Appunti Reti Informatiche

Luca Seggiani
2025

1 Lezione del 23-09-25

1.1 Introduzione

Il corso si pone di presentare le nozioni di base sulle reti informatiche, le tecnologie di rete più diffuse, i protocolli Internet e lo sviluppo di applicazioni distribuite *client-server* e *peer-to-peer* (**P2P**).

In particolare il programma del corso comprende:

- Sviluppo di **applicazioni** in rete:
  - Client-server;
  - Peer-to-peer.
- Reti a **connessione diretta**:
  - Collegamenti punto-punto;
  - Reti locali.
- Reti a **commutazione di pacchetto**;
- **Interconnessione** di reti di tipo diverso;
- **Trasporto** end-to-end e protocolli;
- **Sicurezza**;
- Reti **wireless** e **mobili**, intese come caso particolare delle normali reti **cablate** (*wired*).

1.1.1 Applicazioni in rete

Nel dettaglio delle *applicazioni in rete*, vedremo come già detto i paradigmi *client-server* e *peer-to-peer*, di cui possiamo già fare alcuni esempi:

- Applicazioni client-server:
  - Web;
  - Trasferimento file;
  - Posta elettronica;
  - DNS;
  - Ecc...
- Applicazioni peer-to-peer:
  - Ricerca di contenuti;
  - Torrent;
  - Telefonia online;
  - Ecc...

In questo ci avvarremo del concetto di **socket** come primitiva per la gestione della rete dal lato S/O.

1.1.2 Reti dirette, a commutazione e wireless

Inizieremo con lo studio di *collegamenti punto-punto*, e quindi di trasferimento affidabile di dati fra 2 punti. Vedremo poi le reti locali, ad accesso multiplo, e i casi particolari come *Ethernet*.

Vedremo quindi le reti a *commutazione di pacchetto* per la copertura di grandi regioni. Anche qui approfondiremo tecnologie come gli *switch*, ancora *Ethernet*, ecc...

Per quanto riguarda l'*interconnessione di reti* vedremo il protocollo Internet **IPv4**, il **routing** (cioè l'*instradamento*) e i protocolli di trasporto (**UDP** e **TCP**).

Parleremo anche di reti *wireless* e *mobili*, e quindi di tecnologie come **WiFi**, le **reti cellulari**, e reti senza infrastruttura come **Bluetooth**.

1.1.3 Sicurezza

Vedremo poi le minacce alla *sicurezza* e alcune soluzioni che abbiamo a disposizione per mitigarle. In particolare, tratteremo di **crittografia** e **integrità** dei messaggi.

Nello specifico parleremo di tecnologie a livello applicazione (**PGP**), a livello trasporto (**TLS** (usata in *HTTPS*)), a livello Internet (**IP-Sec**) e difese di sicurezza come **firewall** e **IDS**.

1.2 Terminologia

Iniziamo quindi a definire la terminologia di base usata nel corso, usando Internet come esempio.

1.2.1 Internet

La prima domanda che ci poniamo è *"Che cos'è Internet?"*.

Visione ingegneristica

Iniziamo col vedere la definizione di Internet agli occhi di un ingegnere che si occupa di reti:

- Si tratterà di una rete che connette miliardi di *dispositivi*, detti **host** (*"ospiti"*), che eseguono *applicazioni in rete* al cosiddetto **edge** (*"bordo"*) della rete.
- Una visione **interna** della rete ci dirà invece che è un insieme di **pacchetti** che viaggiano attraverso infrastruttura (*router*, *switch*), per raggiungere il loro destinatario.
- A livello **fisico** potremmo considerare le connessioni fisiche fra dispositivi, date da cavi, segnali radio, ecc...
- Infine, potremo organizzare le **reti** come collezioni di dispositivi, router e connessioni gestite da determinate organizzazioni.

Non è esattamente corretto parlare di *"reti di calcolatori"* in quanto oggi ad essere connessi a Internet sono tutta una gamma di dispositivi non necessariamente orientati al puro *calcolo*: è questo il caso del cosiddetto *Internet of Things* (**IoT**).

Possiamo quindi intendere Internet come una "rete di reti", cioè più **ISP** (*Internet Service Providers*) connessi fra di loro, che a loro volta connettono una gamma dispositivi (host, router, switch, ecc...).

Per governare l'operazione di tali rete si necessita di **protocolli**, che definiscono il modo in cui si inviano e ricevono messaggi in rete.

In particolare, per quanto riguarda Internet notiamo l'**IETF** (*Internet Engineering Task Force*), organizzazione che gestisce diversi standard del settore (anche detti **RFC**, da *Request For Comments memoranda*).

Visione utente

Per l'utente, internet sarà un insieme di **infrastrutture** che forniscono **servizi** finali, fra cui il Web, telecomunicazioni, streaming, ecc... Dal punto di vista delle **applicazioni** in esecuzione sui dispositivi, Internet rappresenterà un'interfaccia di programmazione per consentire la comunicazione fra processi su una o più macchine. In questo parleremo di **hook** che permettono alle applicazioni di **connettersi** a Internet, cioè accedere ad un qualche protocollo di trasporto dei dati.

1.2.2 Protocolli

Un **protocollo** è una precisa *specifica* del formato secondo il quale due dispositivi in rete si scambiano informazioni. Solitamente i protocolli si sviluppano in più fasi, successive nel tempo, dove si portano avanti diverse operazioni necessarie alla comunicazione.

Esempi di protocollo sono il *protocollo Internet* **IPv4**, e il *protocollo di trasporto* **TCP** usato nel *Web* e visto nel corso di progettazione web.

1.2.3 Infrastruttura di Internet

Vediamo più nel dettaglio la struttura di Internet:

- Abbiamo detto che all'*edge* di internet ci sono i cosiddetti *host*, cioè i **client** e i **server**. Notiamo che non vogliamo riferirci alle macchine fisiche client o server, ma ai **processi** che si comportano come tali per l'implementazione di un'applicazione distribuita.

- I dispositivi *terminali* che abbiamo appena nominato accedono ad Internet attraverso le cosiddette **reti di accesso**, cablate o wireless e basate sulle tecnologie utilizzate (router).

  Per collegare i sistemi terminali ai router si usano *reti residenziali*, *reti di accesso istituzionali* (scuola, lavoro, ecc...), nonché *reti wireless e mobili* (Wifi, o reti come 4G solitamente fornite da privati). In questo caso può interessarci la frequenza di trasmissione, in bit al secondo, di una rete di accesso, o se quella rete è ad accesso *condiviso* (pensa WiFi) o *dedicato* (pensa Ethernet).

  Uno standard storico per le reti di accesso è quello della trasmissione sulla linea telefonica su **DSL** (*Digital Subscriber Line*). Negli Stati Uniti si è invece diffuso l'uso dela linea televisiva cablata. Oggi, sfruttiamo invece tecnologie come **ADSL** (*Asymmetric Digital Subscriber Line*) e **FTTC** (*Fiber To The Cabinet*). La differenza principale fra queste è che la linea in ADSL è interamente in rame, sia dalla centrale all'armadio di ripartizione che dall'armadio ripartilinea agli utenti finali, mentre nella linea FTTC si porta il segnale all'armadio attraverso cavi in fibra ottica. Lo standard di ultima generazione è **FTTH** (*Fiber To The Home*), che prevede una linea in fibra ottica anche dall'armadio agli utenti finali.

  Possiamo quindi vedere la rete locale (**LAN** (*Local Area Network*)) di una comune abitazione come composta da un router, connesso a un **modem** DSL (*modem* deriva da modulatore/demodulatore sulla linea telefonica dei messaggi Internet) o direttamente via cavo ad un altro centro di ripartizione, e ad eventuali dispositivi come

access point WiFi che offrono la connessione via rete mobile ai dispositivi finali (una cosiddetta **WLAN**, *Wireless Local Area Network*).

Altre soluzioni per le comunicazioni wireless sono rappresentati da reti **cellulari** su larga scala, che sono quelle usate dagli operatori telefonici (tecnologie come **4G**, ecc...).

- Dalle reti di accesso si arriva a Internet attraverso reti interconnesse di router, arrivando quindi alle *reti di reti* di cui stavamo parlando.

2 Lezione del 24-09-25

2.1 Comunicazione dati su Internet

Abbiamo visto la struttura a livello fisico della rete Internet. Vediamo adesso i meccanismi secondo cui la trasmissione di dati avviene. La rete Internet è una rete a **commutazione di pacchetto**, il compito degli host è di:

- Ottenere messaggi dalle applicazioni;
- Dividere quei messaggi in frammenti più piccoli, detti **pacchetti**, di dimensione L bit;
- Trasmettere quei pacchetti nella rete di accesso ad una *frequenza di trasmissione* (o **bit-rate**) R.

Il tempo T_{packet} necessario a trasmettere un pacchetto da L bit su una linea da R bit al secondo di bitrate sarà quindi semplicemente calcolato come:

$$T_{\text{packet}} = \frac{L}{R}$$

Il bitrate, detto anche frequenza di *link*, dipende appunto dal **link** (o *mezzo*) della trasmissione, cioè l'infrastruttura fisica che sta fra trasmettitore e ricevitore. Possiamo classificare 2 tipi di link:

- Mezzi **guidati**: segnali che si propagano in mezzi solidi (rame, fibra, cavi coassiali, ecc...).
  - Un celebre esempio di mezzo trasmissivo guidato è il classico **doppino telefonico**, o trasmettitore a *Twisted Pair* (**TP**). Questo è formato da due fili di rame isolati e avvolti l'uno sull'altro, che permettono la trasmissione differenziale e quindi la riduzione dei rumori in *common-mode*;
  - Un altro tipo di mezzo trasmissivo guidato è il **cavo coassiale**, formato da due conduttori concentrici in rame separati da un dielettrico. Il segnale è trasferito come campo magnetico fra i due conduttori: questo permette bitrate più alti rispetto al normale doppino telefonico e una migliore schermatura dalle interferenze;
  - Infine, possiamo parlare della **fibra ottica**, formata da fibra di vetro che porta impulsi luminosi ad altissima velocità. Questa tecnologia presenta velocità di trasmissione estremamente alte, e vista la natura luminosa del segnale, non è suscettibile ad interferenze (alta affidabilità, cioè piccola frequenza di errore sui bit).

- Mezzi **non guidati**: segnali che si propagano nell'etere (segnali radio, ecc...). Questi sono solitamente meno sicuri ma significativamente più comodi per l'utente finale (possibilità di spostarsi, mancanza di cavi, ecc...).

  La trasmissione wireless è suscettibile a fenomeni fisici come *riflessi*, *interferenze* e *ostruzione* da parte di oggetti fisici.

  - Il **WiFi** è un esempio di mezzo di trasmissione non guidato che può raggiungere centinaia di Mbps su regioni locali;
  - Reti wireless più ampie possono essere quelle **cellulari**, usate nella telefonia mobile;
  - Infine si può parlare delle **reti satellitari**, usate per l'interconnessione di regioni geografiche fra di loro anche molto distanti.

2.1.1 Commutazione di circuito

Prima della commutazione di pacchetto si usava la tecnica della **commutazione di circuito** (ad esempio sulle linee telefoniche). Questo prevede di dedicare completamente una certa linea di trasmissione alla comunicazione fra due host, invalidandone quindi l'uso da parte di altri host.

Chiaramente, la commutazione di pacchetto permette un carico migliore della linea, dove più pacchetti provenienti da diverse fonti possono viaggiare a istanti temporali molto vicini fra di loro.

In particolare, la commutazione di paccheto è utile per dati trasmessi in *burst*, mentre la rete a commutazione di circuito assicura minima congestione possibile a costo di occupazione completa della linea.

2.1.2 Commutazione di pacchetto

La tecnica della **commutazione di pacchetto** o *packet-switching* permette ad una rete di **router** interconnessi di ricevere ed instradare (letteralmente, *"routing"*) pacchetti provenienti da più fonti in modo che raggiungano la loro destinazione.

Questo inserisce chiaramente un ritardo nel sistema, in quanto il router deve:

- Ricevere il pacchetto *completamente* ed memorizzarlo: questo richiede L/R secondi;
- Leggere l'header del pacchetto per capire il prossimo passo dell'instradamento;
- Trasmettere il pacchetto verso la sua nuova destinazione (un altro router o l'host finale), impeigando ancora L/R secondi.

Abbiamo quindi che il ritardo end-end immesso dal router è necessariamente di almeno $2L/R$ secondi, tralasciando il tempo necessario all'instradamento stesso.

Nel caso generale si abbiano N router (quindi $N+1$ link fra i router) e P pacchetti da inviare, dovremo considerareche il primo pacchetto arriva in $(N+1)\frac{L}{R}$ (deve attraversare tutti i link) e i successivi $P-1$ pacchetti arrivano in $(P-1)\frac{L}{R}$, per cui il tempo complessivo è:

$$T_{end-to-end} = (N+P)\frac{L}{R}$$

Se troppi pacchetti arrivano in un breve lasso di tempo, cioè se la frequenza di arrivo supera quella di trasmissione:

- I pacchetti verrano messi in coda finché non sarà possibile trasmetterli;
- I pacchetti possono essere persi se il buffer di memoria dedicato alla loro memorizzazione nel ruoter si riempie.

2.2 Prestazioni della commutazione di pacchetto

Facciamo qualche considerazione ulteriore sulle prestazioni delle linee a commutazione di pacchetto. Abbiamo ottenuto il valore $T_{\text{packet}} = \frac{L}{R}$ per la trasmissione di un singolo pacchetto da L bit su una linea con bitrate R, e $T_{\text{end-to-end}} = (N + P)\frac{L}{R}$ per più pacchetti su un numero arbitrario di router.

Da quanto abbiamo detto nella scorsa sezione, un modello più sofisticato del packet-switching terrà conto di 4 sorgenti di ritardo:

- Ritardo di **trasmissione** T_{trans}, dato dalle caratteristiche del link. Come abbiamo già detto, questo vale:
$$T_{\text{trans}} = \frac{L}{R}$$
con L lunghezza del pacchetto in bit e R bitrate del link;
- Ritardo di **propagazione** T_{prop}, dato dalle proprietà fisiche del mezzo di trasmissione. In particolare, questo è il tempo fisico di trasmissione del segnale su un link, dato da:
$$T_{\text{prop}} = \frac{d}{s}$$
con d distanza del link e s velocità del mezzo di trasmissione. Chiaramente, per i nostri scopi s sara una frazione significativa della velocità della luce $c \approx 3 \cdot 10^8$ m/s;
- Ritardo di **laborazione** (instradamento) T_{proc}, dipende dalle caratteristiche del router ed è perlopiù costante;
- Ritardo di **accodamento** dato dalla presenza di code T_{queue}. Questo è il più complicato da trattare, in quanto dipende dal numero di pacchetti presenti nel buffer del router. Come vedremo fra poco, una buona euristica per la valutazione di questo ritardo (che è comunque trattabile solo in maniera statistica) è l'*intensità di traffico* sulla linea di trasmissione.

Sommando queste sorgenti di ritardo potremo ottenere una stima del ritardo complessivo su un router (nodo) T_{node}:

$$T_{\text{node}} = T_{\text{trans}} + T_{\text{prop}} + T_{\text{proc}} + T_{\text{queue}}$$

Dati N nodi e P pacchetti, il ritardo end-to-end potrà quindi essere calcolato semplicemente come:

$$T_{\text{end-to-end}} = (N + P)T_{\text{node}}$$

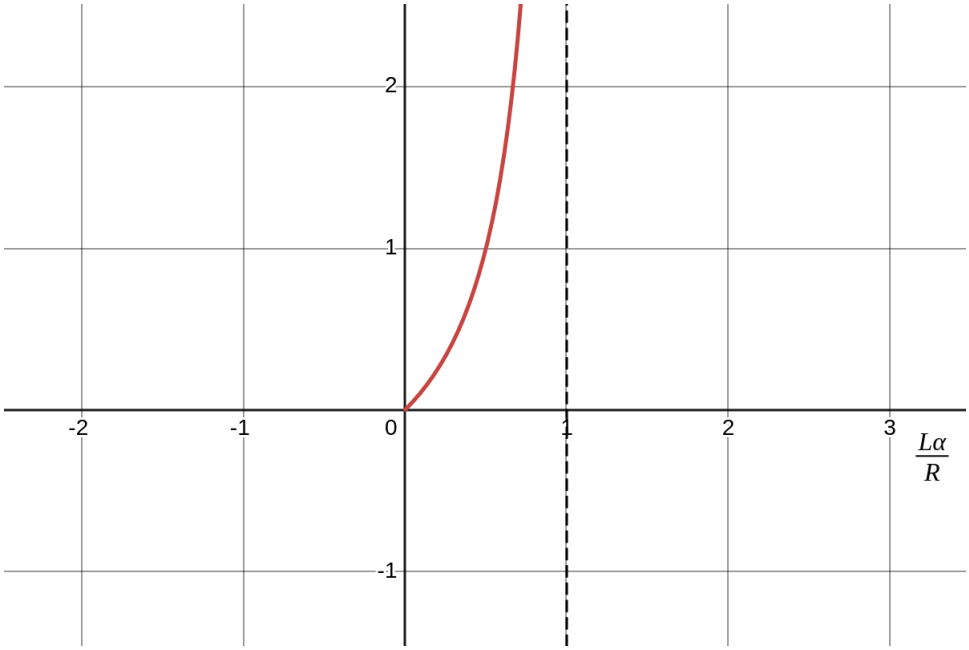
2.2.1 Ritardo di accodamento

Come anticipato, possiamo valutare statisticamente il ritardo di accodamento T_{queue} calcolando l'*intensità di traffico* su una linea:

$$I_{\text{traffic}} = \frac{L\alpha}{R}$$

presa α come la frequenza media di trasmissione di pacchetti sull'istante temporale.

Semplicemente tracciando la funzione si nota che se $I_{\text{traffic}} >= 1$ il tempo di di accodamento tende a infinito (si ha necessariamente perdita di pacchetti), per cui vorremo mantenere $I_{\text{traffic}} < 1$, e idealmente $I_{\text{traffic}} << 1$.



2.2.2 Traceroute

Per fare diagnostica in situazioni reali, si possono usare software di **traceroute**, che inviano pacchetti con lo scopo di tracciare gli host incontrati e misurare il tempo impiegato nella trasmissione *round-trip*, o almeno capire se i pacchetti sono stati inviati o persi.

Più nello specifico, ipotizziamo di avere N router fra sorgente e destinazione. Traceroute invierà $N + 1$ pacchetti speciali sulla linea fra sorgente e destinazione, e ogni router reinvierà il primo di questo che riceve (l'n-esimo) indietro alla sorgente, assieme a informazioni temporali e il suo indirizzo. In questo modo, l'$N + 1$ pacchetto arriverà effettivamente alla destinazione, mentre gli N pacchetti precedenti scandaglieranno ognuno dei router sulla via per la destinazione.

In verità, questo procedimento viene ripetuto 3 volte, per avere risultati più accurati.

Vediamo ad esempio il traceroute che dal portatile su cui sto scrivendo gli appunti arriva ai server dell'Università di Pisa:

```
$ traceroute www.unipi.it
traceroute to www.unipi.it (131.114.104.6), 30 hops max, 60 byte packets
 1  _gateway (10.178.37.4)  3.745 ms  3.552 ms  3.494 ms
 2  172.16.133.129 (172.16.133.129)  300.183 ms  300.156 ms  300.133 ms
 3  172.16.133.38 (172.16.133.38)  60.081 ms  59.673 ms  60.235 ms
 4  172.16.17.50 (172.16.17.50)  52.910 ms  60.413 ms  59.354 ms
 5  151.7.56.70 (151.7.56.70)  61.571 ms  62.259 ms  60.645 ms
 6  151.7.56.66 (151.7.56.66)  89.876 ms  55.029 ms  86.501 ms
 7  151.6.3.199 (151.6.3.199)  52.064 ms 151.6.1.27 (151.6.1.27)  51.777
   ms 151.6.3.199 (151.6.3.199)  44.325 ms
 8  151.6.5.139 (151.6.5.139)  52.076 ms 151.6.1.213 (151.6.1.213)  52.842
    ms 151.6.5.139 (151.6.5.139)  52.399 ms
 9  garr.rom.namex.it (193.201.28.15)  52.662 ms  51.836 ms  52.516 ms
```

```
10  re1-rm02-rs1-rm02.rm02.garr.net (185.191.181.70)  52.317 ms  52.470 ms
      52.278 ms
11  rs1-rm02-rs1-pi01.pi01.garr.net (185.191.181.34)  56.893 ms  56.872 ms
      56.526 ms
12  rs1-pi01-rl1-pi01.pi01.garr.net (185.191.181.39)  56.495 ms  49.338 ms
      50.169 ms
13  rl1-pi01-ru-unipi.pi01.garr.net (193.206.136.90)  49.919 ms  49.808 ms
      50.003 ms
14  jser-jspg.unipi.it (131.114.191.90)  51.377 ms  50.215 ms  51.243 ms
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *
```

I numeri a sinistra indicano l'indice del router raggiunto, seguiti dall'indirizzo e le misurazioni temporali nei 3 tentativi effettuati dal traceroute. In praticolare, notiamo come i router da 2 a 8 sono anonimi, al 9 siamo arrivati nella rete del *GARR*, e dal router 15 in poi non abbiamo più ottenuto risposte ai pacchetti traceroute (probabilmente per via della configurazione dei router, il traffico passa comunque).

2.2.3 Troughput

Introduciamo il concetto di **troughput** come la frequenza (bit al secondo) con cui i bit vengono spediti da trasmettitore a ricevitore. Il throughput può essere:

- **Istantaneo**: calcolato ad un certo istante temporale (per quanto possibile dalla natura discreta della trasmissione);
- **Medio**: calcolato su un periodo temporale più lungo.

Ipotizziamo che un server debba inviare un file di F bit ad un client. La linea di comunicazione fra server e router ha capacità di link di R_S bit/secondo, mentre la linea fra router e client ha capacità di R_C bit/secondo. Chiaramente la capacità totale sarà:

$$R_F = \min(R_S, R_C)$$

cioè link con basse capacità di trasmissione rappresentano *bottleneck* per il throughput di tutto il collegamento fra dispositivi.

Facciamo adesso l'esempio di una rete condivisa fra più coppie client/server. In questo caso, date N coppie, la capacità della rete di interconnessione agli occhi di una singola coppia sarà, preso R come la capacità complessiva:

$$R_i = \frac{R}{N}$$

per cui ogni coppia vedrà una linea di comunicazione con capacità:

$$R_F = \min(R_S, R_C, \frac{R}{N})$$

2.3 Struttura di Internet

Abbiamo detto che Internet è una rete di reti.

- Gli host si collegano a internet attraverso le *reti di accesso* (reti LAN, istituzionali, reti mobili, ecc...);
- Le reti di accesso si collegano agli **ISP** (*Internet Server Provider*);
- Gli ISP devono essere collegati ad altri ISP per permettere la comunicazione fra host su diversi ISP (all'interno di **IXP** (*Internet eXchange Point*, ecc...)).

La struttura di reti interconnesse che si viene a formare è molto complessa, e la sua evoluzione è stata guidata da fattori economici e politici.

Abbiamo quindi una struttura gerarchica:

1. Al livello più alto troviamo i cosiddetti **tier 1 ISP** e i **content provider** (Google, Facebook, ecc...) che si occupano di copertura nazionale e internazionale. In particolare, i content provider preferiscono collegarsi direttamente agli IXP per risparmiare sugli ISP;
2. Seguono gli **IXP**, che collegano più ISP fra di loro, e gli ISP locali (regionali, ecc...);
3. Infine troviamo le reti di **accesso** locale (reti LAN, WLAN, ecc...).

2.4 Sicurezza Internet

La struttura di Internet espone i suoi utenti a diversi pericoli, fra cui:

- **Virus**: programmi maligni che si replicano modificando altri programmi;
- **Worm**: programmi maligni che si replicano con lo scopo di diffondersi in altri computer;
- **Spyware**: programmi maligni che cercano di ottenere informazioni che violano la privacy;

ed altre svariate categorie di *malware*.

2.4.1 Attacchi DoS

Un caso tipico di attacchi informatici in rete è quello degli attacchi **DoS** (*Denial of Service*), dove l'attaccante cerca di sfruttare le risorse di un server (memoria, larghezza di banda) al punto di renderlo inutilizzabile ad altri utente. Questo può essere fatto inviando traffico maligno al server (pacchetti molto grandi o corrotti).

La tecnica può essere espansa a più attaccanti (*botnet*) dando vita a tecniche più sofisticate (come il **DDoS**, *Distributed Denial of Service*).

2.4.2 Intercezione di pacchetti

Una problematica delle reti a commutazione di pacchetto è l'**intercezione** dei pacchetti instradati, o in inglese *packet sniffing*. Questo è più facile su reti wireless piuttosto che cablate, ed espone pericoli ovvi per la sicurezza (password, dati sensibili, ecc...).

Oggi, come vedremo, si usano protocolli che implementano forme di crittografia per mitigare questo tipo di problematiche.

2.4.3 Spoofing IP

Il problema di identità fasulle si presenta in Internet attraverso pacchetti malformati, con indirizzi sorgente sbagliati. Queste tecniche, seppure meno pericolose, possono comunque essere usate per confondere o comunque complicare il traffico sulle reti.

3 Lezione del 26-09-25

3.1 Livelli di protocolli

Analizzando l'architettura di Internet abbiamo introdotto il concetto di **protocollo**. Studiando il *core* della rete abbiamo poi visto che la rete è molto complicata e costruita su più *astrazioni*: host inviano pacchetti di applicazioni attraverso router su vari link, ecc...

Possiamo decidere di organizzare i protocolli che governano tali operazioni attraverso una struttura a *layer* o **livelli**. Ogni livello implementa un dato **servizio**, utilizzando le sue azioni interne, e basandosi sui servizi offerti dal livello sottostante.

Questo procedimento non è fuori luogo in sistemi complessi composti da diversi componenti in relazione fra di loro: permette infatti di gestire ogni componente singolarmente, senza doverci preoccupare di rompere la compatibilità con gli altri componenti (basta basarsi sullo stesso servizio sottostante e offrire lo stesso servizio al livello superiore).

Lo stack di protocolli internet che studiamo è quindi il seguente, a 5 livelli:

1. **Application**: il livello che supporta le applicazioni in rete. Protocolli *IMAP*, *SMTP*, *HTTP*;
2. **Transport**: il livello che supporta il trasferimento dati da processo a processo assicurando sicurezza e astrazione sul modello a pacchetto. Protocolli *TCP* e *UDP*;
3. **Network**: il livello che implementa il trasferimento di *datagrammi* da sorgenti a destinazioni. Protocolli *IP*, *routing*;
4. **Link**: il livello che implementa la trasmissione di dati tra elementi di rete fra di loro "vicini". Protocolli *Ethernet*, *802.11 (WiFi)*, *PPP*;
5. **Physical**: il livello fisico rappresentato dai bit sul mezzo di comunicazione.

Un diverso stack è quello presentato dal modello di riferimento **ISO/OSI**. Questo è diverso dal modello presentato prima in quanto presenta qualche livello in più:

1. **Application**: come sopra;
2. **Presentation**: il livello che permette alle applicazioni di interpetare il significato dei dati. Protocolli di *compressione*, *crittografia*, ecc...;

3. **Session**: il livello che permette *sincronizzazione*, *checkpoint*, *recupero* di dati, ecc...;
4. **Transport**: come sopra;
5. **Network**: come sopra;
6. **Link**: come sopra;
7. **Physical**: come sopra;

Queste funzioni, se strettamente necessarie, dovranno essere implementate nel livello application.

3.1.1 Incapsulamento di protocolli

Adesso che abbiamo stabilito una gerarchia di protocolli che permettono il collegamento internet, possiamo definire in maniera più precisa cosa accade quando ci colleghiamo, ad esempio, ad un werver HTTP per richiedere una pagina Web.

Dal nostro calcolatore, e in particolare dall'applicazione *browser* in esecuzione sul nostro calcolatore, attraversiamo tutti i livelli (application, transport, network, link, e physical) per arrivare ad un router (probabilmente quello della rete di accesso). Man di mano che scendiamo in livelli più bassi corrediamo il messaggio inviato dall'applicazione con altre informazioni di controllo, utili ai livelli più bassi. A questo punto il router ottiene il messaggio attraverso, magari solo il livello link e physical: questo procedimento si ripete, finché non raggiungiamo il server. Quando il messaggio raggiunge la macchina server, risaliamo tutti i livelli (physical, link, network, transport, application) per arrivare all'applicazione server vera e propria, perdendo nel frattempo le informazioni aggiunte dai livelli sottostanti in fase di trasmissione. A questo punto il server può interpretare il messaggio ed eventualmente rispondere.

Con questa sezione abbiamo concluso l'introduzione al funzionamento (ad alto livello) di Internet, visto sia dall'*edge* della rete (cioè dal punto di vista degli *host*) che dal *core* della rete (cioè dal punto di vista dell'*infrastruttura*) di rete.

3.2 Applicazioni in rete

Veniamo quindi allo sviluppo di **applicazioni** in rete, con l'obiettivo di tornare alle specifiche delle reti in un secondo momento.

In questa sezione vedremo i principi delle applicazioni Web *client-server* e *peer-to-peer* (**P2P**), in particolare approfondendo i protocolli Web (HTTP), e-mail (SMTP, IMAP), il sistema DNS, nonché la programmazione con l'API dei *socket* coi protocolli TCP e UDP.

Nostro focus sarà quindi il livello applicazione (e in minor parte di trasporto), e gli aspetti concettuali e di implementazione di applicazioni in rete.

Creare un'applicazione in rete significa scrivere programmi che:

- Girano su diversi sistemi;
- Comunicano via la rete.

Le applicazioni vengono scritte per gli **host**: i router non eseguono applicazioni utente, e anzi *non eseguono* nemmeno lo stack protocollare completo (si limitano al livello link e al massimo network). Sviluppare applicazioni per i sistemi all'*edge* della rete permette invece la facile e rapida propagazione delle stesse.

3.2.1 Paradigma client-server

Il paradigma **client-server** è il più comune per le applicazioni in rete. In questo caso individuiamo due agenti principali:

- Il **server** è un host sempre attivo, con indirizzo IP permanente, solitamente distribuito su data center, per permettere scalabilità (qui si sfruttano tecnologie come *load balancer*, ecc...);
- Il **client** è un host che stabilisce contatto intermittente col sever, che può avere indirizzo IP variabile, e che non interagisce mai direttamente con altri client.

Abbiamo quindi che l'identità di client e server è *forte*, la relazione fra i due è asimmetrica e ben definita. In termini di risorse, il client non dovrà avere particolari risorse computazionali, mentre il server dovrà essere capace di gestire le richieste di tutti i client.

Esempi di protocolli client-server sono il protocollo HTTP, i protocolli di posta IMAP e il protocollo di trasferimento file FTP.

3.2.2 Paradigma peer-to-peer

Nel paradigma **peer-to-peer** non esiste un singolo server always-on, ma ogni peer può comportarsi in modalità intermittente sia da client che da server. Questo significa che i peer ricevono servizi da altri peer, fornendo in cambio altri servizi.

Questa caratteristica permettte l'*auto-scalabilità*: la rete P2P si sviluppa autonomamente man di mano che si aggiungono peer.

I peer hanno un identità molto più labile rispetto a quelle di client e server tradizionali: l'indirizzo IP può essere dinamico, non sono sempre online, il loro servizio potrebbe essere intermittente, ecc...

3.2.3 Socket

I **socket** sono l'API che implementa la connettività di rete per le applicazioni che scriveremo. Sono offerti dal sistema operativo e rappresentano un'astrazione per la connessione di rete (come i file rappresentano un'astrazione per il disco).

L'analogia tipica del socket è quella di una *porta*. I messaggi entrano dalla porta ed escono dalla porta: quello che sta dietro alla porta è parte dell'infrastruttura implementata prima dal sistema operativo (che implementa i livelli protocollari) e poi dalla rete Internet in sé per sé, ed è astratto via dal programmatore.

4 Lezione del 29-09-25

Continuiamo a vedere nel dettaglio l'ambito delo *sviluppo di applicazioni*.

4.1 Comunicazione fra processi

Abbiamo visto come, sebbene debbano girare su una vasta gamma di dispositivi ed interagire con svariate tecnologie di collegamento, queste possono essere riassunte nei paradigmi client-server e *peer-to-peer*. Inoltre, abbiamo visto come maggior parte della complessità dell'infrastruttura di rete sia astratta via nei moderni S/O dietro il meccanismo dei *socket*.

In particolare, vediamo i nostri applicativi come composti da **processi** in esecuzione su macchine fisiche, da cui **processi client** su macchine client e **processi server** su macchine server. Le modalità secondo la quale questi processi si scambieranno **messaggi** saranno rappresentate dalla cosiddetta **IPC** (*Inter Process Communication*).

Notiamo che non è necessario che la IPC sia su macchine diverse, due processi nella stessa macchina possono infatti comunicare fra di loro come se si trovassero su macchine diverse collegate in rete.

Quello che vanno ad implementare i **socket**, standard *de facto* apparso in origine nei sistemi operativi BSD, è appunto l'IPC fra più processi.

4.1.1 Modalità di indirizzamento

Per ricevere messaggi, i processi devono avere degli *identificatori*. Questi sono rappresentati da **indirizzi IP** su 32 bit. L'indirizzo IP della macchina su cui i processi girano non basta, in quanto chiaramente una macchina può avere più di un processo in esecuzione.

4.1.2 Protocolli livello application

Un protocollo di livello *application* dovrà a questo punto definire diverse specifiche sull'IPC:

- Il **tipo** di messaggi scambiati: questi possono solitamente essere *richieste* e *risposte*;
- **Sintassi** dei messaggi: quali campi presentano i messaggi e come i campi sono delineati nella struttura del messaggio;
- **Semantica** dei messaggi: il significato dell'informazione contenuta nei campi;
- **Regole**: su come e quando i processi possono ricevere o inviare messaggi.

Esistono diversi protocolli application, sia **aperti** (interoperabili per applicazioni di più sviluppatori, lo sono il protcollo HTTP, i protocolli di posta, ecc...) che **proprietari** (lo sono ad esempio i protocolli associati a prodotti software proprietari come Skype, ecc...). In particolare, i protocolli aperti possono essere sia **ufficiali** (supportati da agenzie sotto forma di RFC), o **non ufficiali** (basati su documenti non ufficiali ma di pubblico accesso, diventati standard *de facto* dopo grande adozione, ad esempio BitTorrent).

4.1.3 Servizi ad applicazioni

Iniziamo a vedere quali tipi di servizi potrebbero servire ad un applicazione in rete.

- **Integrità dati**: alcune applicazioni (trasferimento dati, transazioni, ecc...) potrebbero richiedere un trasferimento sicuro al 100%. Altre (videogiochi, streaming, ecc...) potrebbero invece poter tollerare perdite paraziale dei dati in fase di trasmissione.
- **Throughput**: alcune applicazioni (multimedia, ecc...) richiedono una quantità minima di throughput per essere efficaci. Altre (le cosiddette applicazioni *elastiche*) ne prendono quanto non hanno a disposizione.
- **Tempo**: alcune applicazioni (ancora videogiochi, telefonia online, ecc...) potrebbero richiedere delay temporali molto contenuti per essere efficaci.
- **Sicurezza**: servizi più o meno critici richiedono livelli variabili di sicurezza (crittografia, ecc...).

Su questa base, potremmo classificare alcune applicazioni sulla base dei requisiti di trasporto che hanno:

Applicazione	Perdita dati	Throughput	Tempo	Sicurezza
Trasferimento file/download	Nessuna	Elastico	Indifferente	Dipende
E-mail	Nessuna	Elastico	Indifferente	Dipende
Web	Nessuna	Elastico	Indifferente	Dipende
Streaming	Tollerante	5 Kbps - 5 Mbps	Pochi secondi	Non importante
Videogiochi	Tollerante	5 Kbps - 5 Mbps	Pochi millisecondi	Dipende
Messaggistica istantanea	Nessuna	Elastico	Perlopiù indifferente	Critica

4.1.4 Protocolli livello transport

Potrebbe essere utile, per capire come realizzare le specifiche sopra descritte, studiare ad alto livello i due protocolli di trasporto principali:

- **TCP** (*Transmission Control Protocol*): assicura il trasporto *affidabile* di messaggi tra processi, controllo del *flusso* e delle *congestioni*, ma ha promesse più scarse nel campo della temporizzazione e del throughput (sebbene assicuri un *throughput minimo*). Orientato alla *connessione* fra processi client e server, è più sicuro dell'alternativa;
- **UDP** (*User Datagram Protocol*): meno sicuro e affidabile, non fornisce servizi di controllo flussi o congestioni, né throughput minimo. Questo lo rende adatto per soluzioni a basso overhead, dove le prestazioni sono più significative della correttezza dei dati.

Il controllo del flusso e delle congestioni si collega a quanto visto nella sezione 2.2:

- Il controllo del **flusso** si assicura che il mittente non potrà sopraffare il destinatario con una mole troppo grande di messaggi;
- Il controllo delle **congestioni** assicura che i ruoter nell'infrastruttura fra mittente e destinatario non vengano sovraccaricati.

Vediamo quindi una tabella riassuntiva delle differenze fra TCP e UDP:

	TCP	UDP
Integrità	Assicurata	Non assicurata
Velocità	Minima assicurata	Massima
Controllo flusso	Si	No
Controllo congestioni	Si	No
Paradigma	Connessioni	Senza connessioni

Possiamo quindi assegnare ad ognuna delle applicazioni viste nella tabella di sezione 4.13 un protocollo adatto:

Applicazione	Protocollo application	Protocollo transport
Trasferimento file/download	FTP	TCP
E-mail	SMTP	TCP
Web	HTTP	TCP
Streaming	HTTP, DASH	TCP
Videogiochi	WOW, FPS o proprietario	TCP o UDP
Messaggistica istantanea	HTTP o proprietario	TCP o UDP (telefonia)

4.1.5 Sicurezza in TCP

I socket TCP e UDP di base non forniscono particolari funzionalità di sicurezza: i dati sono trasmessi senza crittografia, per cui dati sensibili sono visibili in chiaro.

Si può sfruttare il protocollo (in verità protocollo *middleware*, che sta fra livello transport e livello application) **TLS** (*Transport Layer Security*) per fornire connessioni TCP crittografate, con integrità dei dati assicurata e autenticazione del destinatario.

Possiamo sfruttare TLS in 2 modi principali:

- TSL implementato in applicazione, che a sua volta interagisce con socket TCP;
- API che fornisce socket TLS, che ricevono dati in chiaro e li inviano su Internet crittografati.

La manifestazione più comune del protocollo TLS si vede negli URL delle pagine web, che iniziano con `http://` quando si usa HTTP su TCP puro, e con `https://` quando si usa HTTP su TCP con TLS.

4.2 Web e HTTP

Veniamo quindi a dettagliare la più famosa applicazione sviluppata su Internet, cioè il Web. Come abbiamo visto il Web è supportato dal protocollo **HTTP** (*HyperText Transfer Protocol*).

Ricordiamo quindi che una **pagina** Web consiste di oggetti di diversi formati che possono essere allocati su più Web server. Le pagine in sé per se sono anch'esse oggetti, consistono di file **HTML** (*Hypertext Markup Language*), indirizzabili assieme come tutti gli altri oggetti che le compongono da un **URL** (*Uniform Resource Locator*), in forma:

```
<schema>//<nome-host>/<percorso_risorsa>
```

ad esempio, `https://www.bittorrent.org/index.html`.

4.2.1 Protocollo HTTP

Il protocollo HTTP è basato sul modello client-server (richiede un protocollo di livello transport che supporti connessioni client-server, quasi sempre TCP). In questo, il client è rappresentato dal *browser*, che compila richieste per ottenere pagine web, mentre il server è rappresentato dal *Web server*, che riceve le richieste dei browser e risponde inviando oggetti.

Nello specifico, una connessione HTTP si svolge come segue:

- Il client inizia la connessione TCP (lato applicazione, crea il socket) col server, alla porta 80;

- Il server accetta la connessione TCP del client;
- Messaggi HTTP vengono scambiati fra client (browser) e server sulla linea TCP;
- La connessione TCP viene chiusa.

Il protocollo HTTP è *privo di stato*, cioè non si mantiene nessuna informazione riguardo alle richieste passate del client.

Esistono 2 tipi di connessioni HTTP:

- HTTP **non persistente**: si apre la connessione TCP e si invia al più un oggetto sulla connessione prima di chiudere. In questo caso scaricare più oggetti richiede più connessioni TCP;
- HTTP **persistente**: si apre la connessione TCP e si inviano più oggetti sulla connessione prima di chiudere.

Definiamo il **RTT** (*Round-Trip Time*) come il tempo che un piccolo pacchetto impiega per viaggiare da client a server e ritorno.

- Nel caso dell'HTTP non persistente, richiediamo un RTT per iniziare la connessione TCP, un RTT per richiedere il file, più il tempo necessario a trasferire il file vero e proprio, per cui si impiega:

  $$T_{\text{non-pers}} = 2\text{RTT} + T_{\text{tras}}$$

  per ottenere ogni file.
- Nel caso dell'HTTP persistente, dovremmo comunque usare 2 RTT per iniziare la connessione e chiedere il primo file, ma ogni file successivo richiederà solamente il tempo RTT necessario a richiedere la risorsa, per cui risparmieremo tempo.

  Chiaramente in questo caso avremo il problema di dover capire *quando* chiudere la trasmissione: magari dopo n richieste, dopo un tempo T (*Timeout*), ecc...

4.2.2 Richieste HTTP

Abbiamo visto come i messaggi HTTP appartengono a 2 tipi, *richieste* e *risposte*. I messaggi sono in formato ASCII, quindi leggibile dall'uomo.

In particolare, l'header di una richiesta HTTP ha la seguente forma generale:

```
GET /index.html HTTP/1.1
Host: www-net.cs.umass.edu
User-Agent: Firefox/3.6.10
Accept: text/html,application/xhtml+xml
Accept-Language: en-US,en
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Ogni riga è terminata da `\r\n`, caratteri di ritorno carrello e nuova linea, non riportati nell'esempio.

La prima riga definisce il **tipo** di richiesta (GET, POST, HEAD, ecc...), la **risorsa** richiesta e la **versione** del protocollo che il client vuole utilizzare. La seconda linea contiene poi l'host del server richiesto, e la terza il processo (qui il browser Firefox) che effettua la richiesta.

Le successive rige definiscono il tipo di oggetto che il client è disposto ad ottenere (tipo MIME, lingua, codifica e compressione, ecc...).

Infine, l'ultima riga stabilisce le regole di connessione richieste dal client, in questo caso `keep-alive` (quindi HTTP persistente).

Un doppio ritorno carrello e nuova linea segnala la fine delle linee di header e l'inizio dei dati veri e propri.

5 Lezione del 01-10-25

Continuiamo la discussione del protocollo HTTP.

5.0.1 Tipi di richiesta HTTP

Esistono più tipi di richiesta HTTP:

- **GET:** richiede una risorsa dal server;
- **POST:** invia informazioni al server, ad esempio per trasferire un form.
- **HEAD:** richiede solo l'intestazione o *header* della risorsa, ad esempio per controllare se ha già la versione più recente in cache;
- **PUT:** aggiorna o rimpiazza una risorsa ad un dato URL. Se non esiste, la crea;
- **DELETE:** Rimuove una risorsa a un dato URL;
- **CONNECT:** Stabilisce una connessione col server. Spesso è utilizzato per connessioni SSL (HTTPS);
- **TRACE:** Risponde con la stessa richiesta. Usata per motivi di debug, ad esempio dal programma `traceroute` visto in 2.2.2;
- **OPTIONS:** Descrive le opzioni di comunicazione per la risorsa interessata. Utile per trovare quali metodi HTTP sono supportati dal server.

Più informazioni sulle specifiche del protocollo HTTP per sviluppatori web possono poi essere trovate in `https://raw.githubusercontent.com/seggiani-luca/appunti-web/481107c15776fac537b7882b6e0becfcb88b9886/master/master.pdf`.

5.0.2 Risposte HTTP

Ad una richiesta HTTP su un server web alla porta 80 segue una **risposta**. Questa ha la seguente forma generale. Questa ha un header dalla seguente forma generale:

```
HTTP/1.1 200 OK
Date: Sun, 26 Sep 2010 20:09:20 GMT
Server: Apache/2.0.52 (CentOS)
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
Content-Length: 2652
Keep-Alive: timeout=10, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Anche queste righe sono separate da `\r\n`, caratteri di ritorno carrello e nuova linea.

La prima riga definisce la **versione** protocollo, e la risposta del server in codice (200) e testo (OK).

Esiste un sistema di codici che comunicano le varie situazioni che si possono creare alla risposta. Ad esempio, alcuni dei codici più tipici sono **200** (*OK*), **301** (*Moved Permanently*), **400** (*Bad Request*), **404** (*Not Found*) e **505** (*Version not supported*). Vediamo come i codici che iniziano da **100** riguardano *messaggi informativi*, **200** condizioni di *successo*, **300** di *redirezione*, **400** errori *client* e **500** errori *server*. Anche questo argomento viene approfondito all'URL riportato nella sezione precedente.

Seguono poi diverse coppie chiave-valore che contengono informazioni sul contenuto ottenuto, il server che l'ha fornito, ecc...

Possiamo rivolgere la nostra attenzione alle linee `Keep-Alive` e `Connection`: queste stabiliscono il tipo di connessione. Dall'esempio della scorsa lezione avevamo visto come una richiesta può esprimere la preferenza per connessioni `keep-alive` (in HTTP persistente). Qui il server ha consentito la connessione `keep-alive`, stabilendo un timeout di 10 secondi e al massimo 100 richieste HTTP accettate.

5.0.3 Meccanismo dei cookie

Avevamo detto che il protocollo HTTP è *stateless*, cioè privo di stato. Un modo per reintrodurre una qualche nozione di stato nel protocollo è adottare il meccanismo dei **cookie**.

Un cookie è una coppia chiave-valore, o semplicemente anche una sola chiave, che il server invia al client assieme ad una pagina per conservare informazione di stato. Il browser che ha intenzione di preservare lo stato potrà a questo punto annettere il cookie ad ogni richiesta successiva, permettendo al server di "ricordare" quale utente sta lanciando la richiesta.

Prevediamo quindi una nuova linea di stato nell'header delle risposte HTTP (`Set-Cookie:`) che dovrà contenere questi cookie, e una linea simile nell'header delle richieste HTTP (`Cookie:`) che vengono dal browser. A questo punto basterà mantenere un file (o volendo un database) sul browser utente che associa i cookie ad un dominio, e un database (qui obbligatoriamente, probabilmente anche massiccio) di *backend* sul Web server che associa un cookie ad ogni utente.

Con il meccanismo dei cookie il protocollo stateless dell'HTTP diventa effettivamente *stateful*, cioè capace di preservare lo stato fra più richieste (assunto che il browser sia disposto a reinviare ogni volta la stringa di cookie).

5.1 Cache Web

Per ridurre il tempo di accesso agli Web server (che possono trovarsi anche molto lontano dal browser) e ridurre il traffico sugli stessi, si può usare un sistema di *caching*, cioè redirezionare il browser verso una *cache Web* (o **server proxy**).

Un proxy duplica alcuni dei contenuti presenti sul server originale: se possiede la risorsa richiesta dall'utente, la restituisce, altrimenti contatta il server originale, se la procura e la fornisce all'utente. A questo punto mantiene la risorsa per richieste future (chiaramente gestendo la sua memoria, che è finita).

Chiaramente, i *miss* dei proxy sono più lenti di un accesso diretto al server originale, ma di contro la soluzione diventa viabile e effettivamente utile quando si riesce a raggiungere una certa quota di *hit*.

I proxy permettono a content provider meno distribuiti di distribuire contenuti più facilmente. Solitamente sono installati dagli ISP (sia privati che istituzionali), con l'effetto collaterale (positivo) della riduzione del traffico sia sui server originali che sulla linea di accesso a Internet dell'ISP stesso.

Una nota interessante è che il proxy si comporta contemporaneamente sia come **client** che come **server**: dal browser è visto come *server*, mentre dal server originale è visto come *client*.

5.1.1 GET condizionale

Potremmo chiederci come fa li server proxy ad assicurarsi di mandarci sempre la copia più aggiornata della risorsa richiesta.

La soluzione è, lato server originale, non inviare la risorsa se è gia presente in cache: il server proxy può usare la linea di richiesta `If-modified-since:` per specificare l'ultima versione che ha a disposizione, e il server può rispondere con una nuova copia (se esiste), o alternativamente con il codice **304** (*Not Modified*).

Questo mantiene chiaramente un piccolo traffico in circolazione sul link, che però è comunque più piccolo del traffico richiesto per inviare risorse complete, e quindi non si traduce in carichi significativi per la rete.

5.2 Protocollo HTTP/2

L'obiettivo degli aggiornamenti del protocollo HTTP è principalmente quello di ridurre il ritardo in richieste HTTP multi oggetto.

HTTP/1.1 ha introdotto richieste GET multiple eseguite in *pipeline* su una sola connessione TCP. Il server risponde *in-order* usando l'algoritmo **FCFS** (scheduling First Come First Served). Secondo questo algoritmo, gli oggetti più piccoli devono aspettare la trasmissione dietro gli oggetti più grandi (**HOL blocking**, *Head Of Line blocking*).

Inoltre, il meccanismo di ritrasmissione dei segmenti TCP persi può rallentare ulteriormente le trasmissioni.

Potremmo pensare di risolvere i problemi di HTTP/1.1 usando altri algoritmi di scheduling:

- Magari inviando prima gli oggetti più piccoli. Questo però ritarderebbe indefinitamente la trasmissione di oggetti più grandi, in quanto probabilmente ci sarà quasi sempre un'oggetto più piccolo;
- Usando un approccio **RR** (*Round Robin*), dove gli oggetti più grandi vengono divisi in *segmenti* più piccoli, e il server alterna fra i diversi client trasmettendo tali segmenti.

HTTP/2 ha aumentato la flessibilità lato server nell'inviare oggetti al client. In questo caso l'ordine di trasmissione degli oggetti è stabilito da codici di priorità inviati dai client, e da algoritmi più sofisticati lato server (come il RR visto prima). Inoltre, il server può inoltrare (*push*) ai client oggetti che non hanno richiesto.

In HTTP/2 restano comunque i problemi di stallo nel caso di ritrasmissioni di segmenti TCP persi: i browser sono invitati a mantenere più connessioni TCP parallele per ridurre gli stalli e aumentare il throughput complessivo.

Inoltre, non c'è sicurezza sopra il protocollo TCP.

HTTP/3 ha introdotto meccanismi di sicurezza, e controllo di errori per oggetto e congestioni su UDP. Approfondiremo questo aspetto quando parleremo del livello transport.

6 Lezione del 03-10-25

6.1 E-mail

L'**e-mail** è un'altra delle applicazioni di largo uso sviluppate su Internet. Come ogni altra applicazione, è supportata da diversi *protocolli*, fra cui **SMTP** (*Simple Mail Transfer Protocol*), **IMAP** (*Internet Message Access Protocol*), **POP** (*Post Office Protocol*, di cui la più nota è la versione *POP3*).

Nel sistema della posta elettronica ci sono 3 componenti principali:

- Gli **user agent**, cioè i client di posta elettronica usati dagli utenti per scrivere e ricevere messaggi di posta elettronica;
- I **server** di posta elettronica;
- Un **protocollo** di posta elettronica, prendiamo SMTP.

I mail server supportano:

- Una **mailbox** che contiene i messaggi in arrivo per ogni utente;
- Una **coda** di messaggi in uscita: questi vengono poi smistati nelle mailbox dei vari utenti;
- Il **protocollo** SMTP per ricevere i messaggi dai client e inviarli ad altri mail server (vedremo che i client non usano direttamente SMTP per richiedere la posta, ma sfruttano altri protocolli).

Come sempre, notiamo che con *mail server* si intende nello specifico il processo che gestisce le richieste degli utenti (e per estensione tutta la macchina (o macchine) che lo eseguono).

Il protocollo SMTP è un protocollo client-server: nonstante ciò, il mail server può comunicare con altri server, e in tal caso si comporta sia come client che come server.

- Si comporta come *client* quando deve inviare posta;
- Si comporta come *server* quando deve ricevere posta.

Questo è chiaro in quanto è il server che *invia* la posta a dover iniziare la comunicazione, mentre il server che *riceve* deve essere solo pronto, appunto, a ricevere.

6.1.1 Protocollo SMTP

Nello specifico, il protocollo SMTP è basato su TCP aperto alla porta 25. Il trasferimento è diretto, e come abbiamo già detto il server che invia si comporta come client per il server che riceve.

Ci sono 3 fasi di trasferimento:

- Handshake fra i due server;
- Trasferimento di messaggi;
- Chiusura della comunicazione.

L'interazione è richiesta/risposta come in HTTP (anche se le richieste vengono dette *comandi*), con messaggi codificati in ASCII a 7 bit.

I comandi contengono solo testo ASCII, mentre le risposte contengono un codice di stato seguito da testo, sempre ASCII, in linguaggio naturale. Questo è per un motivo diagnostico: i comandi client sono letti dalla macchina, mentre del server la macchina legge solo il codice di stato e il commento ASCII è di debug per i tecnici.

Facciamo un'esempio pratico per capire meglio il funzionamento del protocollo. Poniamo che Alice voglia inviare un messaggio a Biagio:

1. Alice `alice@unipi.it` usa l'user agent per comporre un messaggio e-mail al destinatario Biagio `biagio@altroente.edu`;
2. L'user agent di alice invia il messaggio al mail server di `unipi.it` via SMTP, che lo mettte nella coda dei messaggi;
3. Il mail server di Alice apre una connessione TCP, comportandosi come client, con il mail server `altroente.edu`;
4. Il messaggio di Alice viene trasferito verso tale mail server sulla rete TCP;
5. Il mail server di Biagio mette il messaggio nella mailbox di Biagio;
6. In un secondo momento, l'user agent di Biagio richiede la mailbox al suo mail server, che quindi gli invia il messaggio di Alice.

Qui il protcollo SMTP viene usato nella comunicazione fra l'UA di Alice ai server di `unipi.it`, e nella comunicazione fra questi e i server di `altroente.edu`. Per portare la posta dai server di `altroente.edu` all'UA di Biagio, invece, verrà usato un altro protocollo (come già detto IMAP o POP3).

Abbiamo quindi, per riassumere un ultima volta, che i mail server si comportano come *client* quando devono *inviare* messaggi ad altri mail server, e come *server* quando devono *ricevere* messaggi da altri mail server. Verso gli user agent, invece, sono sempre server: si richiede al server di *inviare* una mail, e si richiede al server di *scaricare la mailbox* corrente (il server non ci contatterà come client per inviarci la posta come farebbe con un altro mail server, ma siamo noi a doverlo invocare).

6.1.2 Parole chiave SMTP

Vediamo l'esempio dei messaggi che viaggiano in una connessione TCP, in plain text, durante lo svolgimento del protocollo TCP. Mettiamo ad esempio di leggere la comunicazione fra il server `unipi.it` al servizio di Alice dell'esempio precedente, quando questo inoltra la posta al mail server `altroente.edu` di Biagio.

Qui `S:` significa server (`altroente.edu`) e `C:` significa client (`unipi.it`):

```
S: 220 altroente.edu
C: HELO unipi.it
S: 250 Hello unipi.it, pleased to meet you
C: MAIL FROM: <alice@unipi.it>
S: 250 alice@unipi.it... Sender ok
C: RCPT TO: <biagio@altroente.edu>
S: 250 biagio@altroente.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Ciao Biagio !
```

```
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 altroente.edu closing connection
```

Iniziamo con l'handshake: il server invia 220 (tutto ok) e il client si introduce con `HELO` e il suo indirizzo: a questo punto il server risponde con `250` (acknowledge dell'`HELO`).

Da qui in poi il client può comunicare la posta che vuole inviare. Con `MAIL FROM` si indica l'indirizzo di posta elettronica dell'utente che sta inviando posta, a cui segue una risposta 250 dal server che segnala se quell'utente è riconosciuto. Con `RCPT TO` si indica poi l'indirizzo di posta elettronica dell'utente destinatario, a cui segue un'altra risposta 250 dal server che segnala se quell'utente è presente sul mail server.

Segue poi la sezione `DATA`, a cui il server risponde con 354, e quindi la mail vera e propria. Alla fine del messaggio (chiuso con un . su una nuova linea) il server risponde con un'altro 250, e il client chiude la connessione con `QUIT`. L'ultimo acknowledge è del server con 221 (chiudo connessione).

6.1.3 Confronto fra SMTP e HTTP

Si può dire che HTTP è un protocollo di tipo **pull**, mentre SMTP è un protocollo di tipo **push**: in HTTP lato client si richiedono risorse, in SMTP si inviano. Inoltre, in SMTP si usano connessioni *persistenti*.

Abbiamo poi che in HTTP ogni oggetto è incapsulato in una risposta, mentre in SMTP si possono avere trasferimenti *multipart* (più messaggi di posta per connessione TCP), e che in SMTP si usa ASCII su 7 bit.

6.1.4 Protocolli di richiesta

Come abbiamo detto, il protocollo SMTP è effettivamente usato solo per la comunicazione fra mail server e per inviare messaggi a mail server: per richiedere la posta dal server si usano altri protocolli come **IMAP** (*Internet Messagge Access Protocol*) e **POP** (*Post Office Protocol*). Questi protocolli forniscono la possibilità di fare richieste di messaggi, eliminazione, accesso a directory di messaggi presenti sul server, ecc...

Inoltre, servizi come Gmail e Hotmail forniscono interfacce Web (via il protocollo HTTP) costruite su SMTP (per inviare messaggi) e IMAP (o POP, in particolare POP3) per richiedere messaggi.

La differenza principale fra IMAP e POP è che *POP* era pensato per singoli dispositivi: la posta veniva scaricata su locale e rimossa dal server, per cui non vi si poteva accedere da altri dispositivi. *IMAP* risolve questo problema mantenendo i messaggi sul server e concedendone l'accesso da parte di più dispositivi. In verità, oggi la maggior parte dei mail server è configurato per mantenere in remoto anche la posta scaricata via POP3. Per client locali questo significa che IMAP e POP3 distinguono fondamentalmente fra due politiche più o meno aggresive di caching, mentre i client su HTTP usavano in ogni caso IMAP (non si scarica mai nessun messaggio sul disco locale).

Per concludere, da quello che abbiamo appreso in 4.2.1, possiamo dire che POP è *stateless*, mentre IMAP è *stateful* (mantiene stato sotto forma di mail precedenti).

6.2 Il DNS

DNS (*Domain Name System*) è il sistema che usiamo per tradurre *nomi di dominio* DNS, semplici da ricordare per gli umani, in *indirizzi IP*, semplici da usare per le macchine.

6.2.1 Motivazione del DNS

Di base, un'**indirizzo IP**v4 (IP versione 4) è rappresentato da un numero a 32 bit, diviso in 4 numeri da 8 bit rappresentati come unsigned e separati da punti: 8.8.8.8 (il servizio DNS di Google) sarebbe `00001000_00001000_00001000_00001000`. Un certo numero di bit dal più significativo vengono detti **maschera** di rete, ed identificano la parte dell'IP che identifica la rete locale: i bit successivi identificano gli host. Ad esempio, `10.104.111.163/24` significa che i primi 24 bit identificano la rete, e i successivi 8 gli host su quella rete.

Il problema è che gli indirizzi IP sono difficili da ricordare: gli umani preferiscono indirizzi leggibili come `tizio.caio.com`, cioè i cosiddetti **nomi di dominio**.

6.2.2 Realizzazione del DNS

Per tradurre da indirizzi leggibili a indirizzi IP (necessari per aprire connessioni TCP o UDP e quindi comunicare effettivamente col server) si usa il DNS.

I problemi del DNS sono quindi:

1. Mantenere l'**univocità** dei nomi di dominio verso gli indirizzi IP;
2. Permettere di risalire (**risolvere**) da un nome DNS all'indirizzo IP corrispondente.

La soluzione che è stata data è di creare un *database distribuito* implementato con una gerarchia di tanti *name server* che contengono **registri** di nomi di dominio. Protocolli a livello application permettono quindi agli host (e a loro volta i name server) di **risolvere** nomi di domini DNS in indirizzi.

Questo rappresenta una funzione base di Internet, implementata al livello application, e che distribuisce la maggior parte della complessità all'*"edge"* di internet.

Per assicurare l'univocità si sono formate *autorità* come **ICANN** (*Internet Corporation for Assigned Numbers and Names*), e la sussidiaria **IANA** (*Internet Assigned Numbers Authority*) che definiscono quali domini possono essere usati in base all'area di applicazione, geografica, ecc... Queste autorità delegano quindi ad altre autorità il compito di assegnare nomi, i cosiddetti identificatori *top-level*, **TLD** (*Top Level Domain*) a determinate regioni geografiche (`.uk`, `.io`, ecc...) e gestirne i domini, e la cosa può chiaramente ripetersi ricorsivamente, fino al cosiddetto livello *autoritativo*, dove la traduzione in IP può effettivamente accadere.

In particolare, l'autorità di registrazione dell'identificatore top-level (`.it`) risiede a Pisa all'interno del CNR.

Come sappiamo, esistono anche domini top-level associati non ad entità geografiche ma generici, come `.com`, `.org`, ecc... Questi sono amministrati direttamente dall'ICANN, mentre i registri sono detenuti da compagnie private.

6.2.3 Funzionamento del DNS

La risoluzione dei nomi di dominio viene fatta in maniera simile a quella degli indirizzi IP, partendo da destra verso sinistra per risalire al name server che contiene l'IP cercato. Ad esempio, nel DNS `www.google.com`, prima si controlla il `com` per capire il primo name server da contattare. Questo a sua volta prendera il `google` per contattare il prossimo name server, e così via. Il `www` è arcaico e viene mantenuto principalmente per ragioni storiche.

Abbiamo quindi che i problemi prima nominati vengono risolti rispettivamente nei seguenti modi:

1. L'*univocità* dei nomi di dominio è assicurata da **autorità** nazionali ed internazionali (ICANN, IANA, ecc...);
2. La risoluzione da un nome di dominio DNS all'indirizzo IP corrispondente è assicurata da **name server** che contengono **registri** di nomi di dominio.

Non è detto che le autorità che gestiscono un TLD gestiscono anche il registro corrispondente (abbiamo detto che questo non è il caso per i TLD generali come `.com`).

6.2.4 Risoluzione DNS

Vediamo i dettagli tecnici. Un server DNS comunica con un attraverso protocollo coi client, fornendo i servizi:

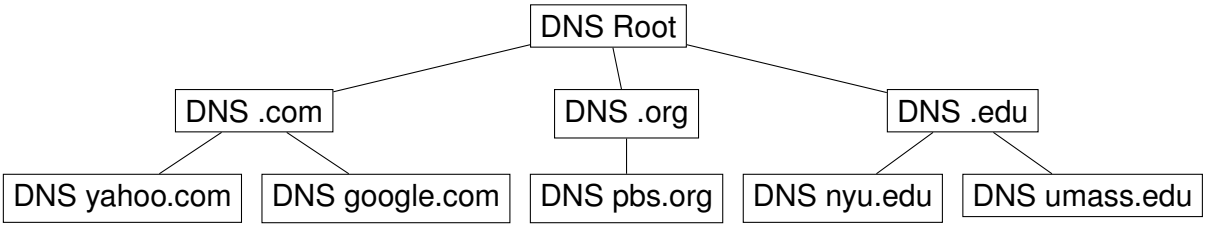
- **Risoluzione** da dominio a indirizzo IP;
- **Aliasing** degli **host**: più nomi possono puntare allo stesso IP;
- **Aliasing** dei server di **posta**: allo stesso dominio possono corrispondere web server come mail server;
- **Distribuzione** del **carico**: più server (con i loro IP) possono rispondere allo stesso nome di dominio per dividersi le richieste.

Abbiamo detto che il DNS è un database **distribuito**, quindi sviluppato gerarchicamente su più *name server*. Questa soluzione è stata scelta, al contrario di un database *centralizzato*, per una serie di motivi:

- Evita un singolo punto di fallimento;
- Riduce il volume di traffico su ogni name server;
- Il database centrale risulterebbe molto distante per larga parte degli utenti (distribuiti su tutto il mondo);
- La manutenzione è più semplice (si fa su unità più piccole).

6.2.5 Database DNS

La struttura gerarchica del DNS è modellizzabile come una serie di registri che rimandano l'uno all'altro:



I registri ad ogni livello sono raggruppati in diverse categorie:

1. **Root**, il registro base che punta agli altri registri;
2. **TLD** (*Top Level Domain*), che contiene i domini top-level nazionali (`.jp`, `.it`, ecc...) o generici (`.org`, `.edu`, ecc...);

3. **Autoritativi**, i server DNS che si raggiungono dai TLD (o da server DNS ad altri *sottodomini*) e che hanno effettivamente il compito di terminare la risoluzione per quel dominio restituendo un IP.

Quando un client vuole risolvere un DNS, ad esempio per `www.google.com`, compie i seguenti passi:

1. Si rivolge al root server per trovare il server DNS `.com`;
2. Si rivolge al server DNS `.com` per trovare il server DNS `google.com`;
3. Si rivolge al server DNS `google.com` per ottenere l'IP di `www.google.com`.

In verità, la situazione è più complicata: i server Root sono più di uno (13), e sono largamente replicati (più di 200 solo negli Stati Uniti). Questo è fondamentale in quanto la sicurezza del servizio DNS deve essere assicurata.

Addirittura, i server sia TLD che autoritativi vengono replicati, usando configurazioni *primario/secondario* o meccanismi come *anycast* per assicurare la ridondanza del servizio.

6.2.6 Server DNS locali

Ogni ISP fornisce solitamente un server DNS proprio, detto anche *Default Name Server*. Anche i DNS locali sono solitamente configurati con name server *primario/secondario* da contattare nel caso l'uno o l'altro fallisca.

Quando un host fa una richiesta DNS ad un server DNS locale, questo controlla la sua cache, comportandosi quindi da proxy, e inoltrando la richiesta al DNS Root (o molto più tipicamente a un servizio di risoluzione DNS, sopratutto nel caso di router che per motivi economici non implementano l'intero protocollo DNS) solamente se non è capace di soddisfare la richiesta da solo.

Il meccanismo di caching è supportato dalla struttura gerarchica dei nomi di dominio DNS: ad esempio se il DNS locale conosce `dns.nyu.edu`, potrebbe rispondere alle richieste di risoluzione di `engineering.nyu.edu` inviando richieste direttamente al server autoritativo `nyu.edu`, che ha già contattato.

Abbiamo quindi che un host connesso ad un ISP (come un comune router) si rivolge solitamente al servizio DNS fornito da tale ISP per le sue richieste DNS. Questo quindi usa un servizio di risoluzione pubblico come 1.1.1.1 (Cloudflare) o 8.8.8.8 (Google) per risolvere il DNS (si dice che si comporta da *forwarder* DNS), o in alcuni casi si comporta esso stesso come risolutore DNS contattando direttamente il server Root.

Nessuno ci nega (e a volte avviene) che l'host eviti il DNS locale rivolgendosi direttamente a un servizio di risoluzione.

6.2.7 Risoluzione iterativa/ricorsiva

La risoluzione dei nomi di domnio DNS da parte di un server DNS locale che si comporta come risolutore DNS (come da parte di un risolutore pubblico come quelli nominati sopra) può accadere in due modi principali: **iterativo** e **ricorsivo**:

- **Iterativo**: in questo caso il server DNS si rivolge ad altri server DNS accettando sia traduzioni complete che riferimenti ad altri server DNS: nel caso il server contattato non possa tradurre potrà almeno reindirizzarci verso un server che può farlo;

- **Ricorsivo**: in questo caso il server DNS accetta solo traduzioni complete, o al limite messaggi che segnalano il fallimento della richiesta. Questa è la modalità in cui vengono interrogati la maggior parte dei servizi di risoluzione pubblici.

  Dal punto di vista interno la soluzione che si ha è che ogni DNS server contattato si prende la briga di contattare il successivo con una nuova richiesta DNS, e non di restituire solamente l'IP del prossimo elemento della catena, effettivamente sollevando il carico dal server DNS locale.

7 Lezione del 06-10-25

Continuiamo la discussione del DNS.

7.0.1 Caching DNS

Vediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferentiVediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferenti.

Quando un qualsiasi server DNS trova una data traduzioni fra dominio DNS e indirizzo IP, la memorizza (per ridurre il traffico DNS) con un certo **timeout**, sostanzialmente una data di scadenza della traduzione (solitamente impostata a 2 giorni). Molto spesso i server TLD (che raramente cambiano) sono memorizzati per primi, in questo modo i server Root vengono visitati di rado.

Infine, possiamo assumere che sia il S/O usato dall'utente che utilizza il DNS, che l'applicazione (metti il *browser*) che usa per collegarsi in rete, mantengono una loro cache di traduzioni. Questo significa che l'applicazione, per risolvere un DNS, effettuerà i passaggi:

- Consulterà la sua cache interna;
- Se non trovato, consulterà la cache del S/O;
- Se non trovato, effettuerà un richiesta al server DNS locale;
- Questo cercherà nella sua cache e se non trovato continuerà con i passaggi visti da 6.2.5 a 6.2.7.

7.0.2 Record DNS

Vediamo la struttura dei *Resource Record* (**RR**) memorizzati dai server DNS.

Questi hanno struttura generale:

```
(name, value, type, ttl)
```

e diversi tipi, fra cui:

- `type=A` (*Address*): `name` è il nome dell'host, e `value` l'indirizzo IP;
- `type=CNAME` (*Canonical NAME*): `name` è l'*alias* per qualche nome *"canonico"*, e `value` il nome canonico;
- `type=NS` (*Name Server*): `name` è il dominio e `value` è il nome di dominio del server autoritativo per questo dominio;
- `MX` (*Mail eXchange*): `value` è il nome del mail server associato a `name`.

7.0.3 Protocollo DNS

Vediamo alcune specifiche del protocollo DNS usato per inviare richieste (*query*) e risposte (*reply*). Anticipiamo subito che il protocollo DNS si basa su UDP e non TCP.

Abbiamo quindi che la struttura delle richieste e delle risposte è la stessa:

```
2 byte                  2 byte
<id>                    <flags>
# domande (query)       # risposte RR (reply)
# autorita' RR (reply)  # RR aggiuntivi (reply)
...
```

L'identificatore (`<id>`) è su 16 bit e viene usato per riconoscere i destinatari delle risposte. I flag (`<flags>`) sono vari e specificano:

- Se è un messaggio *query* o *reply*;
- Se si richiede ricorsione (per messaggi query);
- Se la ricorsione è disponibile (per messaggi reply);
- Se la risposta è authoritative, cioè viene da un server DNS authoritative.

7.0.4 Inserire record in DNS

Vediamo quindi il processo usato per registrare nomi di dominio nel DNS. Mettiamo che la startup *"Alpha S.r.l."* voglia registrare un dominio per il suo sito:

- Dovrà registrare il nome rivolgendosi ad un'azienda specializzata, e fornendo il nome di dominio (che non deve essere già stato registrato), l'indirizzo IP del server authoritative primario e secondario;
- L'azienda specializzata inserirà record di tipo NS e A in un server TLD (mettiamo `.com`):

```
(alpha.com, dns1.alpha.com, NS) # RR NS dal nome di dominio al DNS
    autoritativo
(dns1.alpha.com, 212.212.212.1, A) # RR A dal nome del DNS
    autoritativo al suo indirizzo
```

- A questo punto Alpha S.r.l. dovrà configurare i suoi server DNS authoritative inserendo record di tipo A e volendo MX (per la posta elettronica):

```
(www.alpha.com, 212.212.212.2, A) # RR A dalla pagina web all'IP del
    server che la fornisce
(alpha.com, mail.alpha.com, MX) # RR MX al nome del mail server
(mail.alpha.com, 212.212.212.3, A) # RR A dal nome all'IP del mail
    server
```

7.1 Applicazioni P2P

Abbiamo visto una certa gamma di applicazioni in rete che adottano il paradigma client-server: ad esempio il Web, l'e-mail, e proprio adesso il DNS.

Vediamo adesso l'altro grande paradigma, quello del **Peer-to-Peer**. Ricordiamo che la caratteristica delle applicazioni P2P è che non c'è un server centrale sempre attivo, ma sistemi host finali comunicano fra di loro in maniera arbtiraria: i *peer* richiedono servizi ad altri peer, fornendo in cambio altri servizi.

Questa architetturà è *auto-scalabile* (più peer, più grande la rete), complessa da gestire e (vogliamo) il più possibile decentralizzata.

Ci sono diverse applicazioni realizzabili come P2P; possiamo classificarne alcune:

- **Condivisione contenuti**: i peer rendono disponibili contenuti multimediali o in generale file ad altri peer, in maniera decentralizzata. Dal punto di vista legale questo fornisce una piattaforma abbastanza sicura per lo scambio di contenuti illegali (film e musica pirata, ecc...). Celebre è l'esempio di *Napster*;
- **Messaggistica istantanea**: gli *username* degli utenti devono essere mappati a certi indirizzi IP. Quando un utente va online un *indice* è notificato col suo indirizzo IP: a questo punto altri utenti possono contattarlo in P2P a tale indirizzo.

7.1.1 Indici P2P

Vediamo che in entrambi casi presentati nella scorsa sezione (nel primo non si è detto ma è chiaro riflettendo sulle soluzioni tecniche) abbiamo bisogno di **indici**: per stabilire quali peer contattare per ottenere una data risorsa, bisogna avere un modo di effettuare ricerche sui loro indirizzi IP.

Un indice P2P è organizzato come un database di coppie chiave-valore, ad esempio:

```
Led Zeppelin IV, 203.17.123.38
```

I peer effettuano *query* a questo database per trovare i peer che possiedono date risorse; i peer possono anche *inserire* nuove coppie chiave-valore (ad esempio per segnalare che possiedono una nuova risorsa).

Il database può essere realizzato in più modi:

1. Si può dislocare un singolo **indice centralizzato** per tutti i peer. I problemi di questo sistema sono ovvi: questo può essere facilmente individuato e messo fuori uso (cause legali, guasti, ecc...) o sovraccaricato per rendere l'intero sistema P2P inutilizzabile.

   Diciamo infatti che applicazioni P2P con indici centralizzati sono P2P *"ibridi"*, con funzionalità client-server per l'ottenimento degli IP dei peer.

   Un sistema di questo tipo veniva usato da *Napster*, con celebri risultati (il server centrale viene chiuso e l'intera applicazione cade);

2. Si può sfruttare il **query flooding** per realizzare un indice *decentralizzato*: secondo questo paradigma ogni peer forma una parte dell'indice.

   In questo caso la fase di distribuzione di contenuti è banale: non bisogna collegarsi ad un indice centralizzato, ma ogni peer diventa parte dell'indice nel momento stesso in cui inizia a condividere un contenuto.

   Il problema giunge in fase di ricerca di contenuti: abbiamo che il client che deve ricevere contenuti deve limitarsi a *chiedere* agli altri peer *"vicini"* se hanno il contenuto desiderato. Definiamo i peer come *vicini* se hanno fra di loro una connessione TCP attiva. La richiesta può chiaramente diffondersi di vicino in vicino visitando l'intera rete o un suo sottoinsieme, e in questo modo si implementa effettivamente, se non per "forza bruta", un indice decentralizzato.

   La rete che si viene a formare fra tutti i peer vicini in un sistema P2P di questo tipo viene detta **rete overlay**.

Per ridurre il traffico sulla rete P2P (sistemi di questo tipo tendono a emanare molto traffico) si può usare il **limited-Scope** query flooding, cioè inviare richieste non a tutti gli amici, ma a un sottoinsieme, magari impostando un time-to-live per le richieste oltre il quale i vicini al prossimo *"hop"* non necessitano più inoltrare la richiesta oltre.

Un'altro problema è chiaramente il *bootstrapping*: come trovare il primo set di vicini? In questo caso si possono usare scansioni, cache temporanee di peer, peer preimpostati (che hanno però il problema della centralizzazione) o l'intervento manuale dell'utente.

Questa era la soluzione usata da servizi come *LimeWire*;

3. Un'approccio che combina il meglio dei due scorsi è un'approccio **gerarchico**. In questo caso prevediamo una rete simile a quella usata nel query flooding, ma che prevede dei *Super Nodi* (**SN**) (o *Super Peer*, **SP**), cioè peer con alta bandwidth e disponibilità alle connessioni. Ogni SN gestisce una rete locale, con un suo indice locale, e i peer di quella rete lo interrogano come avrebbero interrogato il server centrale del primo caso. A questo punto la trasmissione avviene normalmente fra peer, ammesso che i peer siano all'interno della stessa rete locale gestita dal SN. Non si esclude la possibilità che un SN non trovi il contenuto richiesto nel suo indice locale: in questo caso è libero di collegarsi ad altri adottanto sostanzialmente il query flooding, che però sarà più efficiente per diverse ragioni (meno supernodi formano una rete più compatta, i supernodi hanno specifiche migliori di trasmissione, i supernodi sono pensati appositamente per questa operazione (a differenza dei peer normali che vogliono perlopiù scaricare e non dare), ecc...).

   Chiaramente il problema sarà di non *"centralizzare"* troppo i SN, cosa che li renderebbe obiettivi di attacchi, cause legali, o comunque renderebbe più fragile l'intero sistema.

   Questa soluzione veniva usata da servizi successivi a *LimeWire*, come ad esempio *FastTrack*;

4. Un approccio interessante è quello dato dalle **DHT** (*Distributed Hash Table*). In questo caso si mira a realizzare l'associazione chiave-valore attraverso un'**hash table distribuita** fra i nodi di un *overlay network*.

   Ci dotiamo di una funzione di hashing *consistente*, cioè dove la rimozione di un bucket (di un bersaglio per la funzione) richiede lo spostamento delle sole chiavi che arrivavano a quel bucket. Usiamo quindi questi bucket per partizionare i peer: a certe chiavi corrisponderanno certi peer che conterrano i dati associati alle chiavi.

   A questo punto la ricerca di contenuti si può fare come per il query flooding, cioè inviando richieste ai vicini finché non si trova un degli host che corrisponde alla chiave cercata (e quindi i contenuti desiderati). Una soluzione più intelligente è però strutturare la *topologia* dell'overlay network in modo che i salti possano essere informati: questo è fattibile quando si usa un apposito algoritmo di hashing, dove ogni nodo ha una qualche informazione su quali dei suoi nodi vicini sono più o meno *"vicini"* alla chiave cercata.

   Questo è l'approccio usato da *BitTorrent* e *eMule*.

8 Lezione del 08-10-25

8.1 Confronto prestazionale fra CS e P2P

Vediamo di valutare quale approccio, fra client-server e peer-to-peer, per un job semplice come un trasferimento di file *minimizza* il **tempo di trasferimento**.

Poniamo di avere N peer (o client, comunque n host che vogliono ricevere il file) e un server che contiene il file (di dimensione F). All'istante in cui il file viene reso disponibile ogni peer richiede il file, e vogliamo minimizzare il tempo che passa affinché tutti lo abbiano ricevuto.

Sia u_s la capacità di upload del server, u_i la capacità di upload dell'i-esimo peer e d_i la capacità di download dell'i-esimo peer.

- Con l'approccio **client-server**, il server deve inviare N copie del file di dimensione F, che con capacità di upload di u_s dà un tempo minimo di NF/u_s. Ogni client deve quindi ricevere la sua copia del file, che con capacità di download di d_i diventa F/d_i. Visto che il più lento è quello che pregiudica tuta la statistica, prendiamo la sua capacità di download come $d_{\min}$ e rifiniamo il bound:

  $$T_{cs} \geq \max\left(\frac{NF}{u_s}, \frac{F}{d_{\min}}\right)$$

  Notiamo che questo bound è $O(N)$.

- Con l'approccio **peer-to-peer**, il server dovrà inviare al minimo una copia del file, per cui il bound diventa F/u_s. Il client più lento sarà comunque un bottleneck, per cui c'è il termine $F/d_{\min}$. A questo punto l'ultimo bound viene preso come il tempo impiegato a trasferire il file a tutti, se tutti distribuiscono, cioè $NF/(u_s + \sum u_i)$. Questo dà il bound:

  $$T_{p2p} \geq \max\left(\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum u_i}\right)$$

  Vediamo che il termine significativo è l'ultimo: il primo non dipende da N, mentre il secondo lo fa in maniera "lasca" (aumentando N troveremo host più lenti, ma non linearmente e prima o poi stabilizzandoci). Il terzo termine dipende da N, ma ancora non linearmente. Posto $\bar{u}$ come la velocita media di upload dei peer, possiamo approssimare come:

  $$T_{p2p} \sim \frac{NF}{u_s + \sum_1^N u_i} \approx \frac{NF}{u_s + N\bar{u}}$$

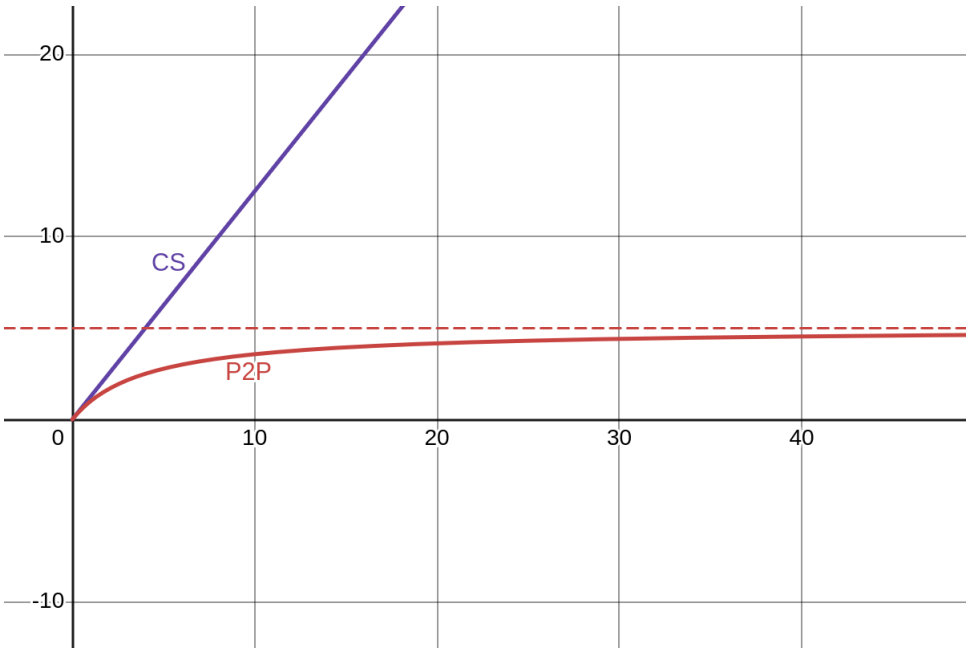
  che per $N \to \infty$ è:

  $$\lim_{N\to\infty} \frac{NF}{u_s + N\bar{u}} = \frac{F}{\bar{u}}$$

  ammesso di conoscere $\bar{u}$.

  Abbiamo quindi che il bound è migliore rispetto a quello dell'approccio client-server, in quanto prima o poi converge ad un valore finito.

Concludiamo tracciando su un grafico l'andamento complessivo dei bound più svantaggiosi dei tempi di trasferimento:



8.2 Protocollo BitTorrent

Approfondiamo le reti P2P, e le *overlay network*, studiando un protocollo reale: il **protocollo BitTorrent**.

In BitTorrent non è previsto un server centrale che distribuisce il file. Ogni file è diviso in più **chunk**, solitamente di 256 KiB ciascuno.

Un **torrent** è un gruppo di peer che si impegnano a scambiarsi chunk di un file. Un server più o meno centralizzato detto **tracker** traccia i peer che partecipano ad un torrent.

- Quando un utente vuole scaricare un file, chiede al tracker la lista dei peer che partecipano al torrent dedicato a quel file. Solitamente questo è un *metafile* che contiene l'IP del tracker, a cui si può quindi richiedere il torrent vero e proprio.
- I peer che l'utente riceve vengono detti **vicini**: sono connessi da una connessione TCP, e possono quindi scambiarsi dei messaggi (chunk del file). Possiamo definire così gli utenti collegati ad un torrent:
  - **Leecher**: (*"sanguisughe"*), non hanno una copia completa del file e lo stanno ancora scaricando;
  - **Seeders**: (*"alimentatori"*), hanno già il file, e lo rendono disponibile agli altri.
- Il trasferimento da seeder a leecher si fa un chunk per volta, adottando una politica **Rarest-First** (*RF*): si inizia a scaricare dal chunk che meno peer sul torrent hanno disponibile. Questo è chiaramente mirato a ottenere i chunk che potrebbero sparire dalla rete il prima possibile.

  La lista dei chunk disponibili è fatta richiedendo periodicamente a ogni peer quali chunk possiedono.

- Man di mano che i leecher ottengono chunk, devono anche iniziare a rimetterli nella rete ai leecher nuovi arrivati. Su BitTorrent si usa la politica **Tit for Tat** (*TT*): si inviano chunk ai 4 peer che ci stanno inviando chunk a frequenza più alta. Questi top 4 vengono ricalcolati ogni 10 secondi, e gli altri vengono *"strozzati"* (non più serviti).

  Questo significa che i peer potrebbero legarsi eccessivamente fra di loro: ogni 30 secondi si sceglie allora un'altro peer a caso, facendo una previsione *ottimistica* (quel peer potrebbe arrivare fra i nuovi top 4 più frequenti).

  In ogni caso, l'obiettivo è quello di trovare buoni partner per il trasferimento file, in modo da ottenere il file più velocemente possibile.

La politica *Tit for Tat* di BitTorrent è chiaramente pensata per scoraggiare i *freeloader*: non si può scaricare se non si dà qualcosa in cambio, cioè si partecipa attivamente al trasferimento rimettendo chunk in circolo.

8.3 Streaming video e CDN

Vediamo come si possono creare tecnologie per il trasferimento massiccio di contenuti multimediali, con riferimento allo **streaming video** e i **CDN** (*Content Distribution Networks*).

I problemi sono la **scala** (come raggiungere molti utenti?) e l'**eterogeneità** (diversi utenti hanno diverse caratteristiche di trasmissione, dispositivi di visualizzazione, ecc...).

La soluzione è infrastruttura **distribuita** al livello application.

8.3.1 Video

Un **video** è una sequenza di immagini dette **frame** mostrate a frequenza costante (standard 24, 30, e 60 **FPS** (*Frames Per Second*)). Un **frame** è un'array di pixel, dove ogni pixel è rappresentato da 3 valori (corrispondenti ai colori primari additivi *rosso*, *verde* e *blu*).

La tecnica del **coding** consiste nell'usare ridondanza intrinseca dentro e fra i frame per ridurre i bit usati nel processo di codifica:

- Si usa la correlazione **spaziale**, dentro i singoli frame;
- Si usa la correlazione **temporale**, fra diversi frame.

Esistono più tipi di coding, fra cui notiamo:

- **CBR** (*Constant Bit Rate*), a bitrate costante;
- **VBR** (*Variable Bit Rate*), a bitrate variabile;

8.3.2 Streaming

Lo **streaming** di video allocati in remoto consiste nel spedire contenuti video abbastanza velocemente da renderli disponibile in *tempo reale* (a differenza del semplice download, che prevede una fase iniziale di acquisizione e una seconda fase di visualizzazione quando il file è ormai già tutto sul disco locale).

Il server dovrà quindi inviare al client frame con una frequenza prefissata (quella del contenuto video), e il client potrà mostrarli appena arrivati. Assunta una rete ideale,

quindi a ritardo *costante*, questo ci permetterebe di visualizzare il video così com'è (al pari di tale ritardo costante).

Purtroppo la rete non è ideale, e il ritardo è quindi *variabile* nel tempo. Possiamo quindi sfruttare il processo del **buffering**: aspettiamo un po' lato client prima di mostrare i frame (per un tempo detto *client playout delay*), mettendoli nel frattempo in un *buffer* di memoria. Quando iniziamo a visualizzare i frame, ci aspettiamo che il buffer sia abbastanza *"pieno"* da permettere la visualizzazione di un segmento di video abbastanza lungo da permettere l'acquisizione dei frame successivi, e così via.

9 Lezione del 10-10-25

Torniamo sull'argomento dello streaming video.

9.1 Tecniche avanzate per lo streaming

Avevamo introdotto l'idea del **playout delay**, dove bufferizzaviamo i dati video in ingresso per "spostare in avanti" in qualche modo la zona di dati compromessa dal *jitter* (irregolarità) della rete.

Visto che il jitter può essere massiccio, e la quantità di dati da inviare non indifferente, esistono alcune strategie più effiaci come ad esempio **DASH**.

9.1.1 DASH

DASH (*Dynamic, Adaptive Streaming over HTTP*) è una tecnologia che prevede di memorizzare i file video in più chunk, ogninuo memorizzato con ridondaza a diversi bitrate. Un *manifest file* fornirà gli URL di tutti i chunk, individuati quindi da indice nel video e livello di compressione.

Il *client* dovrà a questo punto misurare la velocità di trasmissione del server, e quindi consultare il manifest file: sulla base delle misurazioni fatte richiede il chunk di qualità massima sostenibile alla largezza di banda corrente. In questo modo può variare la qualità del video nel tempo, in modo da rispondere in maniera abbastanza elastica alle irregolarità della rete.

Questo richiede *"intelligenza"* dal client: questo deve sapere:

- *Quando* richiedere i chunk in modo da non incorrere né in starvation (restare senza chunk) né in overflow (non avere più spazio per i chunk);
- *A che bitrate* richiedere i chunk sulla base della banda disponibile;
- *Dove* richiedere i chunk: se i server sono duplicati dovrà scegliere il più efficiente.

9.2 CDN

Vediamo un'altra tecnologia, questa volta riguardante la *scalabilità* dei server che forniscono gigantesche quantità di contenuti video a numeri altrettanto grandi di utenti, simultaneamente, in tutto il mondo. Questo è il problema che viene affrontato dai **CDN** (*Content Distribution Network*).

L'idea è di copiare e servire diverse copie dei video su diversi server in siti geograficamente distribuiti. Esistono due approcci per la dislocazione di tali server:

- **Enter deep**: usare server CDN vicini agli access network: l'idea è di spostarsi il più vicino possibile agli utenti;

- **Bring home**: meno cluster, più grandi, distribuiti in **POP** (*Points Of Presence*) *vicini* ma non *sulla* rete di accesso degli utenti.

Facciamo un'esempio pratico, guardando al processo che un servizio di distribuzione segue per fornire un contenuto a un utente. Abbiamo che il meccanismo principale di redirezione è realizzato sfruttando il DNS:

- Il CDN (diciamo `kingcdn.com`) mantiene più copie del contenuto su diversi *nodi* CDN (magari `kingcdn.com/<slug-contenuto-1>`, `kingcdn.com/<slug-contenuto-2>`, ecc...).
- Il servizio di distribuzione rende disponibile il contenuto a `video.netcinema.com/<slug-contenuto>`;
- L'utente richiede un contenuto dal CDN, collegandosi a `netcinema.com` e navigando fino al contenuto richiesto a `video.netcinema.com/<slug-contenuto>` (l'opzione più probabile è che questo contenuto si trovi in un blocco HTML di tipo `<video>`);
- A questo punto, il DNS locale dell'utente chiede l'IP di `video.netcinema.com` ai server autoritativi di `netcinema.com`, che risponde con un record CNAME corrispondente all'IP del server autoritativo di `kingcdn.com`;
- L'utente viene quindi rediretto verso il nodo più vicino (potrebbe anche scegliere un'altro percorso se si verificassero variazioni (in peggio) del servizio). Da qui in poi il video è trasmesso dal nodo del CDN su HTTP.

9.3 Programmazione socket

Abbiamo introdotto il concetto di **socket** (dall'inglese per *presa*) come astrazione della comunicazione in rete per i processi in esecuzione su un S/O.

Lato pratico, il S/O renderà disponibili chiamate a sistema per effettuare almeno alcune operazioni base:

- Apertura di un socket: questo potrà essere **TCP** o **UDP**;
- Invio di dati al socket (o scrittura sul socket);
- Ricezione di dati al socket (o lettura dal socket).

Ricordiamo che il TCP è un protocollo orientato alla connessione, sicuro e affidabile, mentre UDP è un protocollo semplice per il trasferimento diretto ma non affidabile di dati.

9.4 Reti in connessione diretta

Finite le applicazioni, riprendiamo il discorso dal *basso*. Se avevamo discusso come creare applicazioni che sfruttassero la rete, adesso vogliamo discutere come la rete riesce in primo luogo a trasmettere bit fra più processi, o anche solo fra più calcolatori.

Iniziamo col considerare come nodi **adiacenti** nella rete comunicano fra di loro, cioè come 2 host con apposite interfacce di rete comunicano attraverso una connessione diretta: il caso più semplice è quello di un link punto-punto fisico.

Ci sono già diversi problemi da analizzare: come rendere la comunicazione **affidabile** (*framing*, *rilevamento errori* e *correzione*, ecc...). Introdurremo quindi il protocollo **PPP** (*Point to Point Protocol*), i protcolli ad **accesso multiplo**, e le **LAN** (*Local Area Network*).

9.5 Link punto-punto fisici

Supponiamo di avere 2 host (2 macchine) che vogliono comunicare fra di loro. Questi saranno provvisti ciascuno di un interfaccia per una porta seriale che potranno comandare (alto o basso), e le loro porte seriali saranno collegate da un link di qualche tipo (rame, fibra ottica, ecc...).

Le porte seriali permettono la **codifica** di bit attraverso variazioni di stato del mezzo fisico del link.

L'interfaccia della macchina mittente è detta **trasmettitore**, mentre l'interfaccia della macchina destinatario è detta **ricevitore**: bisognerà stabilire un protocollo per la comunicazione fra questi.

In questo caso il protocollo è semplice il *trasmettitore* dovrà occuparsi di **codificare** sequenze di bit in variazioni di stato sul mezzo di link, mentre il *ricevitore* dovrà occuparsi di **decodificare** tali variazioni di stato riportandole in bit sulla macchina destinatario.

9.5.1 Data link

Una caratteristica fondamentale dei link è che sono *inaffidabili*: il canale di trasmissione non è mai ideale (degradazione del segnale, rumore, interferenze, ecc....) e quindi la sequenza codificata potrebbe arrivare al ricevitore radicalmente cambiata rispetto a quella iniziale.

Chiameremo *bit error rate* la frequenza di bit persi sul mezzo di comunicazione.

Per gestire l'inaffidabilità introdurremo un livello superiore a quello di **link fisico**, detto di **link dati** (*data link* o anche *livello logico*).

- La prima cosa che prevede il data link è il **framing**: si divide la comunicazione in *payaload* (insiemi di bit), e si incapsulano i tali payload fra *header* e *trailer* (sostanzialmente intestazioni e terminazioni), andando a formare il cosiddetto **frame**. Da qui in poi, con frame intenderemo sia l'intero frame che il suo payload a seconda del contesto.

  Nell'**header** potremmo inserire diverse informazioni: prima fra tutte l'indice del frame nell'intero blocco di dati da trasferire.

  Se la frequenza di errore è comparabile con la lunghezza dei frame, avremo sicuramente perdite di dati: riducendo la lunghezza dei frame potremmo avere più fortuna.

  Questa tecnica introduce chiaramente un certo *overhead* di trasmissione, dato dal dover introdurre header e trailer. Ci permette però di dividere la trasmissione in unità contenute, su cui è più semplice fare controllo degli errori;

- Altri servizi sono forniti per il controllo degli errori di trasmissione: l'**error detection** mira a rilevare gli errori, mentre l'**error correction** mira a rilevarli e correggerli. Questo è piuttosto semplice nel caso di trasmissioni binarie: assunto di avere un rilevamento dell'errore granulare al bit, correggerlo significherà semplicemente invertirlo.

  Una soluzione alternativa è comunque quella della **ritrasmissione** dei dati in errore, che implica però una comunicazione all'*indietro*, dal ricevitore al trasmettitore per la segnalazione dei frame corrotti.

- Ci sono poi altri servizi che potremmo voler fornire: ad esempio il **controllo di flusso** che permetta di moderare in qualche modo la frequenza di trasmissione.

Ad esempio il ricevitore potrebbe voler segnalare al trasmettitore di dover ridurre il bitrate, magari per problemi di overflow.

- Infine, vogliamo distinguere se offriamo servizi **half-duplex** o **full-duplex**:
  - **Half-duplex**: entrambi i nodi possono trasmettere, ma solo uno per volta;
  - **Full-duplex**: entrambi i nodi possono trasmettere, in *parallelo* (quindi contemporaneamente).

  Configurazioni dove solo un nodo può trasmettere sono dette **simplex**.

9.5.2 NIC

Il livello data link è implementato nella cosiddetta **NIC** (*Network Interface Card*) all'interno di ogni host sulla rete. Questa implementa il livello data link e di conseguenza il livello fisico. Può essere Ethernet, WiFi, ecc...

La NIC è collegata al bus periferiche di un calcolatore (ad esempio bus PCI) e implementa una o più delle funzionalità offerte dal livello data link. Fa questo combinando *hardware*, *software* e *firmware*: l'importante è che sia capace di offrire al calcolatore i datagrammi inviati al livello data link (in questo sia il NIC che il calcolatore vedono il livello data link).

10 Lezione del 13-10-25

10.1 Rilevamento errori

Il *rilevamento degli errori* (**error detection**) è il meccanismo attraverso il cui verifichiamo se i dati arrivati attraverso il link fisico sono corretti.

Dobbiamo innanzitutto stabilire un codice di **rilevamento** degli errori. Per ogni frame di livello Datalink trasmettiamo d bit di dati (i bit D) e r bit di *rilevamento errore*, o di **ridondanza** (i bit R). Minuscolo è il numero di bit, maiuscolo è il campo di bit vero e proprio. Sia i bit D che i bit R vengono trasmessi attraverso il link, che ricordiamo è *suscettibile ad errori*.

Per ricavare i bit R si applica una certa funzione $H()$ (sostanzialmente di *hashing*) sui bit D. Assunto che mittente e destinatario usino la stessa funzione $H()$, una volta ricevuti i dati il destinatario potrà applicare $H()$ sui bit D ricevuti e ottenere la sua copia dei bit R da confrontare con quelli ricevuti.

Se i bit R corrispondono non ci sono stati errori (a meno della sfortunata circostanza in cui sia i bit D che R sono stati alterati per risultare erroneamente corretti, si sceglie la funzione $H()$ in modo che questo sia difficilmente vero), altrimenti qualcosa è andato storto nella trasmissione sul link.

I bit R, nel contesto del *framing* visto in 9.5.1, vengono inseriti nel *trailer* del frame.

10.1.1 Algoritmi di rilevamento errori

Vediamo alcune possibili funzioni $H()$ usate per fare rilevamento errori.

- **Parity checking**: posto che si voglia trasferire una parola da d bit: si sceglie un solo bit R di ridondanza, determinato come segue:
  - Se il numero di bit a 1 fra i bit D è pari, si imposta il bit in R a 1;

– Se il numero di bit a 1 fra i bit D è dispari, si imposta il bit in R a 0.

In questo caso il bit in R viene detto **bit di parità** (*parity bit*). Questo algoritmo viene detto di tipo **even parity**: l'algoritmo opposto si direbbe **odd parity**.

Questo algoritmo funziona bene solo nella circostanza in cui il numero di errori sui bit d è dispari (per quanto ci riguarda, 1). Quando si trasferiscono quantità relativamente basse di bit per parola su canali abbastanza affidabili (8, 16, 24 bit su rame o simili) abbiamo che è abbastanza sicuro. Con parole più grandi chiaramente può essere molto più soggetto ad errori.

- **Checksum**: questo sistema, detto della *"somma di controllo"* viene usato in UDP e TCP. Non si riferisce principalmente a frame, quindi parleremo di blocchi di dati generici.

  In questo caso si prende ogni blocco di dati e si divide in frammenti da n bit. Sommando questi frammenti si ottiene un campo R su $r = n$ bit (quindi modulo 2^n) detto *checksum*, che può essere ricalcolato lato destinatario per fare rilevamento degli errori.

  Questo metodo è più sensibile rispetto al bit di parità, e viene usato perlopiù a livello di *trasporto* UDP o TCP (più che a livello *data link*, dove si fanno altri tipi di controlli meno sensibili).

- **CRC** (*Cyclic Redundancy Check*): questo è il metodo usato più spesso in ambito di *Datalink*.

  Supponiamo di avere R bit dati, e un *pattern* detto **generatore** G di $r+1$ bit (dove ricordiamo che R sono i bit di ridondanza e r il loro numero). Mittente e destinatario dovranno entrambi conoscere il generatore.

  L'idea è quindi che in fase di trasmissione si prende un numero R di r bit tale che la concatenazione $D|R$ dei bit D ed R sia divisibile per G. Notiamo che le somme in CRC si fanno *modulo 2*, quindi senza riporti. In questo caso la relazione appena descritta sarà:

  $$D \cdot 2^r \oplus R = nG$$

  dove $D \cdot 2^r$ è semplicemente lo shift a sinistra di r bit di D, $\oplus$ lo XOR (che implementa la somma modulo 2), e n una costante moltiplicativa naturale qualsiasi, che assicura che il lasto sinistro è divisibile per G.

  Quando il destinatarrio riceve il frame la verifica è molto semplice: ha D ed R ottenuti sul link fisico, e noto G può effettuare la divisione. Se il resto è diverso da 0, allora si rileva un errore di trasmissione.

  Resta quindi da vedere come R viene calcolato nella pratica. Vogliamo:

  $$D \times 2^r \oplus R = nG \implies D \times 2^r = nG \oplus R$$

  come ci è concesso dalle proprietà dello XOR, per cui R dovrà soddisfare:

  $$R = \text{mod}\,(D \times 2^r, G)$$

  dove *mod* indica il resto della divisione fra i due argomenti (cioè l'operatore modulo).

10.2 Correzione errori

Dopo aver discusso il *rilevamento degli errori*, vediamo come procedere nella **correzione degli errori** in caso se ne rilevino.

La soluzione più semplice sarebbe quella di ritrasmettere i frame sbagliati. Se poi si avesse un algoritmo di rilevamento errori che restituisse esattamente *quali* bit sono errati, basterebbe commutarli per risolvere gli errori.

In questo caso non serve più un codice a *rilevamento* degli errori, ma un codice a **correzione** degli errori. Per ogni frame trasmettiamo d bit di dati (i bit D) e r bit di *correzione errore*, o ancora di **ridondanza** (i bit EDC, da *Error Detection Code*).

Questi sono simmetrici al codice definito per il rilevamento errori: la differenza è che dati D ed EDC, lato destinatario possiamo rilevare esattamente l'errore di trasmissione (se c'è stato).

10.2.1 Algoritmi di correzione errori

Vediamo alcuni (1) modi per calcolare i bit EDC ed effettuare quindi la correzione degli errori.

- **Parity checking bidimensionale**: si sistemano i bit D in una struttura matriciale, e si calcolano i bit di parità per ogni riga e per ogni colonna.

  Ad esempio, si può dire:

  $$D = \{0, 1, 0, ...\} = \{d_1, d_2, d_3, ...d_d\} \Rightarrow D_m = \begin{pmatrix} d_{1,1} & d_{1,2} & ... & d_{1,j} \\ d_{2,1} & d_{2,2} & ... & d_{2,j} \\ ... & ... & ... & ... \\ d_{i,1} & d_{i,2} & ... & d_{i,j} \end{pmatrix}$$

  Posti i e j tali che $i \cdot j = d$ (numero di bit in D). A questo punto si orla D_m con i bit di parità di *riga* (parity_{row}) e i bit di parità di *colonna* (parity_{row}):

  $$\begin{pmatrix} D_m & \text{parity}_{row} \\ \text{parity}_{col} & \end{pmatrix} = \left(\begin{array}{cccc|c} d_{1,1} & d_{1,2} & ... & d_{1,j} & d_{1,j+1} \\ d_{2,1} & d_{2,2} & ... & d_{2,j} & d_{2,j+1} \\ ... & ... & ... & ... & ... \\ d_{i,1} & d_{i,2} & ... & d_{i,j} & d_{i,j+1} \\ \hline d_{i+1,1} & d_{i+1,2} & ... & d_{i+1,j} & \end{array}\right)$$

  dove

  $$\text{parity}_{row}: \quad d_{r,j+1} = \text{parity}\left(\sum_{c=1}^{j} d_{r,c}\right)$$

  $$\text{parity}_{col}: \quad d_{j+1,c} = \text{parity}\left(\sum_{r=1}^{i} d_{r,c}\right)$$

  A questo punto un errore su un singolo $d_{r,c}$ verrà rilevato in quanto incrocierà una riga e una colonna (l'intersezione andrà commutata). Due o più errori su righe e colonne disgiunte veranno similmente rilevati, a meno di casi di errori multipli su più righe e colonnne (che potrebbero addirittura dare falsi positivi). Infine, un numero pari di errori sulla stessa riga o sulla stessa colonna ci permettono di rilevare, ma non correggere errori (vedremo 2 colonne sbagliate e una riga giusta, o viceversa, senza poter quindi incrociare).

  Chiaramente i bit EDC nel trailer saranno molti di più: per la precisione $i + j$.

10.3 Trasferimento dati

Abbiamo visto alcuni algoritmi di *rilevamento errori*, introdotto l'ipotesi di effettuare un *reinvio* dei dati corrotti nel caso di errori, e visto anche un algoritmo di *rilevamento e correzione errori*.

Vediamo adesso come realizzare un livello superiore a quello data link, riportando indietro l'ipotesi del reinvio dati, cioè il cosiddetto livello *transfer* o **trasferimento**, che considereremo un livello *affidabile*.

Le considerazioni che facciamo adesso saranno quelle che nel modello OSI vengono implementate nel cosiddetto livello **transport**.

L'idea è quella di usare i servizi offerti dal livello data link per realizzare un'ulteriore astrazione, quella appunto di *trasferimento affidabile*.

Quello che farà la componentistica di livello transport sarà interagire con un *livello superiore* che fornirà *dati*: questi verranno incapsulati in un **pacchetto**, a sua volta incluso come *payload* di un frame di livello Datalink, e quindi spedito sulla lniea inaffidabile vista finora.

10.3.1 Primitive di trasferimento

Per implementare questo tipo di livello *transfer* ci doteremo quindi di alcune primitive a servizio di un'altro *livello superiore*:

- `rdt_send()`: chiamata dal livello superiore (nella macchina *trasmettitore*), implementa l'invio sul canale affidabile (cioè implementa un protocollo **RDT** (*Reliable Data Transfer*));
- `udt_send()`: implementa l'invio sul canale inaffidabile fino al ricevitore;
- `rdt_rcv()`: implementa il ricevimento sul canale affidabile lato ricevitore, cioè ottiene i dati sul canale inaffidabile, e li corregge (se necessario);
- `deliver_data()`: si occupa di inoltrare i dati ottenuti attraverso il protocollo RDT al livello superiore (nella macchina *ricevitore*).

Notiamo che le primitive `udt_send()` e `rdt_recv()` dovranno implementare un qualche tipo di comunicazione bidirezionale (ad esempio se la `rdt_rcv()` vuole chiedere il reinvio di un frame perso).

10.3.2 Macchine a stati finiti

Per descrivere il protocollo **RDT** useremo il formalismo della *macchina a stati finiti* (**FSM**, *Finite State Machine*).

Una macchina a stati finiti rappresenta un *automa* dotato di **stati** e **transizioni** fra tali stati: dato uno stato ed un evento si può determinare la transizione successiva, e quindi come si evolve il protocollo.

Nelle prossime sezioni definiremo *iterativamente* versioni sempre più accurate di RDT rispetto a un qualche protocollo reale.

10.4 RDT 1.0

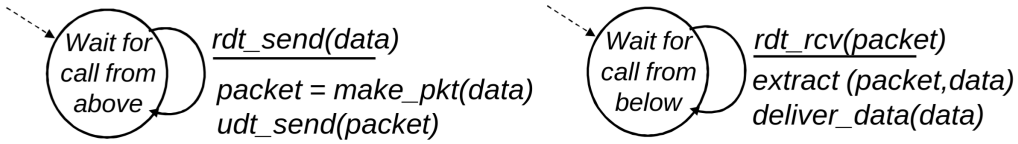
Iniziamo ad iterare sul nostro protocollo *RDT*. Assumiamo come prime ipotesi, largamente semplificate:

- Non ci sono errori di bit;
- Non ci sono perdite di frame.

Prevederemo quindi FSM diverse per *trasmettitore* e *ricevitore*:

- Il trasmettitore invia dati su un canale sottostante;
- Il ricevitore legge i dat dal canale sottostante.

L'FSM in questo caso sarà semplice:



- Il trasmettitore avrà il compito di aspettare la chiamata dall'alto, e una volta arrivata di creare un *pacchetto* da spedire sulla linea inaffidabile (incapsulandolo in un opportuno frame);
- Il ricevitore avrà il compito di aspettare anch'esso la chiamata dall'alto, e una volta ricevuta mettersi in ascolto per il pacchetto spedito dal trasmettitore. Una volta arrivato, dovrà estrarre i dati dal pacchetto e consegnarli al livello superiore.

10.5 RDT 2.0

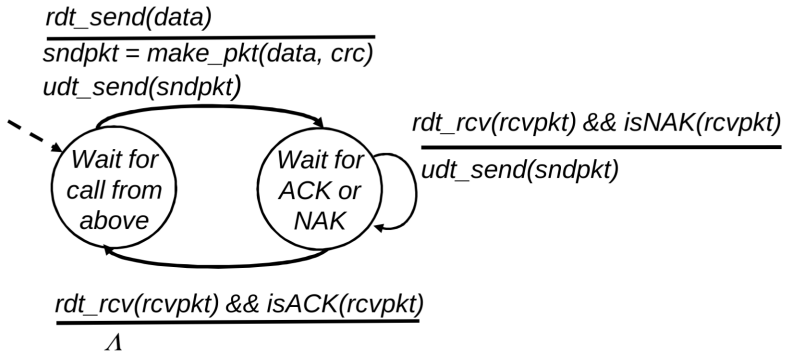
Introduciamo un mezzo di trasmissione inaffidabile che potrebbe avere errori di bit, e presumiamo che tale mezzo implementi un qualche livello sottostante (di tipo data link) che implementa correzione degli errori attraverso CRC o checksum (come visto in 10.1 e 10.2).

Il problema sarà: come riprendersi dagli errori? Abbiamo due modi principali:

- *Acknowledgment* (**ACK**), significa che il ricevitore ha *"capito"*, cioè ha ricevuto il frame correttamente;
- *Negative acknowledgment* (**NAK**), significa che il ricevitor *non ha capito*, cioè non ha ricevuto il frame correttamente.

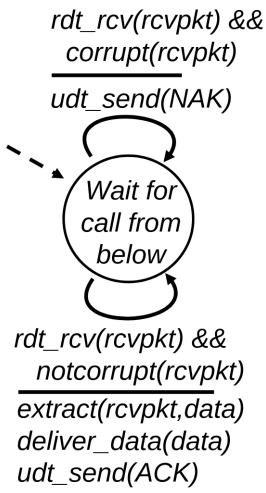
Quando il trasmettitore riceve un **ACK**, sa di poter procedere con il frame successivo, mentre quando riceve un **NAK** sa che deve reinviare il frame corrente.

In questo caso l'FSM del trasmettitore ha il seguente aspetto:



- Il trasmettitore avrà il compito, come prima, di aspettare la chiamata dall'alto e quindi inviare il pacchetto sulla linea inaffidabile;
- A questo punto dovrà aspettare per un ACK o un NAK dal ricevitore:
  - Se riceve un NAK, deve reinviare lo stesso frame;
  - Altrimenti, cioè se riceve un ACK, deve tornare ad aspettare la chiamata dall'alto per il pacchetto successivo.

Lato ricevitore l'FSM sarà invece:



- Il ricevitore dovrà restare in attesa della chiamata dall'alto, e rispondere quindi ai pacchetti ricevuti su linea inaffidabile dal trasmettitore;
- Valutata la correzione degli errori sul pacchetto ricevuto, dovrà:
  - Inviare il NAK nel caso il pacchetto sia corrotto;
  - Inviare l'ACK e inoltrare il pacchetto al livello superiore nel caso il pacchetto sia stato ricevuto correttamente.

11 Lezione del 15-10-25

11.0.1 RDT 2.1

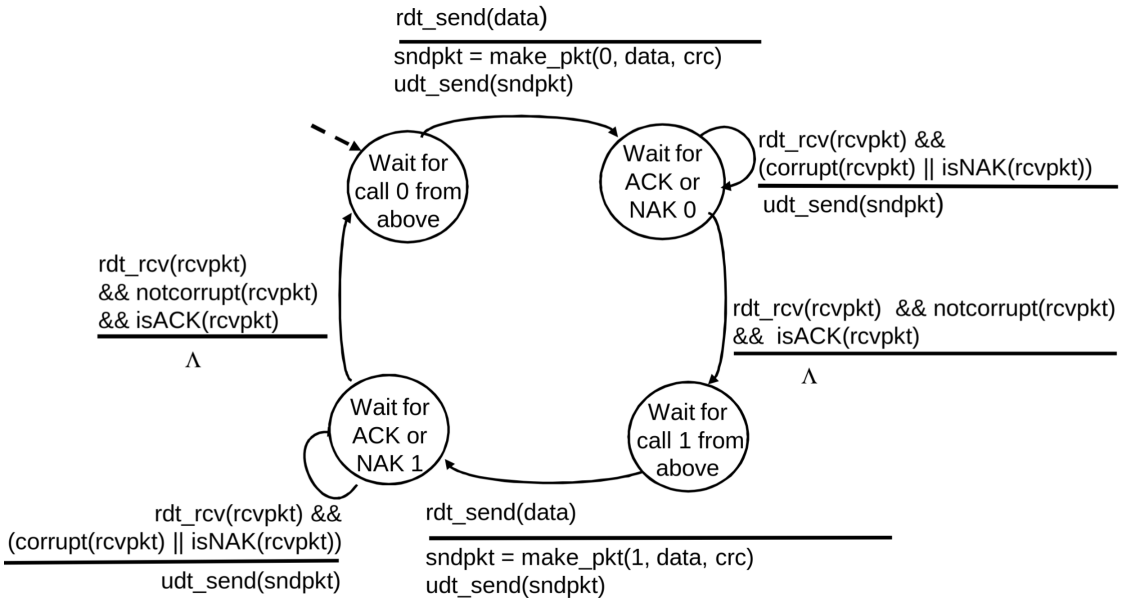
Avevamo introdotto il protocollo di trasferimento fittizio RDT 2.0. Questo era basato su segnali ACK/NAK mandati sul livello datalink dal destinatario al mittente.

Un problema che non avevamo considerato è che questi segnali possono venire corrotti, proprio come i dati veri e propri che volevamo trasmettere. Se il mittente riceve un ACK/NAK corrotto, non può semplicemente ritrasmettere: il destinatario potrebbe ricevere frame duplicati.

Per risolvere questo problema introduciamo fra le *informazioni di controllo* dei frami inviati un cosiddetto **numero di sequenza**. Il mittente allega ad ogni frame un numero di sequenza, e il destinatario implementa un *contatore* del numero di sequenza. I frame con numero di sequenza già visto vengono ignorati.

Abbiamo che per un protocollo *stop-and-wait*, un contatore a 1 bit è più che abbastanza: si inviano due pacchetti alla volta.

In questo caso la macchina a stati del trasmettitore sarà la seguente:

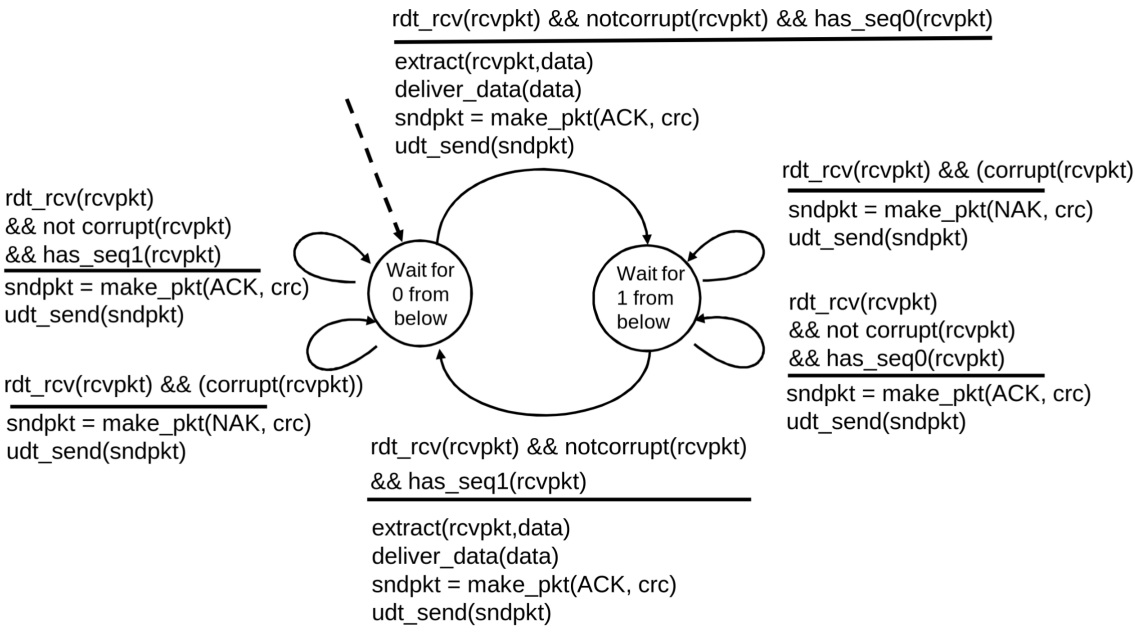


Quello che facciamo è:

- Aspettare la chiamata dall'alto ed inviare il primo pacchetto (con numero di sequenza 0);
- Continuare ad inviare il pacchetto finche non si riceve un ACK non corrotto;
- Aspettare la chiamata dall'alto ed inviare il secondo pacchetto (con numero di sequenza 1);
- Di nuovo, continuare ad inviare il pacchetto finche non si riceve un ACK non corrotto.

Questo andamento si ripete in maniera ciclica.

Il ricevitore potrà a questo punto eseguire il seguente automa:



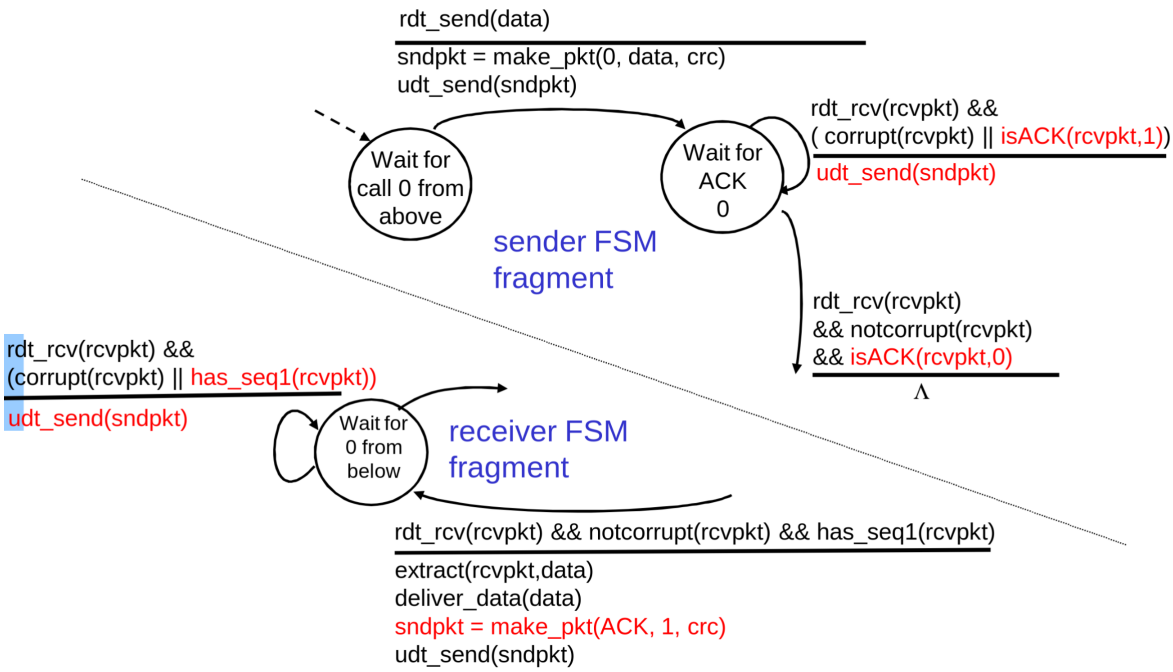
In questo caso il flusso è:

- Si aspetta per il primo pacchetto (dopo la chiamata dall'alto), si invia un pacchetto ACK nel caso questo arrivi integro e con numero di sequenza 0, NAK altrimenti;
- Si aspetta il secondo pacchetto, comportandoci in maniera analoga, ma verificando che il numero di sequenza sia 1.

11.0.2 RDT 2.2

Si può semplificare l'approccio sopra riportato guardando semplicemente agli ACK e ai numeri di sequenza: in questo caso il ricevitore deve solo inviare ACK per gli ultimi pacchetti ottenuti integri. ACK per pacchetti già trasmessi verrano interpretati dal trasmettitore come avevamo interpretato i NAK fino ad ora.

In questo caso le macchine a stati si modificano come segue:



11.1 RDT 3.0

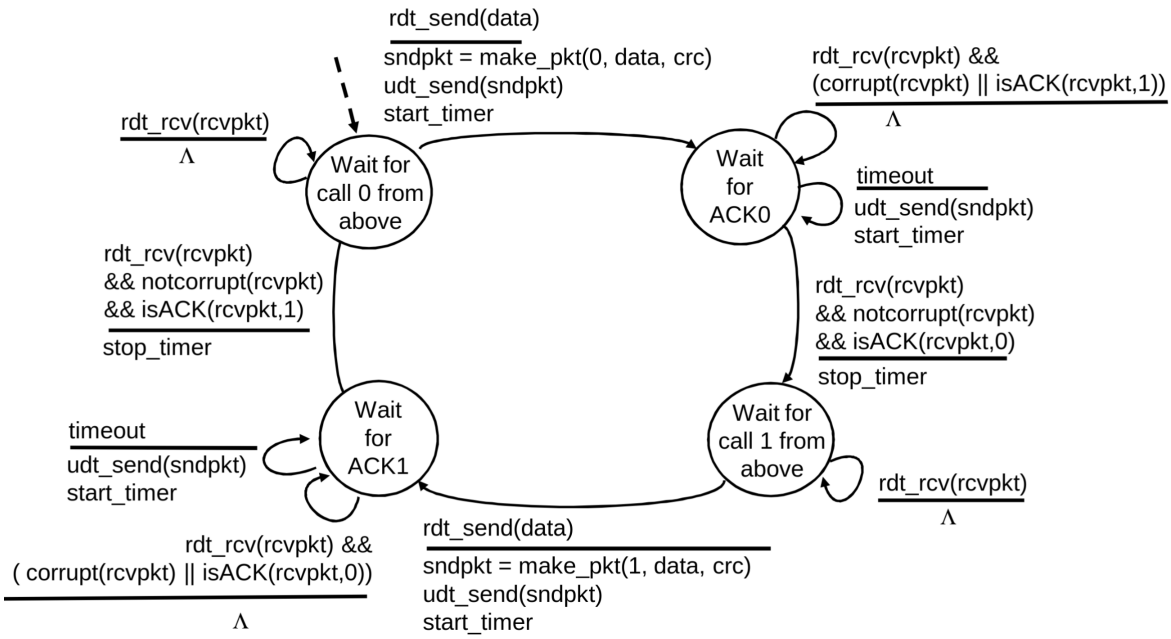
Facciamo una nuova assunzione sui canali: diciamo che il canale sottostante può anche *perdere* (sia data che ACK), mentre prima poteva solo avere errori.

In questo caso potremo sfruttare checksum, numeri di sequenza, ACK e ritrasmissioni, ma non basterà completamente a risolvere il problema.

Un primo approccio potrebbe essere, lato tramsettitore, di aspettare un tempo ragionevole per l'arrivo dell'ACK da parte del ricevitore. *"Ragionevole"* può avere molti significati: assumiamo che il tempo sia tarato sul round-trip-time del link corrente. Dopo che questo tempo passa, quindi, si procede col reinvio del pacchetto:

- Se il pacchetto non era arrivato, il problema è risolto;
- Se il pacchetto era già arrivato e si era perso l'ACK, il ricevitore riceverà pacchetti duplicati: i numeri di sequenza risolvevano già questo prolema. Chiaramente, il ricevitore dovrà specificare il numero di sequenza del pacchetto per cui sta facendo ACK.

La macchina a stati del trasmettitore con questo approccio è:



Ciò che facciamo è analogo a RDT 2.1, con la differenza che:

- Quando inviamo il pacchetto, facciamo partire un timer;
- Se il timer fa timeout durante la fase di attesa per l'ACK, si reinvia il pacchetto al numero di sequenza corrente e si fa ripartire il timer.

Notiamo come si adotta un'approccio *lazy* ai pacchetti ACK corrotti o per pacchetti con numero di sequenza sbagliato (quello che interpretavamo come NAK): in questo caso si ignorano e ci si affida al timeout (che prima o poi arriverà) per effettuare il reinvio.

Lato ricevitore, nulla cambia rispetto a RDT 2.2.

11.1.1 Prestazioni di RDT 3.0

Facciamo alcune note sulle **prestazioni** che riusciamo ad ottenere.

Sia U_{sender} l'**utilizzazione tramsettitore**, cioè la frazione di tempo sul RTT usata dal trasmettitore in fase di invio effettivo del pacchetto.

Se assumiamo di avere un link da 1 Gbps, 15 ms di ritardo di propagazione e di dover trasmettere un pacchetto da 8000 bit. Il tempo per trasmettere un pacchetto sul canale sarà allora circa:

$$D_{\text{trans}} = \frac{L}{R} = \frac{8000}{1000} = 8\,\text{ms}$$

In condizioni ideali, quindi, avremo che dovremo inviare un pacchetto e aspettare un ACK (spedito in tempo trascurabile): questi sono due viaggi sul link (quindi un RTT dato dal due volte il ritardo di propagazione) e il tempo di trasmissione del pacchetto vero e proprio. Si ha quindi:

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.0008}{30 + 0.0008} = 0.00027$$

Cioè l'utilizzazione è terribile: del tempo che impieghiamo a trasmettere un pacchetto, meno dell'0.001% è effettivamente usato per trasmettere i dati che ci interessano! Il problema è chiaramente che passiamo molto tempo ad *aspettare*, quando potremmo avvantaggiarci con l'invio dei pacchetti successivi.

11.2 Pipelining

Il **pipelining** è un meccanismo attraverso il quale il trasmettitore può inviare più pacchetti in massa, senza aver ancora ricevuto l'ACK. Lo poniamo come alternativa allo *stop-and-wait* visto finora.

Riprendendo i conti della scorsa sezione, prendiamo ad esempio di inviare 3 pacchetti invece di 1 per ciclo di ACK. In questo caso si avrà:

$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{0.0024}{30 + 0.0008} = 0.00081$$

che non è molto ma è già di più.

Chiaramente questo comporta delle complicazioni lato ricevitore: dovremmo prevedere la bufferizzazione dei pacchetti in entrata, e ingrandire il contatore del numero di sequenza (2 valori non basteranno più).

Trascurando quanto avviene lato ricevitore, una domanda interessante è *quanti* pacchetti possiamo anticipare prima dell'ACK successivo: dovremmo riempire la finestra del RTT con più tramissioni (ciascuna occupante tempo $\frac{L}{R}$), cioè inviare un numero di pacchetti al massimo pari a:

$$N = \frac{RTT}{L/R}$$

Approcci di questo tipo vengono detti a **sliding window** (*"finestra scorrevole"*), in quanto prendono buffer scorrevoli sul blocco di pacchetti da inviare.

11.2.1 Recupero errori in pipelining

Abbiamo visto che, se vogliamo implementare un protocollo in pipelining, dobbiamo adoperare alcune soluzioni tecniche:

- Buffering al trasmettitore dei pacchetti da inviare;
- Buffering al ricevitore dei pacchetti da ricevere, senza nessuna assicurazione che questi vengano ricevuti in ordine (serve il numero di sequenza);
- Un contatore per il numero di sequenza più grande al ricevitore.

Le specifiche di queste soluzioni vengono definite dal protocollo per il recupero dagli errori che adottiamo. Ne vederemo i 2 principali, che sono **go-back-N** e **selective-repeat**.

11.2.2 Go-back-N

In questo caso il trasmettitore mette fino a N pacchetti senza ACK in pipeline. Al ricevitore viene permesso solo di inviare ACK cumulativi: non effettua ACK se perde uno degli N pacchetti. Il trasmettitore mantiene quindi un timeout per gli ultimi N pacchetti inviati: se il timer scade reinvia tutti gli N pacchetti.

Lato implementativo si ha quindi che il trasmettitore spedisce fino a N pacchetti consecutivi sulla pipeline. Il ricevitore invia da parte sua ACK consecutivi al pacchetto con numero di sequenza più alto ricevuto fino a quel momento (se bufferizzare o

meno i pacchetti che non combaciano col numero di sequenza aspettato è una scelta implementativa).

Quando il primo pacchetto della sequenza spedita viene ricevuto, il trasmettitore sposta in avanti la sua finestra e spedisce il successivo, spostandosi così lungo il blocco di pacchetti.

Se un pacchetto viene perso, il trasmettitore aspetta senza spostare la finestra: prima o poi scatterà il timeout del pacchetto perso e verrà reinviato. Lato ricevitore, possiamo aspettarci che questo passerà lo stesso intervallo temporale a inviare ACK sempre sull'ultimo pacchetto valido (con numero di sequenza consecutivo) ricevuto, e che il trasmettitore abbia ignorato tale ACK in quanto ascolta solo l'ACK più grande ricevuto.

11.2.3 Selective repeat

In questo caso il trasmettitore invia N pacchetti senza ACK in pipeline (come sopra). La differenza è che il ricevitore invia ACK per ogni pacchetto: il trasmettitore deve quindi mantenere un timer per ogni pacchetto inviato, e nel caso questo timer faccia timeout inviare solo il pacchetto corrispondente.

Questo approccio è chiaramente più efficiente: si ritrasmettono solo i pacchetti persi, a costo di un'implementazione più costosa sia lato trasmettitore (dobbiamo mantenere più timer) che ricevitore (dobbiamo inviare ACK separati per ogni pacchetto).

Il caso in cui approcci come il go-back-N vengono comunque usati è quello di dispositivi *constraining* (come quelli che si usano nell'IoT): questi hanno prestazioni meno soddisfacenti e quindi impongono di usare protocolli meno efficienti.