di header cookie nel messaggio HTTP di risposta

2. una riga di header cookie nel messaggio di richiesta
3. un file cookie sull'end system dell'utente e gestito dal browser
4. un database di backend del sito Web

Quando un client contatta un server, il server crea un **numero identificativo** (il cookie) e una voce nel database per l'ID del client. Le seguenti richieste HTTP da quel client conterranno un **cookie ID** che ne permetteranno l'identificazione.

2.2.5 Web caching

Una **Web cache** (o server proxy) è un'entità nella rete che serve richieste HTTP al posto del Web server d'origine. La Web cache **mantiene copie di oggetti richiesti recentemente**. Essa è **sia server che client**. Di solito sono **gestite dagli ISP**; possono ridurre sostanzialmente il tempo di risposta della richiesta di un client e possono ridurre traffico su un collegamento di accesso.

La copia di un oggetto contenuto in cache potrebbe essere obsoleta, ma HTTP fornisce un meccanismo per risolvere il problema: il **GET condizionale**. Un messaggio di richiesta è GET condizionale se contiene la riga di header **If-Modified-Since**. La cache inoltra l'oggetto al browser che lo richiede e ne memorizza una copia, memorizzando anche la data di ultima modifica. Se l'oggetto viene richiesto di nuovo dopo un po' di tempo, la cache ne controlla la validità tramite un GET condizionale al server d'origine. Se l'oggetto non è stato modificato, il server d'origine risponde con codice 304 Not Modified.

2.2.6 HTTP/2

HTTP è stato standardizzato nel 2015 e si prefigge di **ridurre la latenza percepita** consentendo il **multiplexing di richieste e risposte su una sola connessione TCP**, fornire **prioritizzazione delle richieste** e **compressione degli header HTTP**. HTTP/2 modifica come i dati sono formattati e trasportati tra client e server, mantenendo la compatibilità con HTTP/1.1.

HTTP/1.1 soffre di **Head of Line blocking**, ovvero gli oggetti che si trovano all'inizio della pagina HTML, se sono di dimensioni considerevoli rispetto agli altri, bloccano il caricamento degli altri oggetti. I browser, per evitare questo problema, aprono multiple connessioni TCP, invalidando però il controllo della congestione. HTTP/2 cerca di eliminare le connessioni TCP parallele.

La soluzione per l'HoL blocking fornita da HTTP/2 consiste nel **dividere ogni messaggio in piccoli frame** e interlacciando messaggi di richiesta e risposta in una sola connessione TCP. La suddivisione in frame è compiuta dal sottostrato di framing di HTTP/2. Quando un server vuole inviare una risposta HTTP, questa è processata nel sottostrato di framing, dove viene divisa in frame. I frame della risposta sono poi interlacciati con i frame di altre risposte e inviati in una sola connessione TCP persistente. I frame sono codificati in binario per aumentare l'efficienza del trasporto.

La prioritizzazione dei messaggi permette agli sviluppatori di decidere la priorità delle richieste per ottimizzare l'efficienza di una applicazione. Inoltre, HTTP/2 permette al server di inviare multiple risposte per una singola richiesta.

HTTP/2 è destinato ad essere sostituito a breve da HTTP/3, che permette di trasferire HTTP su UDP.

2.3 Posta elettronica

Il sistema delle e-mail è composto da tre parti principali:

- user agents: gmail
- mail servers: mailbox, message queue
- Simple Mail Transfer Protocol (**SMTP**).

Ogni destinatario ha un mailbox contenuto in uno dei mail server. Se un messaggio non può essere

consegnato, viene posto nella coda dei messaggi per essere inviato successivamente. SMTP è il protocollo di posta elettronica principalmente usato. Utilizza il trasporto affidabile di **TCP** per trasferire mail dal mail server del mittente al mail server del destinatario.

2.3.1 SMTP

Passi per inviare un messaggio tramite SMTP:

1. Utente A invoca il suo user agent per comporre l'email, fornendo l'indirizzo email di utente B
2. L'user agent dell'utente A invia il messaggio al mail server, piazzandolo nella coda dei messaggi
3. Il lato client di SMTP, in esecuzione sul mail server di utente A, vede il messaggio nella coda e apre una connessione TCP ad un server SMTP in esecuzione sul mail server di utente B
4. Dopo l'**handshaking**, il client SMTP invia il messaggio di utente A sulla connessione TCP
5. Sul mail server di utente B, il lato server di SMTP **riceve il messaggio**, che viene piazzato nel mailbox di utente B
6. Utente B invoca il suo user agent per leggere il messaggio

SMTP non utilizza server intermedi per inviare mail, la **connessione è diretta**.

Il client SMTP, in esecuzione sul server del mittente, stabilisce tramite TCP una connessione sulla porta 25 del server SMTP, in esecuzione sul server del destinatario. Quando la connessione è stabilita, il server e client eseguono **handshaking a livello applicazione**. Durante l'handshaking, il client SMTP specifica l'indirizzo email del mittente e del destinatario; terminate le introduzioni, il client invia il messaggio.

I comandi principali di SMTP sono: **HELO** (inizializza la sessione SMTP), **MAIL FROM** (inizializza il trasferimento mail), **RCPT TO** (specifica il destinatario), **DATA** (il client richiede al server il permesso di trasferire dati) e **QUIT** (invia richiesta per terminare la sessione SMTP). Ogni messaggio termina con **CRLF**.

2.3.2 Protocolli di accesso alla posta

Tipicamente, il mailbox di un utente è locato in un mail server condiviso con altri utenti. Ci sono due modi per recuperare email da un mail server:

- tramite user agent (che utilizza interfaccia HTTP e SMTP)
- tramite Internet Mail Access Protocol (**IMAP**).

2.4 DNS

Gli host sono identificati tramite hostname e indirizzo IP. Il servizio che si occupa di **tradurre hostname in indirizzi IP** è il domain name system (**DNS**).

2.4.1 Servizi forniti da DNS

DNS è un database distribuito implementato come gerarchia di server DNS e un protocollo a livello applicazione che permette agli host di interrogare il database. Il protocollo **DNS utilizza UDP** sulla porta 53.

Per inviare un messaggio HTTP ad un server, il mittente deve prima conoscere l'indirizzo IP del destinatario:

1. Il browser estrae l'hostname dall'URL e lo passa alla parte client dell'applicazione DNS
2. Il client DNS invia una query contenente l'hostname ad un server DNS
3. Il client DNS riceve una risposta contenente l'indirizzo IP corrispondente all'hostname

4. Il browser inizia una connessione TCP al server HTTP sulla porta 80 di quell'indirizzo IP

DNS fornisce anche altri servizi:

- **Host aliasing**: un host può avere multipli alias, DNS può essere invocato per ottenere il nome canonico
- **Mail server aliasing**
- **Load distribution**: siti molto frequentati sono replicati su più server, quindi più IP sono associati allo stesso hostname; DNS risponde fornendo tutti gli indirizzi IP associati, ma ne varia l'ordine ad ogni risposta

2.4.2 Come funziona DNS

DNS è distribuito. Ci sono tre classi di server DNS:

- Server DNS radice: sono coordinati dall'Internet Corporation for Assigned Numbers Authority (**ICANN**) e forniscono indirizzi IP dei server TLD
- Server DNS top-level domain (**TLD**): server dei top-level domain (.com, .org, ...)
- Server DNS autoritativi: server di organizzazioni con host pubblicamente accessibili

Ogni ISP ha inoltre un server DNS locale: quando un host si connette ad un ISP, l'ISP fornisce all'host l'indirizzo IP di uno dei suoi server DNS locali.

Una query DNS può essere iterativa o ricorsiva.

Il **caching DNS** è una caratteristica importante del sistema DNS. Quando un server DNS riceve una risposta DNS, può memorizzarla nella sua cache locale. Le informazioni in cache sono scartate dopo un certo periodo di tempo.

2.4.3 Record DNS e messaggi

I server DNS che implementano il database DNS memorizzano resource records (RR), tra i quali RR che forniscono le mappature tra hostname e IP. Un **resource record** è una quadrupla contenente i seguenti campi:

- Name,
- Value,
- Type,
- TTL.

TTL è il **time to live** del RR, determina quando una risorsa deve essere rimossa dalla cache. Il significato di Name e Value dipendono da Type:

- Type=**A**: Name è un hostname e Value è l'indirizzo IP per l'hostname
- Type=**NS**: Name è un dominio e Value è l'hostname di un server DNS autoritativo che sa come ottenere gli indirizzi IP degli host in quel dominio
- Type=**CNAME**: Value è l'hostname canonico per alias Name
- Type=**MX**: Value è l'hostname canonico di un server mail che ha come alias Name

I messaggi DNS hanno lo stesso formato sia come query sia come risposta.

- I primi 12 byte sono la **header section**, che è composta da altri campi. Il primo campo è un numero a 16 bit che identifica la query. Il campo delle flag contiene flag per indicare se il messaggio è query risposta, se la ricorsione è preferibile, se la ricorsione è supportata e se la risposta è autoritativa.
- La **question section** contiene informazioni sulla query che si sta facendo: contiene nome che si sta richiedendo e il tipo di richiesta che si sta facendo.
- In un messaggio di risposta, la **answer section** contiene RR per il nome che è stato originariamente richiesto.
- L'**authority section** contiene record per gli altri server autoritativi.
- L'**additional section** contiene altri record utili.

Per inserire un domain name nel database DNS si chiede ad un registrar.

2.5 Peer-to-Peer

Nell'architettura **P2P** non sono previsti server sempre online, bensì una rete di end system direttamente connessi. Ogni peer può distribuire una porzione del file che ha ricevuto agli altri peer. Il tempo di distribuzione è il tempo impiegato per far giungere una copia del file a tutti i peer.

Denotando il numero dei peer con N , i bit di cui è composto il file con F , la velocità di upload del collegamento di accesso del server con u_s , la velocità di upload dell' i -esimo collegamento d'accesso di un peer con u_i e la velocità di download la velocità di upload dell' i -esimo collegamento d'accesso di un peer con d_i , allora il tempo di distribuzione con architettura client-server è :

$$D_{cs} \geq \max\left\{\frac{N * F}{u_s}, \frac{F}{d_{min}}\right\} \text{ (incrementa linearmente con } N \text{)}$$

mentre con architettura P2P è :

$$D_{P2P} \geq \max\left\{\frac{F}{u_s}, \frac{F}{d_{min}}, \frac{N * F}{(u_s + \sum u_i)}\right\}$$

Il protocollo P2P più diffuso è **BitTorrent**. Una collezione di peer che partecipano nella distribuzione di un file è detta *torrent*.

I peer in un torrent scaricano **chunk**, tipicamente di **256 KByte**, di un file tra di loro. Quando un peer entra in un torrent, non possiede chunk, li accumula con il tempo.

Ogni torrent ha un nodo detto **tracker**. Quando un peer entra nel torrent, si registra con il tracker per informarlo periodicamente di essere ancora nel torrent. Mentre scarica, un peer carica chunk agli altri peer, ed un peer può cambiare i peer con cui sta scambiando i chunk. In ogni istante di tempo, ogni peer avrà un sottoinsieme dei chunk del file, con peer diversi aventi diversi sottoinsiemi. Periodicamente, un peer chiede ad ogni peer la lista dei chunk che possiedono, e ne richiede uno che gli manca, **scegliendo per primo il più raro**.

Una volta scaricati i file un peer può decidere se rimanere collegato per condividere i file oppure scollegarsi dal torrent. Un peer che abbandona il torrent è detto **Churn**.

Ogni peer quando deve inviare dei file, ad ogni istante di tempo, può essere collegato **al massimo a 5 peer** contemporaneamente.

I primi 4 vengono scelti in base a chi sta trasmettendo in modo più veloce, detti **unchoked**; ogni 10 secondi la Top 4 viene rivalutata. Il 5° peer viene selezionato in maniera casuale tra i richiedenti e viene rivalutato ogni 30 secondi, questo è detto **optimistically unchoked**.

2.6 Applicazioni per streaming video

Nella distribuzione dei contenuti multimediali sono molto utilizzate le **CDN** per distribuire i dati sul territorio e avvicinarli ai consumatori finali.

Un video è una sequenza di immagini, dette frame. I frame possono essere codificati come una sequenza di pixel, ogni pixel rappresentato da un bit. Se inviassimo indipendentemente ogni frame avremmo uno spreco di spazio perché avremmo molti frame simili tra di loro.

Soluzioni:

- Codifica **differenziale spaziale**: invece che inviare N valori dello stesso colore, invio solo due valori, colore e numero di ripetizioni (N)
- Codifica **differenziale temporale**: invio il frame i e per il frame $i+1$ invio solo le differenze

dal frame i

Il **bit rate di codifica** è la quantità di bit che si possono inviare in un'unità di tempo.

- **CBR**: bit rate costante, adatta per live streaming ma poco flessibile. La qualità potrebbe non essere ottimale se ci dovessero essere tante variazioni tra i frame. (MPEG1 CD-ROM)
- **VBR**: bit rate variabile, il numero di bit offerti è variabile in base alla complessità. La qualità video sarà superiore ma la complessità nettamente maggiore. Ottimo per streaming on demand (contenuto già pronto. Eg video). (MPEG2 DVD, MPEG4 Internet)

Nello streaming di stored video senza un buffer client e server devono viaggiare alla stessa velocità per evitare tempi di caricamento. Quindi viene aggiunto un buffer ed un delay aggiuntivo per accumulare i dati in un buffer per cercare di arginare il problema del caricamento in caso di asincronia tra il client e il server.

Il **jitter** è la variabilità di ritardo nel tempo.

Protocollo **DASH**:

Ogni video viene codificato a più bit rate

Il server:

- Divide il file video in più chunks
- Ogni chunk è codificato a più bit rate
- I file vengono duplicati in più CDN
- Viene creato un **manifest file** che contiene l'URL di ogni chunk

Il client:

- Periodicamente stima la larghezza di banda server-to-client
- Consulta il manifest e richiede un chunk alla volta
 - Sceglie il massimo coding rate sostenibile con la larghezza di banda corrente
 - Può scegliere diversi coding rate in momenti diversi

3 Livello trasporto

3.1 I servizi del livello trasporto

Un protocollo a livello trasporto fornisce **comunicazione logica tra i processi applicativi** in esecuzione su host diversi (per l'applicazione, è come se gli host fossero direttamente connessi).

Il livello trasporto **converte i messaggi ricevuti da un'applicazione in pacchetti di livello trasporto**, detti **segmenti**. Questi sono poi passati al livello rete.

Internet fornisce due protocolli nel livello trasporto, **TCP** e **UDP**, e uno per il livello rete, **IP**. Il modello **IP è un servizio di spedizione best-effort**: non garantisce che due segmenti vengano correttamente inviati tra host.

La responsabilità principale di TCP e UDP è estendere il servizio di spedizione dell'IP da un servizio tra end system ad un servizio tra due processi in esecuzione sui suddetti sistemi: questa operazione è detta multiplexing e demultiplexing del livello trasporto.

TCP:

- Affidabilità, consegna in ordine dei pacchetti
- Controllo della congestione
- Controllo di flusso
- Setup di connessione

UDP:

- Inaffidabile, consegna NON in ordine dei pacchetti
- Best effort puro

Il jitter si può verificare in entrambi i casi.

3.2 Multiplexing e demultiplexing

Un processo può avere una o più socket. L'atto di consegnare dati nel livello trasporto alla socket corretta è detto **multiplexing**. L'atto di aggregare blocchi di dati nell'host sorgente, incapsulare ogni blocco con informazioni di header per creare segmenti e spedire i segmenti al livello rete è detto **demultiplexing**.

Il multiplexing richiede che le socket siano unicamente identificabili e che ogni segmento abbia dei campi che indicano a quale socket il segmento è destinato. Questi campi sono il **numero di porta sorgente** e il **numero di porta destinazione**. Ogni porta è un numero a 16 bit (0-65535). I numeri da 0 a 1023 sono chiamati numeri di porta ben conosciuti e sono riservati ai protocolli ben conosciuti, come HTTP.

Quando si crea una socket UDP, bisogna specificare il numero di porta ad essa associata (la parte client di solito la assegna automaticamente).

Un processo nell'host A, con un determinato numero di porta UDP, vuole inviare un blocco di dati ad un host B.

Il livello trasporto dell'host A crea un segmento che include i dati dell'applicazione, il numero di porta sorgente e quello di destinazione.

Il livello trasporto inoltra il segmento così creato al livello rete, che lo incapsula in un datagramma IP e cerca di consegnarlo all'host B.

Se il pacchetto arriva all'host B, il livello trasporto esamina il numero di porta di destinazione contenuto nel segmento e lo consegna alla porta corrispondente.

Un **socket UDP** è identificato da una **tupla** contenente:

- l'indirizzo IP di destinazione
- un numero di porta.

Il numero di porta sorgente funge da indirizzo di ritorno. I datagrammi che hanno la stessa porta di destinazione ma diverso indirizzo o porta sorgente sono inoltrati allo stesso socket nell'host destinatario.

Un **socket TCP** è identificato da una **quadrupla**:

- indirizzo IP sorgente,
- porta sorgente,
- indirizzo IP destinazione
- porta destinazione.

Un server può supportare molte connessioni TCP simultanee, con ogni socket collegata ad un processo, ed ogni socket identificata dalla sua quadrupla. Quando un segmento TCP arriva ad un host, tutti e quattro i campi sono usati per demultiplexare il segmento alla socket appropriata.

3.3 Trasporto connectionless: UDP

UDP è un protocollo di trasporto molto elementare, oltre a error checking e multiplexing, non aggiunge nulla all'IP. Con UDP **non vi è handshaking** tra entità mittenti e destinatari e, per cui è detto **connectionless**.

UDP consente di avere maggior controllo su quali dati vengono spediti e quando: non appena processo passa i dati ad UDP, UDP li incapsula in un segmento e li passa immediatamente al livello rete (TCP effettua controllo della congestione e può rallentare il passaggio trasporto-rete per

evitarne la congestione). UDP non stabilisce connessione con il destinatario, non mantiene lo stato della connessione ed ha un minore overhead nell'header (solo 8 byte rispetto ai 20 di TCP).

UDP è utilizzato per:

- Streaming di applicazioni multimediali (tolleranti alle perdite)
- DNS, al massimo il DNS ripete il messaggio se viene perso
- SNMP, configurazione di dispositivi di rete da remoto
- HTTP/3

Se necessario dell'affidabilità in UDP:

- Aggiungo l'affidabilità richiesta al livello applicazione
- Aggiungo il controllo della congestione al livello applicazione

3.3.1 Struttura del segmento UDP

I dati dell'applicazione occupano il campo dati del segmento.

L'header ha solo **quattro campi**, ognuno grande due byte (16 bit):

- porta sorgente
- porta destinazione
- lunghezza segmento
- checksum.

3.3.2 Checksum UDP

Il checksum permette di determinare se sono avvenute delle modifiche ai bit mentre il segmento veniva trasmesso.

UDP dalla parte del mittente compie il **complemento ad 1** della somma delle parole a 16 bit del segmento e scrive il risultato nel campo checksum. UDP fornisce checksum perché non vi è garanzia che il protocollo a livello collegamento faccia error-checking.

Il ricevente calcola il checksum e lo confronta con quello ricevuto, se il risultato è diverso il pacchetto viene scartato, perché non sa dove è l'errore.

Trasmesso

$1^{\text{st}} = 5, 2^{\text{nd}} = 6, \text{sum} = 11$

Ricevuto

$1^{\text{st}} = 4, 2^{\text{nd}} = 6, \text{sum} \neq 11$ ERRORE

1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0

1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 wraparound

1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 sum

0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 checksum

3.4 Trasporto dati affidabile

TCP è un protocollo di trasferimento dati affidabile implementato su un protocollo inaffidabile, IP.

3.4.1 Costruire un protocollo di trasferimento affidabile

Il caso più semplice è considerare di avere un **canale di trasporto completamente affidabile**.

Con **rdt1.0**, le macchine a stati finiti mittente e destinatario hanno **un solo stato**.

Il **lato mittente** di rdt accetta dati dal livello superiore, crea un pacchetto contenente i dati e inoltra il pacchetto sul canale.

Il **lato destinatario**, rdt riceve un pacchetto dal canale sottostante, estrae dati dal pacchetto e passa i dati al canale superiore.

In questo semplice protocollo, i pacchetti sono trasmessi da un host all'altro senza bisogno di controllarli perché ci si fida del canale di trasmissione.



Un modello più realistico ammette la possibilità che i pacchetti possano contenere bit corrotti. Questo protocollo prevede l'utilizzo di riscontri positivi (**ACK**) e riscontri negativi (**NAK**) per indicare se un messaggio è stato ricevuto correttamente (e quindi non serve rimandarlo) o meno (e quindi ritentare la consegna).

Protocolli basati su questo metodo di ritrasmissione sono chiamati Automatic Repeat reQuest (**ARQ**).

Tre capacità fondamentali richieste:

1. rilevazione degli errori
2. feedback del destinatario
3. ritrasmissione.

Il lato **mittente** di **rdt2.0** ha **due stati**.

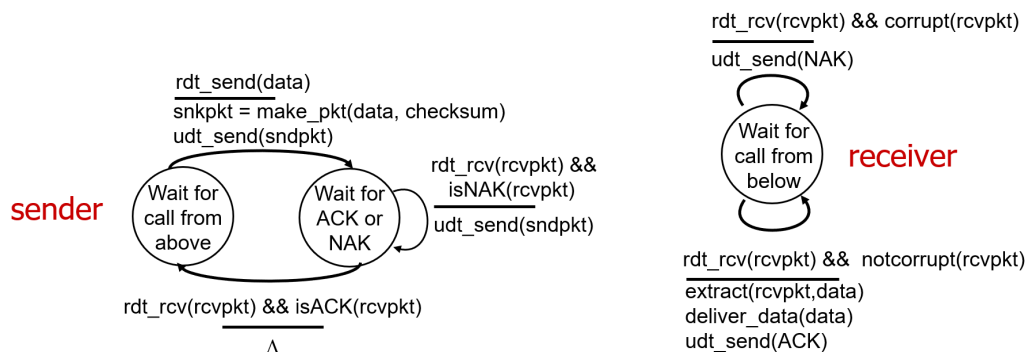
Nel primo, il lato mittente del protocollo attende che i dati gli siano passati dal livello superiore. Quando l'evento `rdt_send(data)` accade, il mittente crea un pacchetto contenente i dati da inviare ed un checksum.

Nel secondo stato, il protocollo attende un ACK o NAK dal destinatario. Se riceve un pacchetto ACK, il mittente sa che il pacchetto è stato ricevuto correttamente e ritorna nello stato di attesa dati; se riceve un NAK, il protocollo ritrasmette l'ultimo pacchetto inviato ed attende un ACK o NAK.

Quando il mittente è nello stato di attesa di ACK o NAK, non può ricevere ulteriori dati dal livello superiore.

Protocolli che esibiscono questo comportamento sono detti protocolli **stop-and-wait**.

Il lato **destinatario** di **rdt2.0** ha **un solo stato**; all'arrivo di un pacchetto, il destinatario risponde con ACK o NAK a seconda dell'integrità del pacchetto ricevuto.



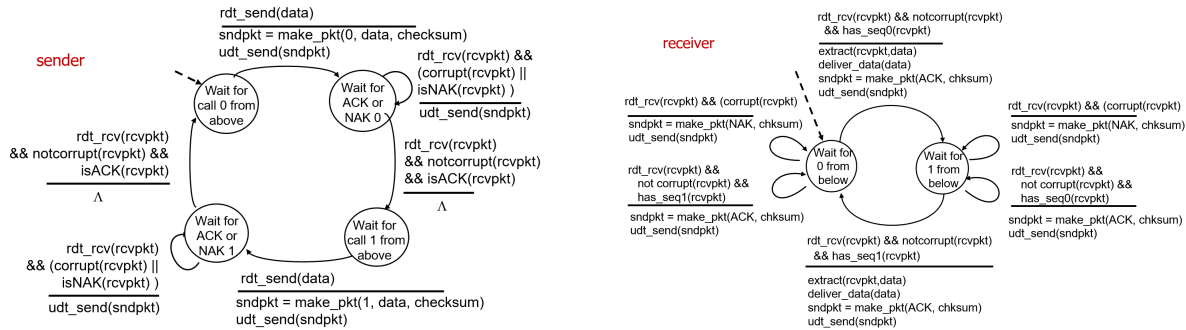
Per ovviare al problema dei possibili pacchetti **ACK e NAK corrotti**, il mittente numera i propri pacchetti dati con un numero di sequenza (0 e 1 alternati).

La ritrasmissione non è accettabile perché si rischia di avere pacchetti duplicati.

Il protocollo **rdt2.1** usa riconoscimenti sia positivi sia negativi da mittente a destinatario. Quando viene ricevuto un pacchetto fuori ordine, il destinatario invia un riconoscimento positivo per il

pacchetto che ha ricevuto.

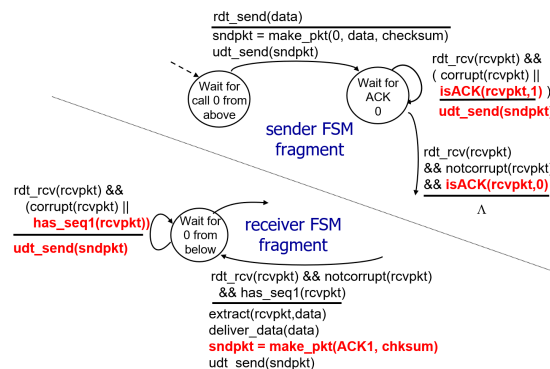
Quando viene ricevuto un pacchetto corrotto, il destinatario manda un riconoscimento negativo.



Si possono ottenere gli stessi effetti di un NAK se si invia un ACK per l'ultimo pacchetto correttamente ricevuto.

Un mittente che riceve due ACK per lo stesso pacchetto (**ACK duplicati**) sa che il destinatario non ha ricevuto correttamente il pacchetto seguente a quello che ha ricevuto due ACK.

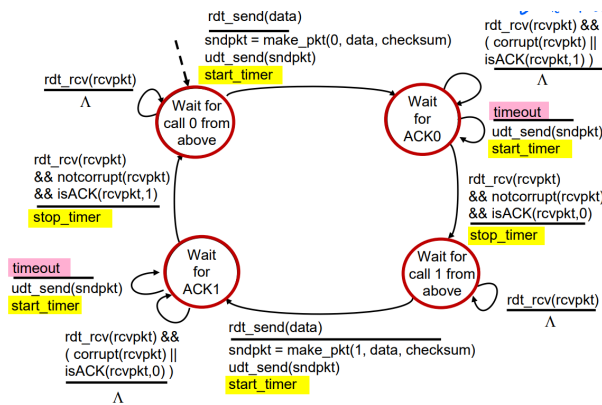
Questo protocollo senza NAK è **rdt2.2**. Il destinatario deve ora includere il numero di sequenza del pacchetto che viene riconosciuto da un ACK e il mittente deve controllare il numero di sequenza di quel pacchetto.



Esistono vari modi per gestire la perdita di pacchetti.

Se un mittente trasmette un pacchetto e questo viene perso, può aspettare e poi ritentare una ritrasmissione. Il tempo da aspettare è il **RTT** tra il mittente e destinatario più il tempo impiegato a processare un pacchetto. In pratica, il tempo di ritrasmissione è scelto in modo arbitrario dal mittente. Questo introduce la possibilità di pacchetti duplicati, che sono gestiti dai numeri di sequenza. Implementare un sistema di ritrasmissione basato sul tempo richiede un **countdown timer** che interrompa il mittente dopo un certo lasso di tempo.

Il mittente deve quindi essere in grado di **avviare** il **timer** ogni volta che un pacchetto viene inviato, **rispondere ad un'interruzione** e **interrompere** il **timer**. Poiché la sequenza dei pacchetti si alterna tra 0 e 1, il protocollo **rdt3.0** è detto **alternating bit protocol**.



3.4.2 Pipeline

L'utilizzazione è il lasso di tempo in cui un mittente è attualmente occupato ad inviare bit su un canale, ed è pari a $(L \text{ dimensione pacchetto}, R \text{ velocità di trasmissione}, RTT \text{ tempo da quando invio l'ultimo bit a quando ricevo l'ACK})$

$$U_{sender} = \frac{L/R}{RTT + L/R}$$

L'utilizzazione con il protocollo **rdt3.0** è molto bassa, quindi invece di inviare un solo pacchetto ed aspettare un ACK, se ne inviano molti allo stesso momento senza aspettare ACK. Questa tecnica è detta **pipelining**.

3.4.3 Go-Back-N

In un **Go-Back-N** protocol, il mittente può **trasmettere multipli pacchetti senza aspettare ACK**, ma **non** può avere **più di N pacchetti senza riscontro** nella pipeline.

La **finestra di trasmissione** W_t rappresenta la quantità massima di pacchetti in sequenza che il trasmettitore è autorizzato ad inviare in rete senza averne avuto riscontro: la dimensione è limitata dalla quantità di memoria allocata in trasmissione.

La **finestra di ricezione** W_r rappresenta la sequenza di pacchetti che il ricevitore è disposto ad accettare e memorizzare: la dimensione è limitata dalla quantità di memoria allocata in ricezione. Il **trasmettitore**, con finestra N, invia fino a N pacchetti, facendo di ognuno una copia, attiva **un solo timer** per gli N pacchetti e si pone in attesa di ACK. Se il timer scade prima dell'arrivo della conferma di ricezione relativa al pacchetto che l'ha settato, ripete la trasmissione di tutti i pacchetti non ancora confermati.

Il **ricevitore**, quando riceve un pacchetto, ne controlla la correttezza e il numero di sequenza: se è corretto, invia la conferma di ricezione; se contiene il primo numero di sequenza non ancora ricevuto lo invia ai livelli superiori.

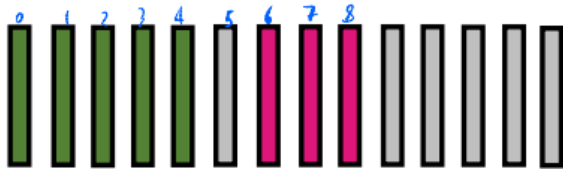
La semantica associata al pacchetto di riscontro può essere diversa da protocollo a protocollo, quindi **trasmettitore e ricevitore si devono accordare preventivamente**.

Con **ACK cumulativo**, si notifica la corretta ricezione di tutti i pacchetti con un numero di sequenza inferiore a quello specificato nell'ACK: **ACK(n)** significa "ho ricevuto tutti in sequenza fino ad n escluso". Con ACK negativo, si notifica la richiesta di ritrasmissione di un singolo pacchetto: **NAK(n)** significa "ritrasmetti il pacchetto n", ma non dà indicazioni su quali pacchetti sono stati ricevuti.

Nel caso di flussi di informazione bidirezionali è sovente possibile scrivere l'informazione di riscontro nell'intestazione di pacchetti che viaggiano nella direzione opposta, risparmiando ACK (**piggybacking**).

Eg. Arriva solo ACK(6), per il sender tutti i pacchetti precedenti sono stati ricevuti. Invio il

pacchetto 7.



Il receiver continua ad inviare ACK(4).

Il sender invia tutta la finestra anche se il receiver ha ricevuto 6,7,8 ecc...

3.4.4 Selective Repeat

Nel protocollo Go-Back-N, il ricevitore può accettare solo pacchetti in sequenza, ma accettare pacchetti corretti fuori sequenza migliorerebbe le prestazioni.

Il protocollo **selective repeat** (SR) usa finestre di trasmissione e ricezione di dimensioni maggiori di 1.

Il trasmettitore funziona in maniera analoga a Go-Back-N.

Il ricevitore, quando riceve un pacchetto, se questo è corretto ma non in sequenza: se è dentro alla finestra di ricezione lo memorizza, altrimenti lo scarta. Invia comunque un'ACK relativo all'ultimo pacchetto ricevuto in sequenza.

In caso di **perdita singola**, SR si comporta come GBN in termini di velocità di trasmissione e occupazione del canale. Si ottengono vantaggi rispetto a GBN se il RTT è minore del tempo di trasmissione della finestra, in quanto un nuovo ACK permette di spostare in avanti la finestra prima di completare la ritrasmissione, ed in presenza di **perdite ripetute** sui dati, poiché si memorizza in ricezione, quindi è sufficiente che una sola copia di ogni pacchetto sia arrivata al ricevitore.

Modificando il comportamento del trasmettitore e vincolandolo a ritrasmettere solo il primo pacchetto perso nella finestra si riduce l'occupazione del canale.

Nel protocollo SR vale la relazione $W_t + W_r \leq 2^k$, che lega la dimensione delle finestre di trasmissione e ricezione con i k bit di numerazione.

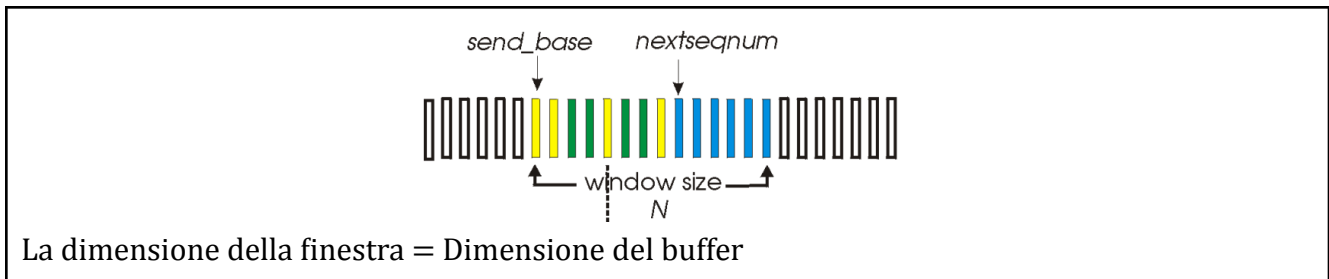
Mittente:

- Dati dall'alto:
 - Se il prossimo numero di sequenza disponibile è nella finestra, invia il pacchetto
- Timeout(n):
 - Ritrasmetti il pacchetto n, riavvia il timer
- ACK(n) in [sendbase, sendbase+N-1]:
 - Segna il pacchetto n come ricevuto e ferma il timer
 - Se n è il pacchetto non riconosciuto (unACKed) più piccolo, avanza la base della finestra al prossimo numero di sequenza non riconosciuto (unACKed)

Ricevitore:

- Pacchetto n in [rcvbase, rcvbase+N-1]:
 - Invia ACK(n)
 - Fuori ordine: memorizza in buffer
 - In ordine: consegna (anche consegna i pacchetti in buffer, in ordine), avanza la finestra al prossimo pacchetto non ancora ricevuto
- Pacchetto n in [rcvbase-N, rcvbase-1]:
 - ACK(n) Si è perso l'ACK

- Altrimenti:
 - Ignora



3.5 TCP

3.5.1 La connessione TCP

TCP è **connection-oriented** in quanto prima che un processo possa iniziare ad inviare dati ad un altro, questi devono prima inviarsi dei segmenti per stabilire i parametri dell'imminente trasferimento dati.

Una connessione TCP fornisce un servizio **full-duplex**: se esiste una connessione TCP tra processo A su un host e processo B su un altro, allora i dati a livello applicazione possono scorrere da processo A a processo B nello stesso momento in cui i dati a livello applicazione scorrono da processo B a processo A.

Una connessione TCP è sempre **point-to-point**, ossia tra un solo mittente e un solo destinatario.

Quando un processo (client) vuole stabilire una connessione TCP con un altro processo (server), innanzitutto informa il suo livello trasporto; TCP nel client stabilisce quindi una connessione con TCP nel server. Il client inizialmente invia un segmento TCP speciale, il server risponde con un secondo segmento speciale ed infine il client risponde di nuovo con un terzo segmento. I primi due segmenti non contengono dati dell'applicazione. Questa procedura è chiamata **three-way handshake**.

Una volta stabilita la connessione, i due processi possono iniziare ad inviarsi dati.

Il client inoltra un flusso di dati attraverso il suo socket e passati a TCP.

TCP dirige i dati nel **send buffer** della connessione, che è uno dei buffer messi da parte durante il three-way handshake. Ogni certo lasso di tempo, TCP prende dati da questo buffer e li inoltra al livello rete.

La quantità massima di dati che è possibile prendere e mettere in un segmento è limitata dalla dimensione massima del segmento (maximum segment size, **MSS**). Il MSS è determinato dalla **lunghezza del più lungo frame a livello collegamento** (unità massima di trasmissione, **MTU**): MSS è impostato per assicurare che un segmento TCP più l'header TCP/IP possano essere contenuti in un singolo frame del livello collegamento.

TCP associa ogni blocco di dati dell'applicazione con un header TCP, formando un **segmento TCP**. I segmenti sono passati al livello rete, dove sono incapsulati in **datagrammi IP**. Questi datagrammi sono infine inviati sulla rete. Quando TCP riceve un segmento, lo piazza in un receive buffer, dal quale flusso di dati l'applicazione in esecuzione su quell'host leggerà.

3.5.2 Struttura del segmento TCP

Il segmento TCP consiste di un **campo header** e un **campo dati**.

L'header include:

- Numero di porta sorgente
- Numero di porta destinazione

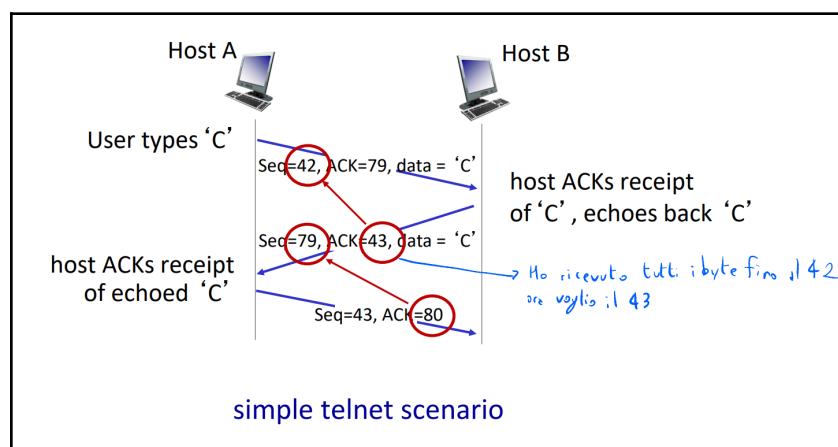
- checksum
- Numero di sequenza (32 bit) [Finestra]
- Numero di riscontro (32 bit)
- finestra di ricezione (16 bit) usata per controllo del flusso
- lunghezza header (4 bit) che specifica la lunghezza dell'header TCP in parole a 32 bit
- ulteriori opzioni per negoziare il MSS
- bit di ACK, indica che il valore nel campo di riscontro è valido; tre bit per RST, SYN e FIN, usati per il setup della connessione

TCP vede i dati come un **flusso di byte non strutturato ma ordinato**.

Il **numero di sequenza** per un segmento è il numero di flusso di byte del primo byte del segmento.

Il **numero di riscontro** che un host A inserisce nel suo segmento è il numero di sequenza del prossimo byte che l'host A si aspetta da un host B.

Poiché TCP riscontra solo i byte fino al primo byte mancante nel flusso, si dice che **TCP fornisce riscontri cumulativi**.



3.5.3 Stima del RTT e timeout

TCP usa un sistema di **timeout** e **ritrasmissioni** per recuperare i segmenti perduti.

Il timeout deve essere più grande del RTT, ma come stimare il RTT?

Il **RTT esempio** di un segmento è il lasso di tempo che intercorre tra l'invio di un segmento e la ricezione dell'ACK per quel segmento.

Invece di misurare un RTT esempio per ogni segmento trasmesso, TCP lo misura circa una volta ogni RTT.

TCP mantiene una media, detta **RTT stimato**, dei valori di RTT esempio.

Ogni volta che ottiene un nuovo RTT esempio, TCP aggiorna la media tramite la formula:

$$RTT \text{ stimato} = (1 - \alpha) * RTT \text{ stimato} + \alpha * RTT \text{ esempio.}$$

Il **valore raccomandato di α** è **0.125**. Questa media è detta exponential weighted moving average (**EWMA**).

La variazione di RTT si calcola con la formula :

$$RTT \text{ dev} = (1 - \beta) * RTT \text{ dev} + \beta * |RTT \text{ esempio} - RTT \text{ stimato}|$$

Il **valore raccomandato per β** è **0.25**.

L'intervallo di timeout è pari a :

$$T = RTT \text{ stimato} + 4 * RTT \text{ dev}$$

3.5.4 Trasferimento dati affidabile

Il servizio di trasferimento affidabile di TCP garantisce che il flusso di dati letto dal receive buffer

sia **integro, privo di duplicati ed in sequenza**.

TCP usa un **singolo timer** di ritrasmissione, invece di uno per ogni segmento non ancora riscontrato, questo parte solo per il primo pacchetto della finestra (il segmento più vecchio).

Ci sono tre eventi legati alla trasmissione e ritrasmissione dei dati del mittente TCP:

1. dati ricevuti dall'applicazione,
2. timeout del timer
3. ricezione di ACK.

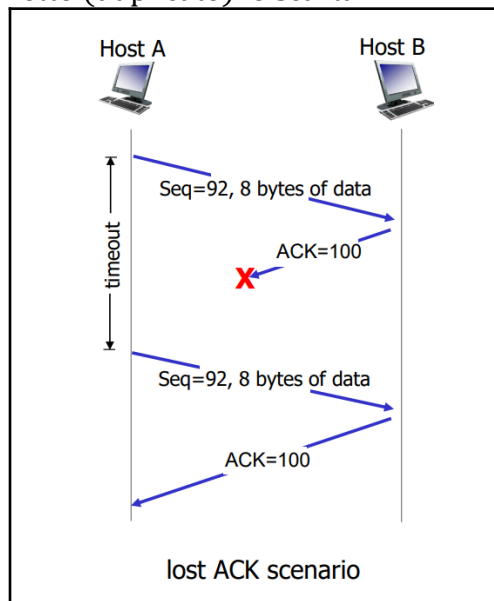
TCP **riceve** dati dall'applicazione, **incapsula** dati in un segmento e lo **passa** ad IP, se il timer non è attivo per un altro segmento, TCP lo avvia quando il segmento è passato ad IP.

TCP risponde al **timeout** ritrasmettendo il segmento che lo ha causato e riavvia il timer.

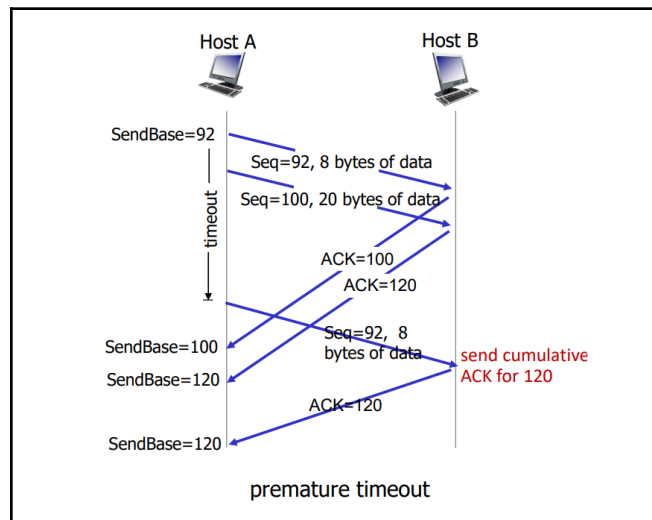
Quando viene ricevuto un **ACK** per un segmento che era in attesa di riscontro, TCP aggiorna il suo stato ed avvia il timer se ci sono segmenti ancora senza riscontro.

Tre scenari interessanti:

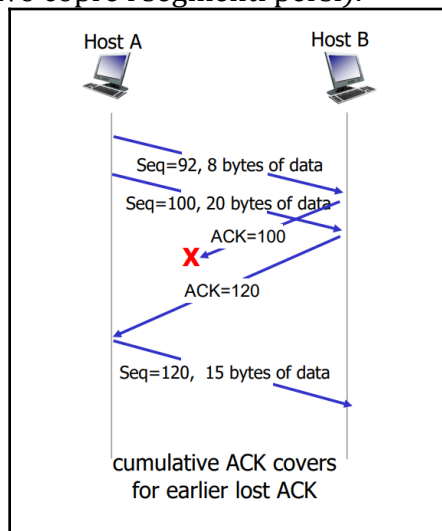
- Host A invia un segmento ad host B ed attende un ACK; host B riceve il segmento e invia un ACK, ma questo viene perso. Avviene l'evento di timeout, e host A ritrasmette il pacchetto. Quando host B riceve il pacchetto (duplicato) lo scarta.



- Host A invia contemporaneamente due segmenti, che arrivano a B intatti. B invia due ACK separati, ma nessuno dei due giunge ad A. Quando avviene il timeout, A ritrasmette il primo pacchetto e riavvia il timer. Se l'ACK per il secondo segmento arriva prima di un altro timeout, il secondo segmento non verrà ritrasmesso.



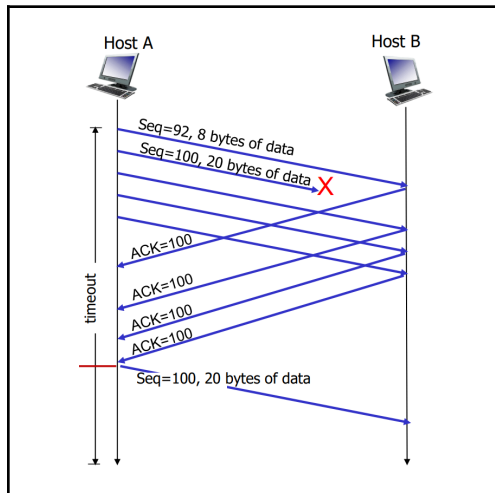
- Host A invia due segmenti; l'ACK del primo segmento viene perso, ma prima del timeout A riceve l'ACK del secondo segmento. In questo caso nessuno dei due segmenti sarà ritrasmesso (l'ACK cumulativo copre i segmenti persi).



Ogni volta che TCP ritrasmette, **raddoppia il valore dell'intervallo di timeout**. Tuttavia, ogniquale volta il timer viene avviato dagli altri due eventi (ricezione pacchetti dall'applicazione e ricezione ACK), l'intervallo di timeout viene derivato dal **RTT stimato**. Questa modifica fornisce una forma di **controllo della congestione**, in quanto lo scadere del timer è molto probabilmente causato dalla congestione nella rete.

Il mittente può individuare **perdita di pacchetti** prima del timeout tenendo conto degli ACK duplicati: ACK che riscontrano un segmento per il quale il mittente ha già ricevuto un riscontro. Quando TCP destinatario riceve un segmento con numero di sequenza più elevato del successivo, individua un buco nel flusso dei dati: TCP ritrasmette un ACK per l'ultimo byte di dati in ordine ricevuto.

Nel caso vengano ricevuti **un ACK originale + tre ACK duplicati**, TCP fa un **fast retransmit**, ritrasmettendo il segmento mancante prima dello scadere del timer.



3.5.5 Controllo del flusso

TCP fornisce un servizio di **controllo del flusso** alle sue applicazioni per eliminare la possibilità da parte del mittente di riempire completamente il buffer del destinatario.

La dimensione del buffer spesso viene decisa dal processo applicativo, ma il SO può modificare la dimensione in base alle necessità.

TCP richiede al mittente di gestire una variabile detta **finestra di ricezione**.

Quando host A deve inviare un file a host B, B alloca un receive buffer (L = lunghezza buffer). B_{read} è l'ultimo byte nel flusso di dati letto dall'applicazione sull'host B; B_{rcv} è l'ultimo byte che è stato piazzato nel buffer. Quindi, $B_{rcv} - B_{read} \leq L$. La **finestra di ricezione** W è grande quanto lo spazio libero nel buffer (è quindi **dinamica**). B fa sapere ad A quanto spazio rimane nel buffer mettendo il valore di W nel campo corrispondente di ogni segmento inviato ad A.

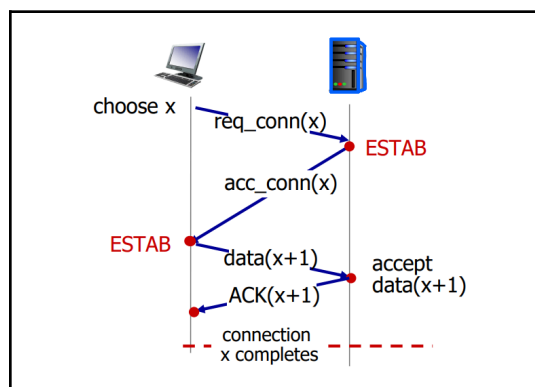
Host A tiene traccia di due variabili, B_{sent} (ultimo byte inviato) e B_{ack} (ultimo byte riscontrato). $B_{sent} - B_{ack}$ è la quantità di dati non riscontrati che A ha inviato a B, quindi per evitare overflow bisogna che $B_{sent} - B_{ack} \leq W$.

TCP invia segmenti contenenti il valore di W anche se il buffer è pieno e non ha ulteriori dati da inviare, per evitare un blocco.

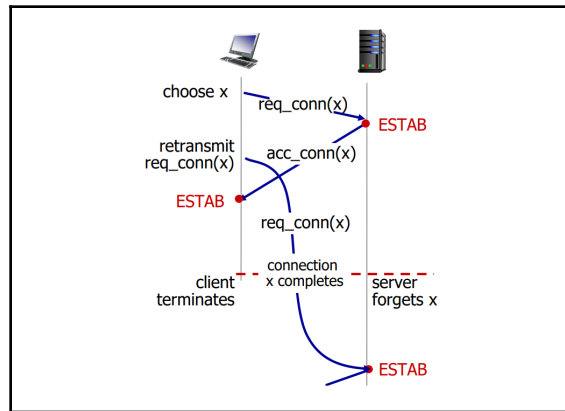
3.5.6 Gestione della connessione TCP

3.5.6.1 2-way handshake

1. Scenario senza problemi:



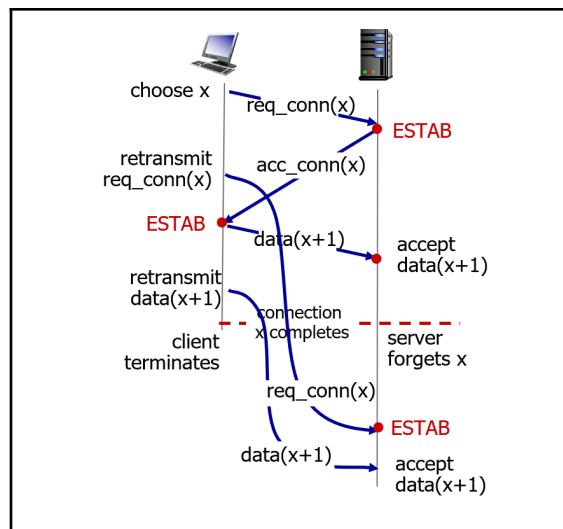
2. Problema, connessione aperta per metà (no client):



Il client è morto, non esiste più la connessione. Rischio di saturare le risorse con connessioni aperte per metà.

Inoltre, rischio che un attaccante sfrutti queste connessioni aperte senza client.

3. Problema: dati duplicati accettati:

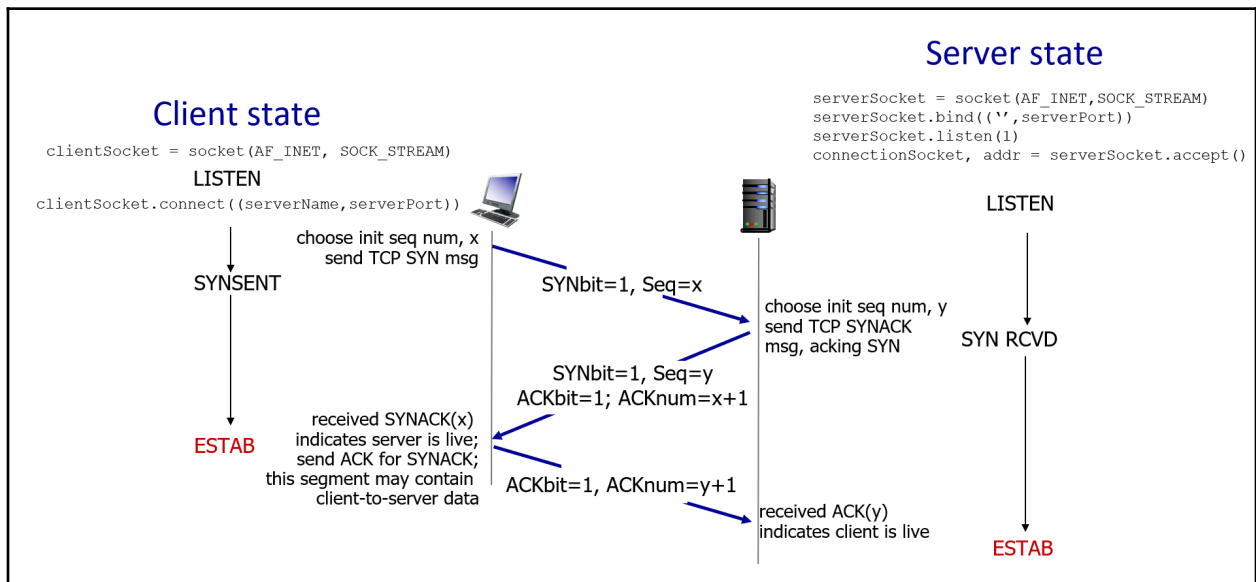


Non vengono inviati ACK, ma il server accetta dati duplicati. Il server accetta una connessione senza che il client abbia fatto un ESTAB.

3.5.6.2 3-way handshake

Quando un client vuole stabilire una connessione con un server:

1. Il lato client di TCP invia uno speciale segmento TCP al lato server di TCP. Questo segmento non contiene dati del livello applicazione. Una delle flag dell'header, il bit **SYN**, è impostato a 1. Il client inoltre sceglie un numero di sequenza iniziale e lo mette nel campo corrispondente.
2. Quando il datagramma IP contenente il segmento TCP SYN giunge al server, il server estrae il segmento dal datagramma, alloca i buffer ed invia un segmento di connessione stabilita al client. Neppure questo segmento contiene dati dell'applicazione, ma ha il bit SYN impostato ad 1, campo di ACK impostato a **numero di sequenza** inviato dal client + 1 e contiene numero di sequenza scelto dal server. Questo segmento è chiamato segmento **SYNACK**.
3. Alla ricezione del segmento SYNACK, il client alloca buffer per la connessione e invia un riscontro al server: questo segmento ha bit **SYN impostato a 0**, dato che la connessione è stata stabilita.



Quando il processo applicativo vuole chiudere la connessione, il client TCP invia un segmento con **bit FIN** impostato a 1 al server. Quando il server riceve questo segmento, invia un riscontro al client ed invia un segmento con bit FIN a 1. Il client riscontra il segmento e le **risorse** sono **deallocate**.

Durante una connessione TCP, il protocollo in esecuzione sugli host transiziona tra diversi stati TCP.

- Il client inizia in **stato CLOSED**.
- Dopo l'invio di un segmento SYN, passa allo **stato SYN SENT**.
- Alla ricezione del riscontro dal server, il client passa allo **stato ESTABLISHED**:
 - mentre è in questo stato, il client può inviare e ricevere segmenti TCP contenenti dati applicativi.
- Quando il client vuole terminare una connessione:
 - invia segmento FIN e passa allo **stato FIN WAIT 1**.
 - Quando riceve il riscontro dal server, passa allo **stato FIN WAIT 2** ed attende segmento FIN dal server e
 - quando lo riceve passa allo **stato TIME WAIT**, che permette al client di rinviare il riscontro finale nel caso sia stato perso. Dopo l'attesa, la connessione viene chiusa.

Se un host riceve un segmento TCP indirizzato ad una **porta chiusa**, invia un segmento con il **bit RST** impostato a 1.

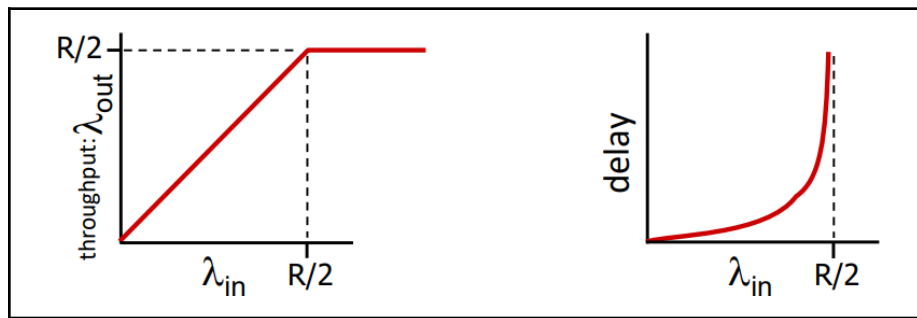
3.6 Controllo della congestione

3.6.1 Cause e costi della congestione

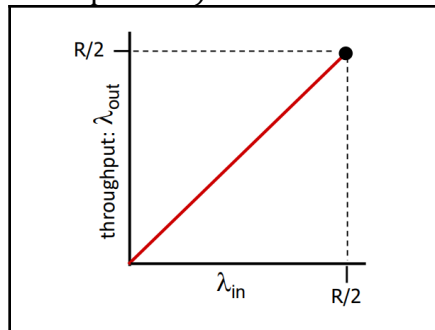
1. Lo **scenario più semplice** è quello in cui si ha un **singolo router** e **buffer infiniti**.

I pacchetti passano attraverso il router su un collegamento di capacità R; il router ha buffer che gli permettono di memorizzare pacchetti in caso la loro velocità di arrivo ecceda la capacità del collegamento.

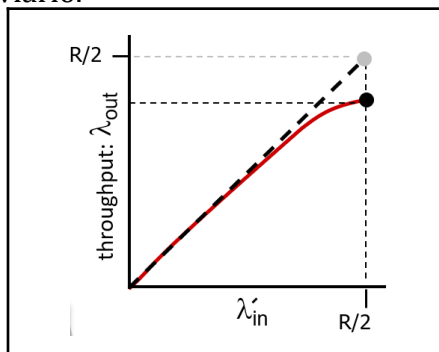
Quando la velocità di trasmissione supera $\frac{R}{2}$, il throughput sarà sempre $\frac{R}{2}$, ma il numero di pacchetti in coda sarà illimitato e l'attesa sarà infinita. Quindi, quando il **tasso di trasmissione** dei pacchetti si avvicina alla **velocità di trasmissione** del collegamento si verificano **ritardi di accodamento**.



2. Nel **secondo scenario**, i buffer sono finiti, e i pacchetti persi sono ritrasmessi. Se il mittente ritrasmette solo i pacchetti che sa essere stati persi, il tasso di trasferimento sarà $\frac{R}{3}$: il mittente deve ritrasmettere per compensare i pacchetti perduti, ed eventuali ritrasmissioni non necessarie sprecano banda del collegamento (riducendo ulteriormente il tasso di trasmissione dei dati).
Idealmente: se avessimo una conoscenza assoluta: $\lambda_{in} = \lambda'_{in}$ (Il mittente invia solo quando i buffer del router sono disponibili).

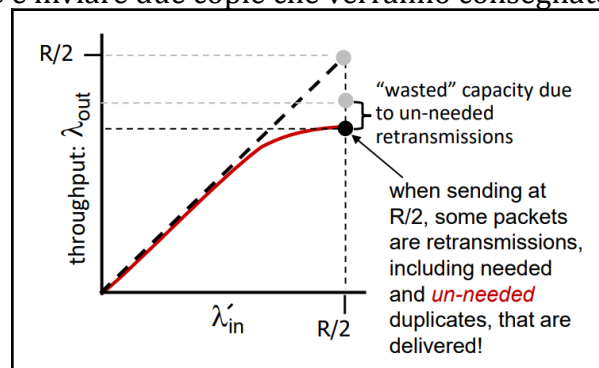


Nel caso in cui un pacchetto dovesse perdersi il mittente saprebbe quando un pacchetto è stato perso e potrebbe riinviarlo.



Scenario realistico: duplicati non necessario

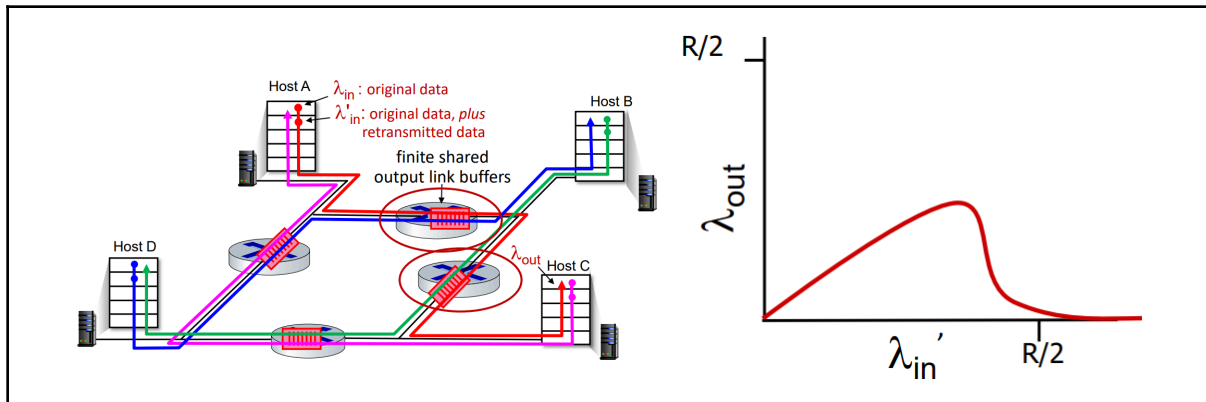
I pacchetti possono essere persi ma anche arrivare in ritardo, ma al mittente può scadere il timeout prematuramente e inviare due copie che verranno consegnate entrambe.



3. Nel **terzo scenario**, i mittenti sono 4 e sono presenti più router.

All'aumento del traffico di dati trasmessi da tutti i mittenti, il throughput di 2 mittenti diminuisce, data la limitazione dei buffer. Inoltre, quando un pacchetto viene perso, la capacità di trasmissione che era stata usata ad ogni router per inoltrare quel pacchetto fino al punto in cui viene perso viene effettivamente sprecata.

A cascata posso avere congestione e causare un deadlock.



3.6.2 Approcci al controllo della congestione

End-to-end:

Nessun feedback esplicito dalla rete. La congestione viene dedotta osservando le perdite e i ritardi.

Approccio utilizzato da TCP vanilla, in quanto IP non fornisce feedback sulla congestione della rete.

Assistito dalla rete:

I router forniscono **feedback diretti** agli host mittenti/destinatari con flussi che transitano da router congestionati. Potrebbe indicare il livello di congestione o settare esplicitamente il rate di invio.

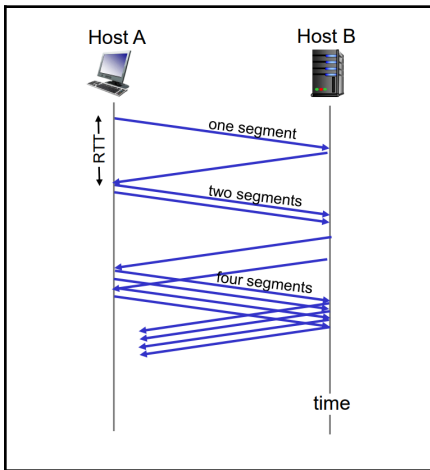
3.7 Controllo della congestione con TCP

3.7.1 Controllo della congestione classico

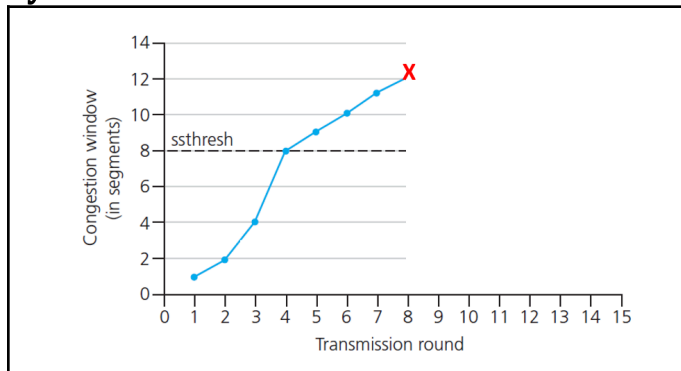
L'approccio di TCP è **limitare il tasso** con il quale invia traffico sulla sua connessione in base alla **congestione percepita**.

Il meccanismo di controllo di TCP tiene traccia di una variabile detta **finestra di congestione** (cwnd), che impone un limite sul flusso di traffico che un mittente TCP può inviare sulla rete: la quantità di dati non riscontrati non può eccedere il minimo tra finestra di congestione e finestra di ricezione. La congestione viene identificata dalla perdita di pacchetti (i buffer dei router sono pieni), che si verifica quando avviene un timeout o vengono ricevuti tre ACK ripetuti: questa tecnica è chiamata self clocking.

Quando una connessione TCP viene stabilita, il valore di cwnd è tipicamente inizializzato ad un valore basso ed incrementato ogni volta che viene ricevuto un ACK, quindi raddoppiando ogni RTT (**slow start**).

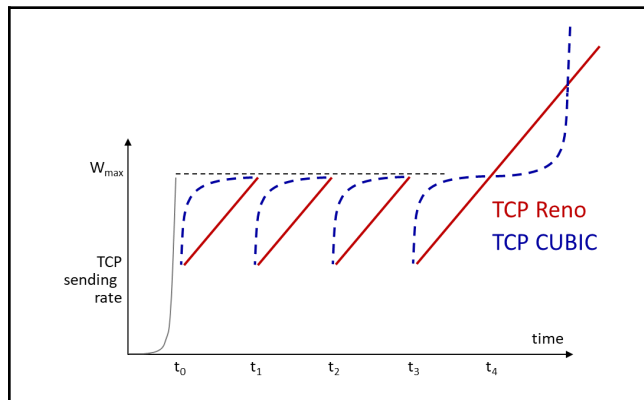


Quando si verifica una **perdita**, indicata dal timeout, il mittente resetta il valore di $cwnd$ ed il processo di slow-start ricomincia. Inoltre viene impostata una variabile di limite $ssthresh = \frac{cwnd}{2}$: se questo limite viene superato, TCP incrementa più cautamente la sua $cwnd$ (crescita lineare invece che esponenziale) (**congestion avoidance**). Se la perdita si verifica a causa di **triplo ACK ripetuto**, TCP dimezza il valore di $cwnd$ e pone il valore del limite a metà del valore che $cwnd$ aveva quando sono stati ricevuti gli ACK ed entra nella modalità **fast-recovery**.



Il controllo della congestione di TCP (in caso di tripli ACK) consiste quindi in una crescita lineare (additiva) e poi dimezzamento: per questo motivo è anche chiamato **additive-increase, multiplicative-decrease (AIMD)**.

Esiste un metodo migliore per testare la larghezza di banda. **TCP CUBIC** funziona in maniera simile ad AIMD, ma aumenta la dimensione della sua finestra di congestione in funzione del cubo della distanza tra il tempo attuale ed il tempo in cui la finestra raggiungere di nuovo il limite. Il throughput medio di una connessione è: $\frac{0.75 * W}{RTT}$.



3.7.2 Controllo della connessione assistito

L'**Explicit Congestion Notification** è la forma di controllo della congestione attuata in Internet.

A livello della rete, **due bit nel campo ToS** dell'header del datagramma IP sono utilizzati per **ECN**: il primo bit è usato dal router per indicare **presenza di congestione**, il secondo è usato per indicare che mittente e destinatario sono **capaci di utilizzare ECN**. Quando TCP nel destinatario riceve una notifica di congestione, questo informa il TCP nel mittente; il mittente reagisce **dimezzando la finestra di congestione**, e notifica il destinatario della riduzione.

TCP(classico, CUBIC) aumentano il rateo di invio di TCP fino a quando un pacchetto non viene perso nell'output di qualche router: il bottleneck link. Una volta compresa la congestione è utile concentrarsi sul link che fa da collo di bottiglia.

Con il **Delay-based Congestion Control**, il mittente misura il RTT di tutti i pacchetti riscontrati.

Il throughput della rete non congestionata è $\frac{cwnd}{RTT_{min}}$: se il throughput misurato è molto simile al throughput non congestionato, la finestra viene aumentata linearmente, altrimenti viene diminuita linearmente ("Keep the pipe full, but not fuller").

Mantenere una pipe tra il mittente e il ricevitore, abbastanza piena ma non completamente, mantiene il link che fa bottleneck impegnato, ma **evita un buffering o un ritardo alto**.

3.7.3 Equità

Equità: se K sessioni TCP condividono lo stesso collegamento collo di bottiglia con larghezza di banda R , ognuno di essi dovrebbe avere un tasso di trasferimento $\frac{R}{K}$.

TCP è corretto (in condizioni ideali) in quanto AIMD bilancia automaticamente la larghezza di banda delle connessioni.

4 Livello rete

4.1 Introduzione al livello rete

Il livello rete è composto da due parti che interagiscono tra loro, piano dati e piano controllo.

Il livello rete è l'ultimo livello di molti dispositivi di rete (router).

Mittente: incapsula i segmenti nel datagramma e lo passa al livello link

Ricevitore: consegna i segmenti al protocollo del livello trasporto

Router: esaminano i campi dell'header di tutti i datagrammi IP che passano attraverso.

Muove i datagrammi dalle porte di input alle porte di output per far proseguire il viaggio al datagramma.

4.1.1 Forwarding e routing

- **Forwarding (inoltro)**: muovere un pacchetto da un router al giusto collegamento in uscita
- **Routing (instradamento)**: la rete deve determinare il **percorso** che i pacchetti devono seguire per fluire da mittente a destinatario
- **Piano dati**: funzione locale per ogni router. Determina come il datagramma in arrivo sulla porta di input del router viene inoltrato alla porta di output
- **Piano controllo**: logica della rete. Determina come il datagramma viene instradato lungo la rete dalla sorgente all'host destinatario.

- Algoritmo di **routing tradizionale**: I router si mettono d'accordo tra di loro sulle strade che devono prendere i pacchetti.
Adatto per reti di grandi dimensioni.
- Software-defined networking (**SDN**): Un entità centrale (server remoto), che ha visione della rete sottostante, decide i percorsi di tutti i pacchetti. Problemi: se il server muore perdo tutto (single point of failure), potrei anche avere della congestione del server.
Adatto per reti di piccole dimensioni, e dove non è critica l'interruzione del server

Un elemento chiave in ogni router è la **forwarding table**: un router inoltra un pacchetto esaminando i valori dell'header del pacchetto e li usa per indicizzare nella sua forwarding table.

4.1.2 Network Service Model

Il **network service model** definisce le caratteristiche della consegna dei pacchetti tra mittente e destinatario. Il livello rete fornisce un singolo servizio, detto servizio **best-effort**: non vi sono garanzie sulla consegna dei pacchetti, tempo e ordine di consegna o larghezza di banda.

Ciò è stato fatto per rendere la rete semplice. Se si vogliono dei servizi ulteriori vanno aggiunti al di sopra. L'evoluzione della banda ha ridotto i possibili problemi.

4.2 Cosa c'è dentro un router?

Si possono identificare 4 componenti in un router:

- **Porte di input**. Una porta di input svolge, a livello fisico, la funzione di terminare il collegamento fisico al router; inoltre, a livello collegamento, svolge le funzioni richieste per interoperare con il livello collegamento dall'altra parte del collegamento in arrivo. Qui la forwarding table viene consultata per determinare la porta d'uscita.
- **Matrice di commutazione**. La matrice di commutazione connette le porte di input alle porte di output.
- **Porte di output**. Una porta di output memorizza pacchetti ricevuti dalla matrice di commutazione e li trasmette al collegamento in uscita.
- **Processore di routing**. Il processore di routing svolge funzioni del piano di controllo. Nei router tradizionali, esegue i protocolli di routing; nei router SDN comunica con il controllore remoto.

Queste componenti sono implementate di HW, mentre le funzioni del piano di controllo sono implementate in SW.

4.2.1 Destination-based forwarding

La forwarding table è copiata dal processore di routing alle schede di linea tramite un bus separato; in questo modo, le decisioni di inoltra sono svolte localmente, senza invocare il processore principale. Il router confronta il prefisso dell'indirizzo di destinazione del pacchetto con le voci della tabella: se c'è corrispondenza, il router inoltra il pacchetto sul collegamento associato a quel prefisso. Quando ci sono più corrispondenze, il router segue la regola del **prefisso più lungo**.

Quando la porta d'uscita di un pacchetto è stata determinata, può essere inviato alla matrice di commutazione.

Destination Address Range				Link interface
11001000	00010111	00010***	*****	0
11001000	00010111	00011000	*****	1
11001000	00010111	00011***	*****	2
otherwise				3

11001000	00010111	00010110	10100001	which interface? 0
11001000	00010111	00011000	10101010	which interface? 1

4.2.2 Switching

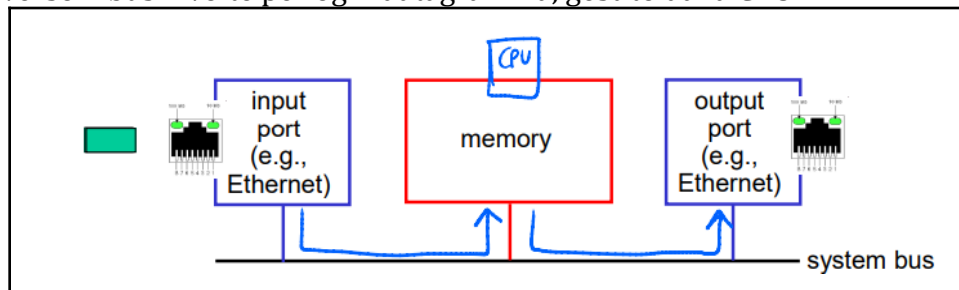
Switching rate: ratio a cui ogni pacchetto può essere trasferito dall'input all'output. N input, R velocità del link, $\text{rate} = N * R$.

Lo switching può essere implementato in vari modi:

- **Switching tramite memoria.** Le porte di input e output funzionano come tradizionali porte I/O.

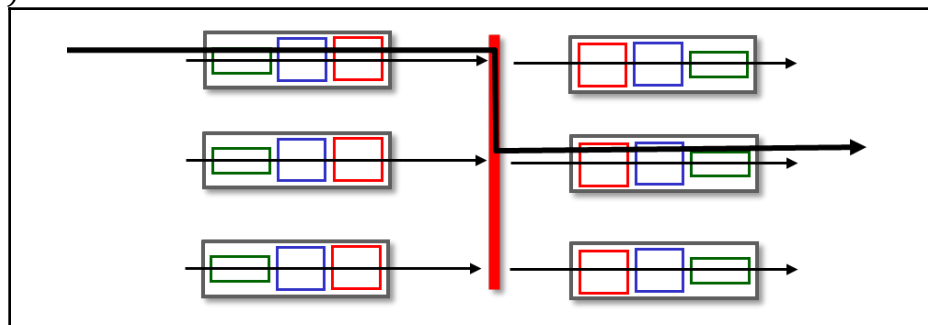
I pacchetti vengono copiati nella memoria di sistema. La velocità è limitata dalla banda della memoria.

Attraverso il bus 2 volte per ogni datagramma, gestito dalla CPU



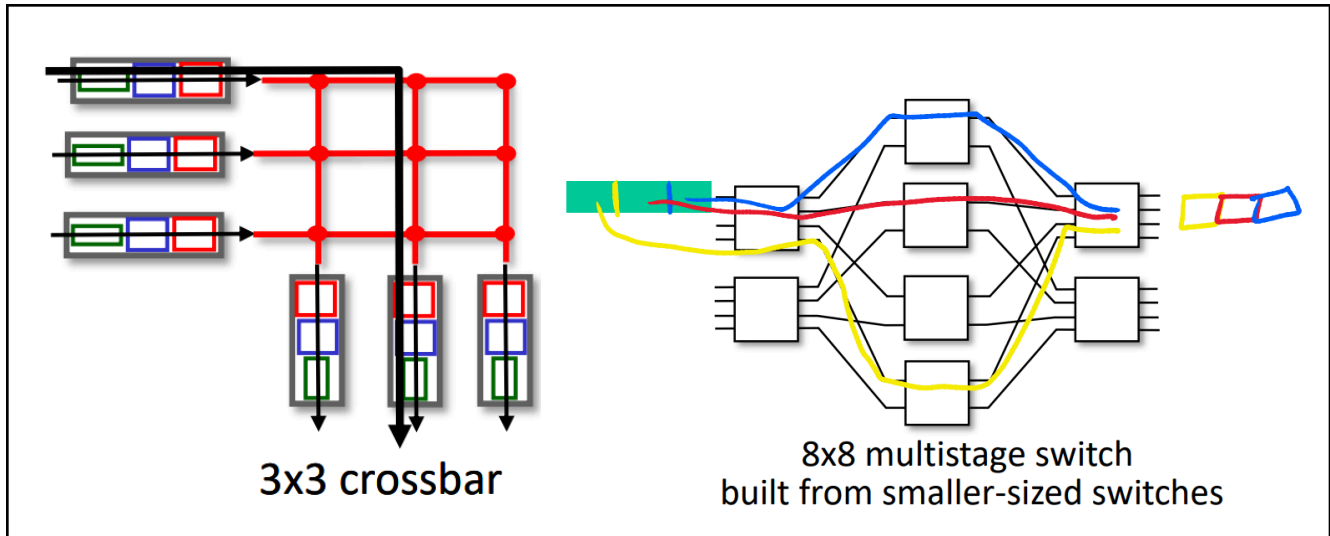
- **Switching tramite bus.** Le porte di input trasferiscono i pacchetti direttamente alle porte di output tramite bus condiviso, senza intervento del processore; tutte le porte ricevono il pacchetto, ma solo la porta corretta lo terrà. La velocità di switching è limitata dalla larghezza di banda del bus.

Il bus viene utilizzato 1 volta ma comunque devo accedere al bus. Nel caso in cui ci dovesse essere congestione devo aspettare (1 pacchetto alla volta). Buono per i router di accesso (domestici).



- **Switching tramite rete interconnessa.** Viene sfruttato parallelismo per evitare

bottlenecking.



4.2.3 Dove avviene l'accodamento

Se la matrice di commutazione è più lenta delle porte di input, si può verificare l'accodamento alle porte di input.

La matrice può trasmettere un solo pacchetto destinato ad una determinata porta per volta: fenomeno del **head-of-line blocking**.

Se, nelle porte di output, non vi è abbastanza memoria nei buffer per memorizzare un pacchetto in arrivo, deve essere presa la decisione di ignorare quel pacchetto (**drop-tail**) oppure di eliminare un pacchetto nella coda.

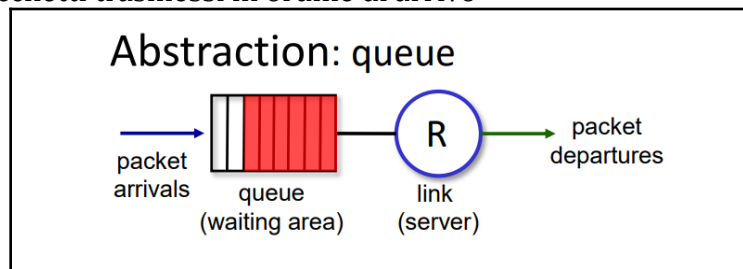
Il buffering richiesto è pari a $(C = \text{capacità trasmissiva}, N = \text{flussi TCP}) = \frac{RTT * C}{\sqrt{N}}$ ("La dimensione del buffer è come il sale un po' va bene ma troppo no")

Troppo buffering può causare ritardi, TCP potrebbe rimandare il pacchetto se ci fosse troppo buffering.

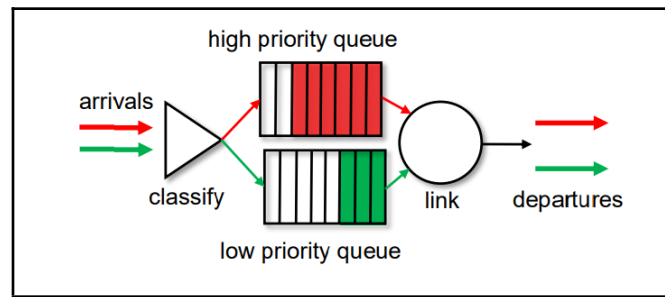
4.2.4 Scheduling dei pacchetti

Esistono quattro schemi di scheduling:

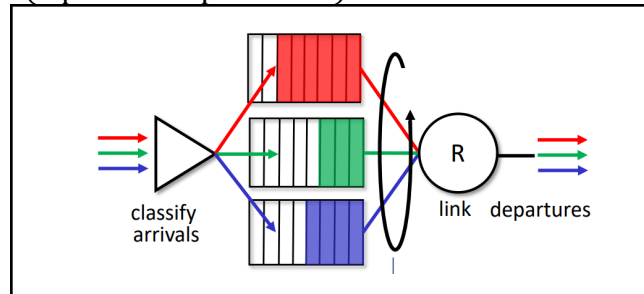
1. **FCFS (FIFO):** Pacchetti trasmessi in ordine di arrivo



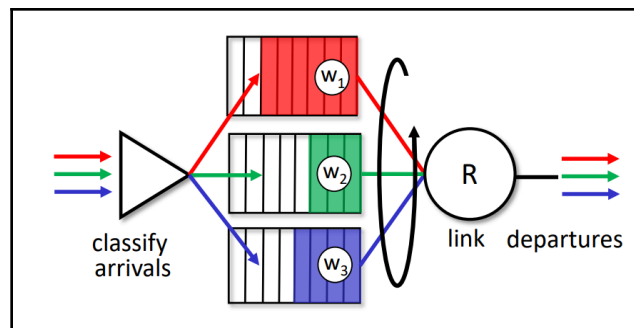
2. **Priorità:** Il traffico in arrivo viene classificato e diviso in più code in base alla loro priorità. Prova prima a svuotare le code con priorità più alta. I pacchetti con priorità bassa potrebbero rimanere in attesa all'infinito.



3. **Round robin:** Il traffico in arrivo viene classificato e diviso in più code. Le code vengono svuotate ciclicamente (1 pacchetto per classe)



4. **Weighted Fair Queuing (WFQ):** generalizzazione del Round Robin. Ogni classe i ha peso w_i ed ottiene un importo ponderato del servizio per ogni ciclo: $\frac{w_i}{\sum_j w_j}$



4.3 Internet Protocol

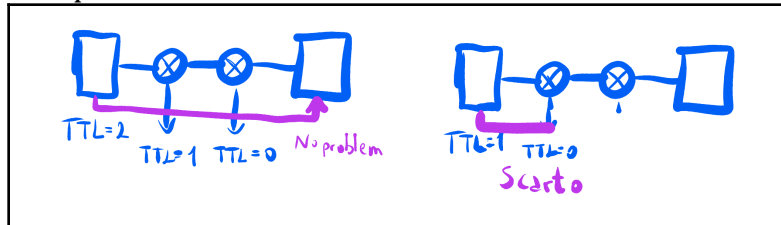
Due versioni di IP in uso: IPv4 e IPv6.

4.3.1 Formato del datagramma IPv4

Campi chiave di un datagramma IP:

- **Versione.** 4 bit definiscono la versione del datagramma, IPv4 o IPv6.
- **Lunghezza dell'header.** Determina il punto in cui inizia il payload vero e proprio.
- **Tipo di servizio.** Per distinguere un tipo di datagramma da un altro. Serve ai router per capire quale servizio fornire.
Valori da 0 a 7, (0:5) **diffserv**, servizi differenti per ogni flusso, che priorità dare al flusso. (6:7) **ECN**, Explicit Congestion Notification, segnalare all'utente che c'è congestione. Allora l'host potrebbe diminuire la congestion window. Usato soprattutto nei data center, perché gestisco sia i router che gli host.
- **Lunghezza del datagramma.** Massimo 2^{16} byte. Tipicamente 1500 byte (lunghezza massima di ethernet) o meno. Un pacchetto più piccolo è più efficiente per identificare gli errori.

- **Identificatore.** 16 bit per identificare il messaggio
- **flags:** segnala se fare frammentazione o se è già frammentato
- **offset di frammentazione.** Un datagramma IP di grandi dimensioni può essere diviso in datagrammi più piccoli e poi essere riassemblato grazie alle informazioni contenute in questi campi. L'offset serve a capire quale parte della frammentazione è.
- **Time-to-live.** Numero di pacchetti che il pacchetto può attraversare, quasi sempre settato al massimo possibile, 2^8 . Serve quando in condizioni critiche il pacchetto potrebbe ritrovarsi in un loop.



- **Upper layer.** Indica a quale protocollo del livello trasporto deve essere passato questo datagramma. (TCP, UDP) Serve per programmare i router
- **Checksum** dell'header.
- **Indirizzi IP sorgente e destinazione.** 32 bit
- **Opzioni.** Aggiungere funzionalità (quasi sempre non utilizzato)
- **Payload** (dati).

4.3.2 Indirizzamento

Il confine tra host e il livello fisico è detto **interfaccia**.

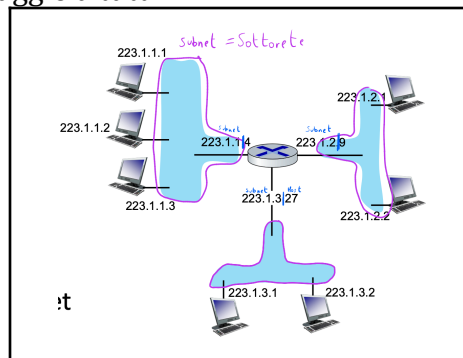
Un router possiede più interfacce, una per ogni suo collegamento.

Un indirizzo IP è associato ad un interfaccia, non all'host o al router che la contengono.

Una **subnet** è un'insieme di interfacce connesse tra di loro senza dover passare da un router. Dispositivi su una stessa subnet hanno gli **high-order bit** del loro indirizzo IP in comune (il numero di bit comuni è rappresentato con slash-notation).

L'indirizzo IP la cui parte dell'host è **tutta a 0** non rappresenta nessuna interfaccia ma la rete stessa.

L'indirizzo con **tutti 1** nella parte dell'host è il **Broadcast** che identifica tutti i dispositivi della subnet. Posso inviare un messaggio a tutti.



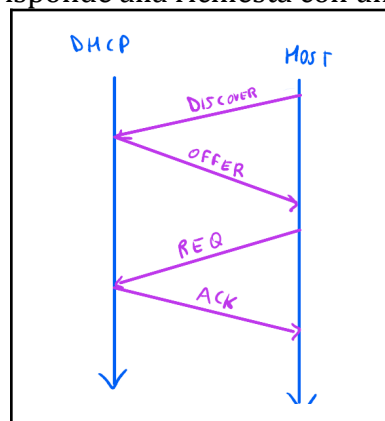
La strategia di assegnamento degli indirizzi di Internet è detta **CIDR**: l'indirizzo IP a 32 bit è diviso in due parti e ha la forma $a.b.c.d/x$, dove x indica il numero di bit nella prima parte dell'indirizzo. Gli x bit più significativi costituiscono la porzione della rete dell'indirizzo IP, e vengono chiamati **prefisso dell'indirizzo**.

Gli indirizzi degli host possono essere configurati manualmente oppure essere dinamicamente e automaticamente assegnati tramite Dynamic Host Configuration Protocol (**DHCP**).

DHCP è un protocollo client-server. Il client è tipicamente un host appena arrivato che vuole ottenere un indirizzo IP. Se la subnet non ha un server DHCP, dovrà esserci un router che conosce l'indirizzo di un server DHCP.

Quattro fasi per ottenere informazioni di rete:

- **Scoperta del server DHCP.** Il primo passo per un host è trovare un server DHCP con il quale interagire. Per farlo, l'host invia un messaggio di scoperta DHCP (**UDP** sulla **porta 67**). Il client DHCP crea un datagramma IP contenente questo messaggio, l'indirizzo broadcast 255.255.255.255 e l'indirizzo personale 0.0.0.0. Il client poi passa il datagramma al livello di collegamento, che a sua volta lo invia a tutti i nodi della subnet.
- **Offerte del server DHCP.** Quando un server DHCP riceve un messaggio di scoperta, risponde al client inviando un'offerta DHCP a tutti i nodi della subnet. Ogni messaggio d'offerta contiene l'ID del messaggio di scoperta ricevuto, l'indirizzo IP proposto al client, la network mask e la data di scadenza dall'indirizzo IP.
- **Richiesta DHCP.** Il client sceglie tra le offerte DHCP ricevute e risponde all'offerta scelta con un messaggio di richiesta. (In certe reti ci possono essere più DHCP e l'host deve accettare un solo indirizzo)
- **DHCP ACK.** Il server risponde alla richiesta con un messaggio di acknowledgement.



Gli ISP ricevono blocchi di indirizzi da un'associazione, **ICANN**. Gli ISP poi provvedono a suddividere il prefisso più grande in prefissi più piccoli da distribuire ai clienti

4.3.3 NAT

La **NAT** (Network Address Translation) è una tecnica utilizzata nei router e nei firewall per consentire a più dispositivi di **condividere un unico indirizzo IP pubblico** per accedere a Internet. Un router NAT non appare come un router al resto del mondo, ma come un singolo dispositivo con un **singolo indirizzo IP**. Tutti i datagrammi in uscita da una rete locale hanno lo stesso indirizzo IP, ma numeri di porta diversi. Per indirizzare i datagrammi in arrivo al router viene usata una **tabella di traduzione NAT**.

Il router NAT deve rimpiazzare l'indirizzo IP sorgente ed il numero di porta di ogni datagramma in uscita con indirizzo IP NAT e nuovo numero di porta; ricordarsi (nella tabella di traduzione NAT) ogni traduzione effettuata; rimpiazzare l'indirizzo IP NAT ed il numero di porta di ogni datagramma in entrata con indirizzo IP sorgente e nuovo numero di porta.

NAT è controverso, in quanto i router non dovrebbero manipolare i datagrammi e il problema del numero di indirizzi dovrebbe essere risolto da IPv6.

NAT Traversal si riferisce a tecniche che consentono a dispositivi situati dietro un router NAT di comunicare con successo con dispositivi esterni alla rete, come server o altri dispositivi su Internet.

Poiché la NAT modifica gli indirizzi IP e i numeri di porta nei pacchetti di dati, può creare problemi quando si tenta di stabilire connessioni bidirezionali o di ricevere connessioni in

ingresso da Internet. Le tecniche di NAT Traversal, come **UPnP** (Universal Plug and Play) o protocolli specifici come **STUN** (Session Traversal Utilities for NAT), **ICE** (Interactive Connectivity Establishment) e **TURN** (Traversal Using Relay NAT), consentono di superare queste limitazioni e stabilire connessioni bidirezionali attraverso router NAT.

4.3.4 IPv6

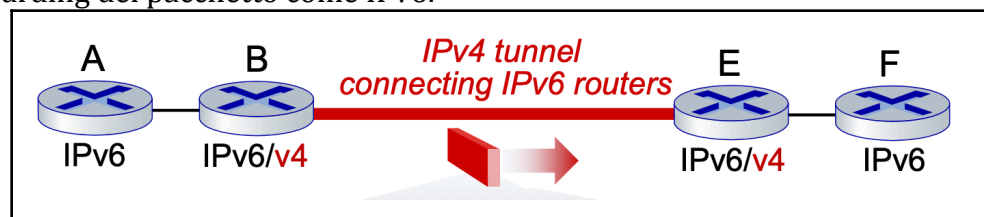
Con IPv6, gli indirizzi hanno lunghezza di **128 bit** e gli header sono grandi **sempre 40 byte**. I campi di un datagramma IPv6 sono:

- **Versione.**
- **Classe di traffico.** Può essere usato per dare priorità ad alcuni datagrammi in un flusso.
- **Etichetta del flusso.** Identifica il flusso dei datagrammi. Diverso da IP. Ci pensa l'operatore ad identificare il flusso.
- **Lunghezza del payload.**
- **Prossimo header.** Identifica il protocollo al quale i contenuti del datagramma devono essere consegnati.
- **Hop limit.**
- **Indirizzi IP sorgente e destinazione.**
- **Payload.**

Non vi sono campi di checksum (perché è compito dei protocolli del livello trasporto e collegamento), frammentazione (IPv6 non la supporta) e opzioni.

L'approccio adottato per il passaggio da IPv4 a IPv6 coinvolge il **tunneling**: il datagramma IPv6 è trasmesso come payload in un datagramma IPv4.

Tutto ciò che è all'edge della rete inizia ad essere IPv6, invece il core rimane prevalentemente IPv4. Il pacchetto IPv4 ha come indirizzo il router che si occuperà di spaccettare il pacchetto per continuare il forwarding del pacchetto come IPv6.



4.4 Inoltro generalizzato e SDN

Nell'inoltro generalizzato, una **tabella di match-plus-action** generalizza la nozione della tabella di inoltro basata sulla destinazione. Solitamente, le capacità di match-plus-action sono implementate tramite controllore remoto che gestisce le tabelle.

Ogni voce della tabella match-and-action, anche detta tabella di flusso, include:

- Un insieme di campi di header ai quali un pacchetto in arrivo sarà abbinato.
- Un insieme di contatori, aggiornati quando i pacchetti vengono abbinati alle voci della tabella.
- Un insieme di azioni da eseguire quando un pacchetto viene abbinato

4.4.1 Match

L'astrazione di match di **OpenFlow** ammette abbinamenti su campi di header di protocolli appartenenti a tre diversi livelli.

4.4.2 Action

Ogni voce nella tabella di flusso ha una lista di zero o più azioni che è possibile eseguire su un

pacchetto. Le azioni più importanti sono: **inoltro**, **scarto** e **modifica**.

5 Livello rete: piano di controllo

5.1 Introduzione

Il piano di controllo si preoccupa del routing, ovvero determinare il percorso che i pacchetti devono compiere per giungere alla loro destinazione. Esistono due approcci per strutturare il piano di controllo: tradizionale router a router e controllo centralizzato (definito da software).

5.2 Protocolli di routing

L'obiettivo dei protocolli di routing è **determinare il percorso migliore da mittente a destinatario**.

Per formulare i problemi di routing vengono utilizzati i **grafi**: i nodi del grafo rappresentano i router e gli archi i collegamenti fisici tra essi. Gli algoritmi di routing possono essere classificati in **algoritmi centralizzati** e **algoritmi decentralizzati**.

Un algoritmo centralizzato computa il percorso meno costoso avendo **conoscenza completa della rete** a priori. Questo tipo di algoritmi vengono chiamati **algoritmi link-state** (LS).

Un algoritmo decentralizzato computa il percorso in maniera **iterativa**, distribuita tra i router. Nessun nodo ha conoscenza completa dei costi di tutti i collegamenti, ma lo acquisisce man mano scambiando informazioni con i nodi adiacenti.

Con algoritmi di routing statici, i percorsi cambiano lentamente nel corso del tempo, spesso a causa di interventi umani; gli algoritmi di routing dinamici cambiano i percorsi al variare della topologia della rete o del traffico.

Inoltre, gli algoritmi possono essere **load-sensitive** (i costi dei collegamenti variano dinamicamente per riflettere il livello di congestione) o **load-insensitive**.

5.2.1 Algoritmo LS

L'algoritmo di **Dijkstra** è centralizzato (conosce i costi dei collegamenti tra tutti i nodi, grazie al link state broadcast) e computa i percorsi con costo minore da un nodo sorgente a tutti gli altri nodi, creando una forwarding table per quel nodo; è iterativo, quindi dopo k iterazioni si conoscono i percorsi a k destinazioni. Quando i costi dei collegamenti dipendono dal volume del traffico, sono possibili delle variazioni di percorso.

Ogni router invia con chi è collegato e quale costo a tutti gli altri router della rete.

Complessità:

algoritmo = $O(n^2) \rightarrow O(n \log n)$ implementazioni più efficienti

messaggio = $O(n^2)$ Ogni router manda un messaggio in broadcast agli altri nodi

5.2.2 Algoritmo Distance Vector

L'algoritmo DV è **iterativo**, asincrono e **distribuito**.

E' distribuito nel senso che ogni nodo riceve informazioni da uno o più nodi adiacenti, esegue i calcoli e distribuisce i risultati ai suoi vicini.

E' iterativo nel senso che questo processo continua fino a quando non vi sono più informazioni da scambiare.

E' asincrono nel senso che non richiede che tutti i nodi operino in sincronia tra di loro.

Questo algoritmo è basato sull'equazione di **Bellman-Ford**: dopo aver percorso la distanza tra un nodo x e un nodo v , se si prende la distanza minima tra v e un nodo y , il costo del percorso sarà uguale alla distanza da x a v più la distanza dal nodo y (in questo caso, si prende la distanza minima). La soluzione all'equazione di Bellman-Ford fornisce le voci della forwarding table di un determinato nodo.

Di tanto in tanto, ogni nodo invia il proprio **distance vector stimato** ai propri vicini; quando un nodo riceve una nuova stima, aggiorna il suo distance vector usando l'equazione di Bellman-Ford: in condizioni normali, la stima delle distanze converge al costo attuale minimo del percorso. Quando il costo di un collegamento cambia, il nodo locale individua la variazione di costo e ricalcola il suo DV, notificando i vicini.

Si possono verificare dei routing loop se il costo aumenta (**count-to-infinity**), in quanto si avranno tante iterazioni di ricalcolo quanto è l'aumento del costo di collegamento.

Inoltre, se un router è malfunzionante e trasmette informazioni incorrette, il routing sarà a sua volta incorretto.

Complessità:

Dipende da quanti vicini ha ogni nodo

LS è un algoritmo più robusto. In DV gli errori si propagano (basta che un router annunci un costo sbagliato). In DV posso bloccare il traffico annunciando un costo basso con un router che non porta da nessuna parte e scarto tutti i pacchetti.

5.3 OSPF

Nel mondo reale, le reti sono complicate da due fattori: scala (non è possibile memorizzare tutte le destinazioni nelle tabelle di routing) e autonomia amministrativa. Questi problemi possono essere risolti aggregando i router in regioni dette **sistemi autonomi** (AS, oppure domini).

Ogni AS consiste di un gruppo di router sotto lo stesso controllo amministrativo. I router nello stesso AS eseguono tutti lo stesso algoritmo di routing (sistema di routing intra-autonomo) e si conoscono.

Al bordo di ogni dominio avrò dei router che mettono in comunicazione più AS **Inter-AS**.

Il routing **OSPF** e **IS-IS** sono largamente utilizzati per il routing intra-AS.

OSPF è un protocollo **link-state** che usa **flooding** delle informazioni link-state (direttamente su IP invece che usare TCP/UDP) e algoritmo di **Dijkstra**. Tutti i messaggi OSPF sono **autenticati** per garantire maggiore sicurezza. Ogni router esegue localmente l'algoritmo di Dijkstra per determinare il percorso più breve a tutti i suoi subnet. I costi di ogni collegamento sono configurati da un amministratore.

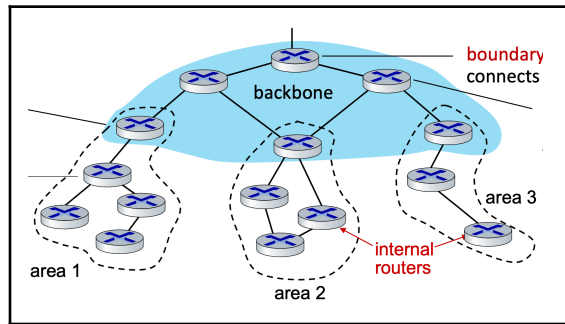
OSPF fornisce:

- **Sicurezza.** Gli scambi tra router OSPF possono essere autenticati; solo i router autenticati possono partecipare al protocollo OSPF in un AS.
- **Multipli percorsi dallo stesso costo.** OSPF permette di utilizzare più percorsi.
- **Supporto integrato per routing unicast e multicast.**
- **Supporto per gerarchia all'interno di un singolo AS.**

Gerarchia a due livelli intradominio:

- local area
- backbone

Il link-state advertisement avviene solo nella propria area



5.4 Routing tra ISP: BGP

Per instradare un pacchetto attraverso più AS serve un protocollo di routing inter-AS, il Border Gateway Protocol (**BGP**).

5.4.1 Il ruolo di BGP

BGP permette ad una subnet di **pubblicizzare la sua esistenza**, e le destinazioni che può raggiungere, al resto di Internet. Inoltre, determina il percorso migliore ai prefissi

5.4.2 Pubblicizzazione

Per ogni AS, ogni router può essere un router gateway o un router interno: un **gateway** è un router al bordo di una AS che si connette direttamente con uno o più router di altre AS; un **router interno** connette solo con altri host all'interno dell'AS.

Una connessione BGP che si estende su più AS è detta connessione **BGP esterna**, mentre una tra due router nella stessa AS è detta connessione **BGP interna**.

eBGP ottiene informazioni sulla raggiungibilità delle subnet dagli AS vicini; **iBGP** propaga queste informazioni a tutti i router interni all'AS.

Due router BGP si scambiano messaggi BGP su una connessione **TCP semi-permanente**.

I messaggi BGP sono:

- **OPEN**: apre una connessione TCP al peer BGP remoto e autentica il BGP mittente
- **UPDATE**: pubblica un nuovo percorso
- **KEEPALIVE**: mantiene la connessione viva in assenza di updates; inoltre manda ACK per la richiesta di open
- **NOTIFICATION**: riporta errori nei messaggi precedenti; usata anche per chiudere una connessione

5.4.3 Determinare i percorsi migliori

Quando un router pubblica un prefisso su una connessione BGP, include degli **attributi BGP**: l'insieme di **prefisso e attributi** è detto **percorso**.

I due attributi più importanti sono **AS-PATH** e **NEXT-HOP**:

- **AS-PATH** contiene la lista di AS attraverso il quale è passato il messaggio;
- **NEXT-HOP** è l'indirizzo IP dell'interfaccia del router che inizia l'AS-PATH.

L'algoritmo di **routing** più semplice è quello della **patata bollente**: viene scelto il **percorso più corto al prossimo NEXT-HOP**.

In pratica, BGP usa un algoritmo più complesso:

1. Ad un percorso viene assegnato un valore di preferenza locale. Vengono scelti i percorsi con valori più elevati.
2. Tra i percorsi rimasti, viene scelto quello con AS-PATH più corto.
3. Tra i percorsi rimasti, viene usato l'algoritmo della patata bollente.
4. Se rimane più di un percorso, il router usa identificatori BGP per selezionare un percorso.

5.4.4 IP anycast

BGP è spesso utilizzato per implementare il servizio **IP-anycast** usato comunemente nel DNS.

5.4.5 Politica di routing

Un gateway che riceve un messaggio di percorso usa la **politica di importazione** per accettare o declinare il messaggio.

La politica AS determina inoltre se pubblicizzare il percorso agli AS vicini o meno.

5.5 Il piano di controllo SDN

5.5.1 Perché usare una logica centralizzata per il piano di controllo?

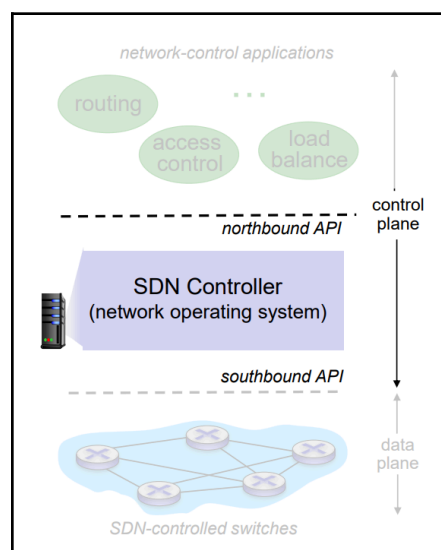
E' più semplice gestire la rete, evita configurazioni sbagliate dei router, più flessibile per il flusso del traffico

5.5.2 Switch del piano dati

- Switch semplici e veloci che implementano il forwarding del piano dati generalizzato sull'hardware
- Tabelle di flusso (inoltro) calcolate e installate sotto la supervisione di un controller superiore
- API per controllo degli switch basato su tabelle (Eg OpenFlow)
- Protocolli per comunicare con il controller (Eg OpenFlow)

5.5.3 Controller SDN (OS di rete)

- Mantiene le informazioni sullo stato della rete
- Interagisce con le applicazioni di controllo della rete ("Superiore"), tramite API northbound
- Interagisce con gli switch di rete ("Inferiore") tramite API southbound
- Implementato come un sistema distribuito per le performance, scalabilità, tolleranza ai guasti e robustezza

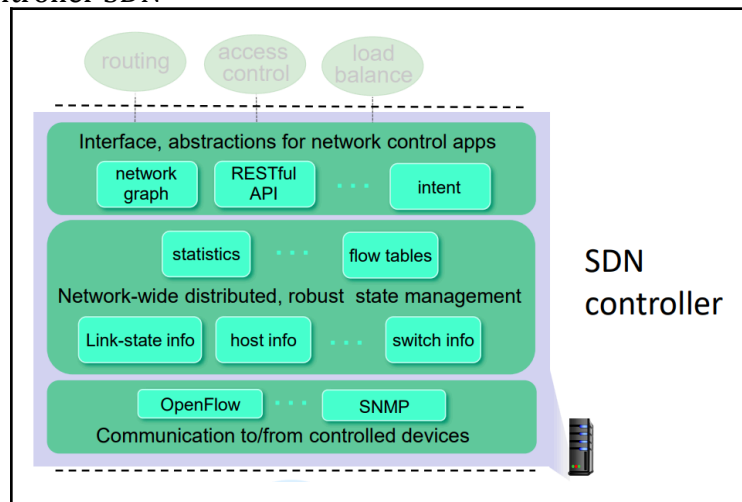


5.5.4 Applicazioni di controllo della rete

- Cervello del controllo: implementa le funzioni di controllo usando servizi di basso livello, API fornite dal controller SDN
- Disaggregato: può essere fornito da 3^e parti, distinte dal routing vendor o dal controller

SDN

5.5.5 Componenti di un controller SDN



5.5.6 Protocollo OpenFlow

Il protocollo OpenFlow opera tra il controller e gli switch; viene utilizzato **TCP** per i messaggi con l'opzione della crittazione.

Esistono tre classi di messaggi OpenFlow:

1. Controller to switch

- Features: il controller interroga le funzionalità dello switch, e lo switch risponde (Eg velocità, funzionalità)
- Configure: il controller interroga o setta i parametri di configurazione dello switch
- Modify-state: aggiunge, rimuove, modifica le entries della tabella di flusso di OpenFlow
- Packet-out: il controller può mandare questo packet out su una specifica porta dello switch

2. Asincrono (**switch to controller**)

- Packet-in: trasferisce i pacchetto al controller. Vede i messaggi di packet out da parte del controller
- Flow-removed: entry della tabella di flusso cancellate sugli switch
- Port status: informa il controller di un cambiamento di porta

3. Simmetrico

5.6 ICMP

Internet Control Message Protocol (**ICMP**) è usato da host e router per comunicare informazioni di livello rete tra di loro.

Tipicamente viene usato per **riportare errori** o per fare echo richiesta/risposta (per fare il ping)

I messaggi ICMP vengono trasportati all'interno di datagrammi IP.

I messaggi ICMP hanno campi **tipo** e **codice**, e contengono l'**header** e i **primi 8 byte del datagramma IP** che hanno causato la generazione del messaggio ICMP.

5.7 Mantenimento della rete e configurazione

- **CLI**: gli operatori scrivono direttamente codice sul dispositivo (dopo aver fatto accesso con ssh)
- **SNMP/MIB**: gli operatori interrogano e settano i dati dei dispositivi usando il protocollo Simple Network Management Protocol

- **NETCONF**: comunicazioni compatibili con YANG actions/data to/from/among dispositivi remoti
- **YANG**: linguaggio di modellazione stile xml

LABORATORIO

Byte order

I dati nei vari host possono essere salvati in ordine diverso (Big-endian, Little-endian)

htons(): host to network short

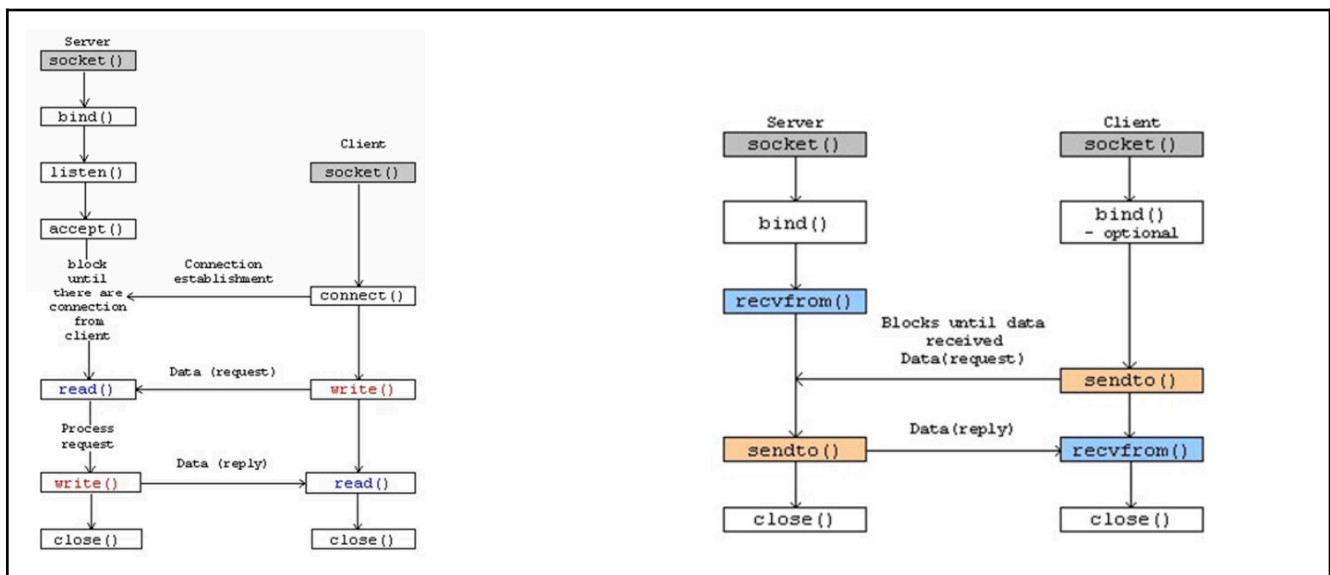
htonl(): host to network long

htohs(): network to host short

htohl(): network to host long

TCP

UDP



TCP

Client

```
simpleSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Questa funzione crea un socket TCP.

I parametri sono:

AF_INET: Indica che stiamo usando il protocollo IPv4.

SOCK_STREAM: Indica che stiamo usando un protocollo orientato alla connessione (TCP).

IPPROTO_TCP: Specifica l'uso del protocollo TCP.

```
returnStatus = connect(simpleSocket, (struct sockaddr *)&simpleServer, sizeof(simpleServer));
```

Questa funzione stabilisce una connessione con il server. I parametri sono:

simpleSocket: Il socket creato precedentemente.

simpleServer: La struttura che contiene l'indirizzo del server.

sizeof(simpleServer): La dimensione della struttura dell'indirizzo.

returnStatus = **read**(simpleSocket, buffer, sizeof(buffer));
Questa funzione legge i dati dal server. I parametri sono:

simpleSocket: Il socket connesso.
buffer: Il buffer dove memorizzare i dati letti.
sizeof(buffer): La dimensione del buffer.

Server

simpleSocket = **socket**(AF_INET, **SOCK_STREAM**, IPPROTO_TCP);

returnStatus = **bind**(simpleSocket, (struct sockaddr *)&simpleServer, sizeof(simpleServer));
Questa funzione associa il socket a un indirizzo e una porta specifici. I parametri sono:

simpleSocket: Il socket creato.
simpleServer: La struttura con l'indirizzo e la porta.
sizeof(simpleServer): La dimensione della struttura dell'indirizzo.

returnStatus = **listen**(simpleSocket, 5);
Questa funzione mette il socket in modalità di ascolto, permettendo al server di accettare connessioni. I parametri sono:

simpleSocket: Il socket associato.
5: La lunghezza della coda di connessioni pendenti.

simpleChildSocket = **accept**(simpleSocket, (struct sockaddr *)&clientName, &clientNameLength);
Questa funzione accetta una connessione in entrata. I parametri sono:

simpleSocket: Il socket in ascolto.
clientName: La struttura dove memorizzare l'indirizzo del client.
clientNameLength: La dimensione della struttura dell'indirizzo del client.

write(simpleChildSocket, &maxNum, sizeof(maxNum));

sscanf(buffer, "%d", &num)
Questa funzione estrae un numero intero dal buffer. I parametri sono:

buffer: Il buffer contenente i dati ricevuti.
"%d": Il formato per la lettura di un intero.
&num: Il puntatore dove memorizzare il numero letto.

close(simpleSocket);

UDP client

simpleSocket = **socket**(AF_INET, **SOCK_DGRAM**, IPPROTO_UDP);

sendto(simpleSocket, buffer, strlen(buffer), 0, (struct sockaddr *)&simpleServer, serverLen);
Questa funzione invia dati al server. I parametri sono:

simpleSocket: Il socket creato.
buffer: I dati da inviare.
strlen(buffer): La lunghezza dei dati.
0: Flag (spesso impostato a 0).
simpleServer: L'indirizzo del server.
serverLen: La lunghezza della struttura dell'indirizzo del server.

```
returnStatus = recvfrom(simpleSocket, buffer, sizeof(buffer), 0, (struct sockaddr *)&simpleServer,  
&serverLen);
```

Questa funzione riceve dati dal server. I parametri sono:

simpleSocket: Il socket creato.
buffer: Il buffer dove memorizzare i dati ricevuti.
sizeof(buffer): La dimensione del buffer.
0: Flag (spesso impostato a 0).
simpleServer: La struttura dove memorizzare l'indirizzo del server.
serverLen: La lunghezza della struttura dell'indirizzo del server.

```
close(simpleSocket);
```

Server

```
simpleSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);  
  
returnStatus = bind(simpleSocket, (struct sockaddr *)&simpleServer, sizeof(simpleServer));  
  
sendto(simpleSocket, buffer, strlen(buffer), 0, (struct sockaddr *)&clientName, clientNameLength);  
  
close(simpleSocket);
```