Reti di Calcolatori I

appunti scritti da Vincenzo De Rosa, tratti dalle slide del prof. Riccardo Caccavale

Primo Semestre A.A. 2024-25

Contents

1 Introduzione

> Computer Networking: il processo di connettere insieme dei computer in modo tale che essi possano scambiarsi informazioni
>
> *Cambridge Dictionary*

Internet, la più importante rete del mondo, è probabilmente il sistema ingegneristico più largo mai creato: include centinaia di milioni di dispositivi di rete, link e computer offrendo centinaia di servizi agli utenti.

1.1 Componenti delle reti

I dispositivi (**device**) di rete e i **link** sono infrastrutture che consentono agli host di connettersi. Gli **host** sono dispositivi su cui le applicazioni o programmi vengono eseguiti.

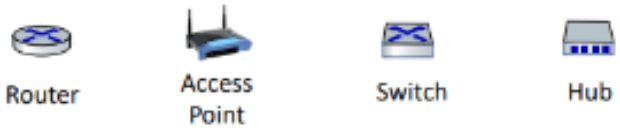


Figure 1: Device di rete

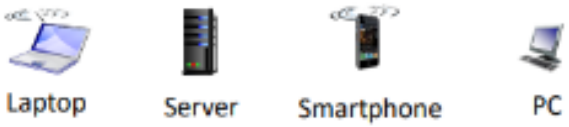


Figure 2: Host

Vengono utilizzate terminologie della teoria dei grafi, tra le quali:

- I dispositivi connessi tramite la rete sono chiamati **nodi**, mentre le connessioni tra i nodi sono chiamate **link** (o **channel**)
- Una sequenza di nodi/link è detta percorso (**path**)
- Gli end-point della rete, che forniscono o usano servizi, sono nodi speciali detti **host**

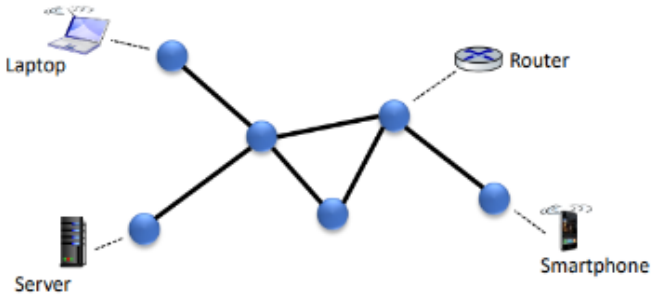


Figure 3: Rappresentazione a grafo di una rete

1.2 Data Communication

L'obiettivo delle reti è di garantire lo scambio di informazioni. La **Data communication** è il processo di scambio di dati tra dispositivi mediante una qualche forma di mezzo di trasmissione (e.g. wired o wireless). Solitamente è un compromesso tra:

- **Reliability**: il dato è ricevuto correttamente (affidabilità)
- **Performance**: il dato è ricevuto in un intervallo ragionevole di tempo

La **telecomunicazione** è il campo che studia come comunicare informazioni a distanza.
La Data communication è formata di base da 5 componenti:

1. **Messaggio**: contiene l'informazione che si intende comunicare
2. **Mittente**: l'entità che invia il messaggio
3. **Ricevente**: l'entità che dovrebbe ricevere il mesaggio
4. **Medium**: il canale interposto tra mittente e ricevente dove il messaggio viaggia
5. **Protocollo**: un insieme di regole, note a mittente e ricevente, usate per gestire il messaggio.

L'informazione da comunicare può essere rappresentata in diverse forme (testo, numeri, immagini, audio, video, etc.)
In base al tipo e allo scopo della comunicazione il **data flow** (flusso dei dati) può essere:

- **Simplex**: monodirezionale
- **Half-Duplex**: bidirezionale (a turni)
- **Full-Duplex**: bidirezionale (in modo simultaneo)

Alcune grandezze per le misurazioni

- La **trasmission rate** rappresenta la massima quantità di informazioni che un dispositivo può trasmettere, misurata in bit/sec (o byte/sec)
- La **bandwidth** (larghezza di banda) è la massima quantità di informazioni che un path può trasmettere, misurata in bit/sec (o byte/sec)
- Il **troughput** è la quantità attuale (istantanea) di informazioni che è trasmessa su un path, misurata in bit/sec (o byte/sec)

1.3 Tipi di connessione

Gli host di una rete possono essere connessi in **Point-to-Point**, dove c'è un link dedicato tra due device, oppure in **Multipoint** (broadcast) dove più di due device condividono un singolo link. Entrambe possono essere wired o wireless.

1.4 Topologie di rete

Una **topologia** di rete è la disposizione fisica delle componenti di una rete. Di seguito vedremo quelle fondamentali:

- **Bus**: gli host sono connessi ad un cavo centrale detto **backbone**. I messaggi inviati da 2 host generano collisioni. Le reti bus hanno **1** duplex link e **n** link (dove n è il numero di host). **Pro:** semplice e economica, ideale per piccole reti; **Contro:** Single point of failure nel bus, in caso di rottura parte o la totalità della rete non è più funzionante.

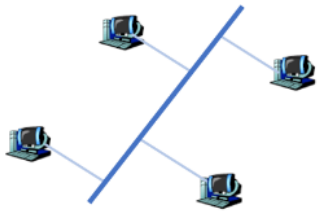


Figure 4: Topologia bus

- **Ring**: ogni host è connesso ad altri 2 mediante connessioni point-to-point bidirezionali. Il segnale è inoltrato lungo l'anello, device per device, finché non raggiunge la destinazione. In totale ci sono **n** duplex link.
  Pro: semplice ed economica, con prestazioni migliori del bus. **Contro:** l'aggiunta di nuovi nodi è più difficile ed il malfunzionamento di alcuni nodi potrebbero causare problemi alla rete.

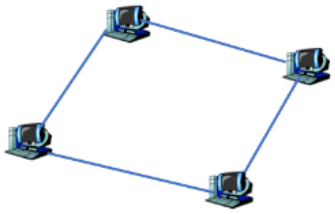


Figure 5: Topologia Ring

- **Star:** gli host sono connessi ad un controller centrale (hub,switch o router) che gestisce i messaggi. **1** controller e **n** duplex link.
  Pro: Meno costosa, semplice, robusta e facilmente scalabile. **Contro:** il controller deve essere raggiungibile da tutti gli host. Single point of failure nel controller.

Figure 6: Topologia Stella

- **Tree**: più topologie a stella integrate tramite un cavo bus.
  Pro: versatile, scalabile, robusto. Ben supportata da fornitori hardware e software. **Contro:** difficile da configurare e debolezza del bus.

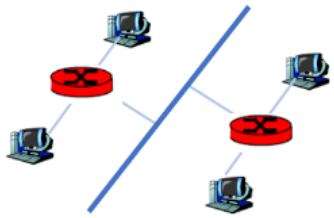


Figure 7: Topologia Tree

- **Mesh:** gli host sono connessi point-to-point in modo non gerarchico. Si parla di **Full mesh** se tutti i nodi sono connessi con tutti gli altri, **Partial mesh** se i nodi sono connessi con alcuni nodi. Nel caso della full mesh sono presenti $\frac{n(n-1)}{2}$ link duplex!
  Pro: poco traffico, robusta, sicura. **Contro:** difficilmente scalabile e costosa dato che i device devono avere molte porte.

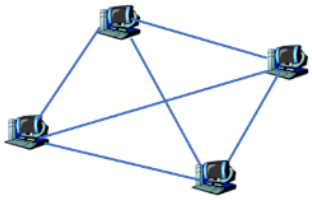


Figure 8: Topologia Full Mesh

Ovviamente le diverse topologie possono essere mescolate in una rete ibrida

1.5 Categorie di rete

Le reti vengono categorizzate in base alle dimensioni, il numero di host e la bandwidth: **Local Area Network** (LAN), **Metropolitan Area Network** (MAN) e **Wide Area Network** (WAN).
La complessità delle topologie può crescere con l'incrementare delle dimensioni della rete. Le WAN che connettono nazioni o continenti possono ovviamente essere estremamente complicate ed eterogenee.

1.6 Service Providers

Il ruolo principale delle reti è consentire lo scambio di dati tra dispositivi da differenti locazioni, ma i dati sono solo un mezzo per raggiungere risorse e servizi forniti dalla rete. Le **risorse** di rete sono "oggetti" remoti a cui vogliamo accedere (pagine web, video o audio, files ecc.). Un **servizio** di rete è un' "azione" che un dispositivo remoto svolge per noi. La differenza tra le due definizioni è spesso sfumata e ambigua.
Le entità che offrono servizi su internet sono chiamati **Service Providers**. Internet è uno dei servizi di base, una **Internet Service Provider** (ISP) è un'organizzazione che fornisce servizi per l'accesso e l'utilizzo di Internet. Differenti ISP scambiano dati attraverso i **Network Access Points** o gli **Internet Exchange Points** che funzionano come una zona neutrale tra ISP, tipicamente gestite da terze parti.

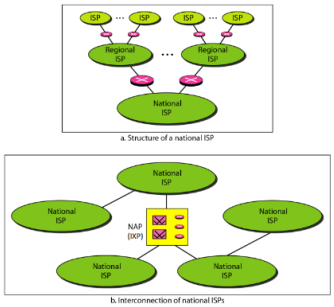


Figure 9: Internet Service Provider

Una rete ISP è gerarchicamente definita:

- Un **Point of Presence**(PoP) è un gruppo di un o più router usati dalle ISP per raggiungere i consumatori.
- **Access** ISP per aree locali.
- ISP **Regionali** per ampie aree.
- ISP **Nazionali** per le nazioni.

1.7 Connessione ad Internet

Le connessioni private ad internet sono stabilite via **Modem** abilitate dalla rete telefonica, mentre le aziende (in particolar modo quelle medio/grandi) possono avere una connessione dedicata all'ISP.

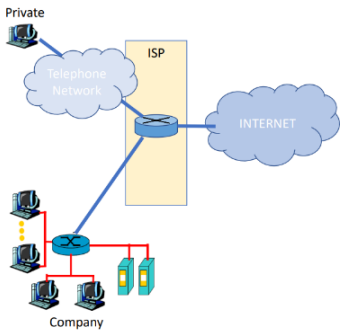


Figure 10: Connessione ad Internet

Un esempio interessante di accesso ad internet è quello delle università italiane gestito dal **GARR** (Gruppo per l'Armonizzazione delle Reti della Ricerca) che è la rete nazionale per le università e i centri di ricerca. La rete GARR è connessa con altre reti nazionali in Europa e nel mondo, costituendo una parte integrale di Internet.

1.8 Condivisione di servizi e risorse

L'obiettivo principale delle reti è quello di condividere servizi e risorse tra differenti dispositivi. Essi possono essere privati e disponibili localmente, accessibili solo a chi è connesso nella rete locale. Tuttavia la maggior parte dei servizi odierni sono pubblici e forniti su internet dai Service Provider. Esistono diverse modalità per fornire servizi:

- Le risorse e i servizi possono essere pubblici e fornite da server distanti. Un **server** è un host speciale pensato per fornire uno o più servizi. Ad esempio per usufruire di una pagina web, il corrispettivo server potrebbe essere dall'altra parte del mondo ma noi saremmo ignari di tale distanza. Allo stesso modo potrebbero esserci un cluster di server che cooperano per fornire servizi senza che l'utente finale ne sia a conoscenza.
- Il **grid computing** è un approccio decentralizzato per la condivisione di risorse hardware (potenza di calcolo e ram). Tutti gli host connessi alla rete condividono una parte o la totalità delle proprie risorse per raggiungere obiettivi comuni come un calcolo estremamente complesso o salvare grandi quantità di dati. Un esempio famoso è il **SETI**, un gruppo per la ricerca di intelligenza extraterrestre, che attraverso il progetto **SETI@home** consente a diversi PC sparsi per il mondo di condividere risorse per analizzare segnali radio dallo spazio in cerca di intelligenza aliena.
- Il **cloud computing** è un'architettura centralizzata dove le risorse, tipicamente gestite da service provider, sono offerte on-demand come servizi a host esterni. Esempi di tali servizi possono essere: applicazioni per la produttività, giochi, social networks, storage, database condivisi, macchine virtuali, server ecc. Il **NIST** fornisce una definizione di cloud computing molto diffusa: è un modello che permette un accesso a risorse condivise in modo conveniente e on-demand che possono essere rapidamente fornite e rilasciate senza il minimo sforzo o l'interazione col service provider. Tale modello si basa su 5 caratteristiche essenziali:

  1. On-demand self-service
  2. Ampio accesso alla rete
  3. Condivisione delle risorse
  4. Rapida elasticità
  5. Servizio misurato

Il NIST ha suddiviso i servizi forniti da un'architettura cloud computing in 3 categorie ,(**service models**), in base a quanto dell'infrastruttura di calcolo è gestita dal service provider. Le 3 categorie sono:

  - **Software as a Service**: l'intera applicazione è gestita dal service provider.
  - **Platform as a Service** le piattaforme di lavoro sono fornite con specifiche configurazioni.
  - **Infrastructure as a Service**: la gran parte dell'hardware viene fornito al consumatore.

- L'**Internet of Things** è l'approccio di dotare semplici dispositivi con sensori, unità di calcolo e di connettività in modo da poterli controllare e monitorare da remoto con Internet.

1.9 Standard e Protocolli

Per poter offrire tutti questi servizi c'è bisogno di un insieme di regole e linee guida da seguire. Infatti sin dagli albori di internet lo sforzo maggiore è stato quello di stabilire **protocolli** e **standard** per regolare la comunicazione. Per permettere la comunicazione sia ricevente che mittente devono accordarsi su un protocollo; se più dispositivi comunicano con un protocollo comune esso diventa uno **standard**. Ci sono centinaia di protocolli che regolano ogni aspetto della comunicazione, dalle connessioni fisiche fino alle applicazioni. Gli standard odierni sono definiti dall' **Internet Engineering Task Force IETF** organizzato in gruppi di lavoro aperti, ognuno dei quali concentrato su uno specifico aspetto di internet, producendo documenti numerati chiamati **Request For Control (RFC)** che definiscono protocolli, concetti e metodi sottostanti ad uno standard

1.10 Network Models

Un approccio ragionevole per far fronte alle diverse problematiche di comunicazione è definire un design a livelli (divide et impera) dove:

- Ciascun livello è concettualmente responsabile di una task specifica
- Ciascun livello fa affidamento a servizi del livello sottostante e fornisce servizi al livello sovrastante.

Un vantaggio è quello della **modularità**, con lo svantaggio della poca **scalabilità**, infatti ogni livello aggiunge un costo computazionale.

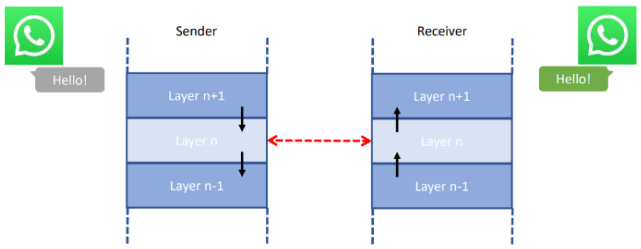


Figure 11: Network Model a livelli

1.10.1 Modello ISO/OSI

Negli anni 80 la **ISO** (International Standards Organization) definì un modello a livelli per le reti: l'**OSI**(Open System Interconnection) model. Il modello ISO/OSI contiene 7 livelli, partendo dal basso:

1. **Livello fisico**: responsabile del movimento dei bit da un nodo al successivo
2. **Livello data link**: responsabile di muovere i frame (porzioni) dei dati da un nodo all'altro
3. **Livello di rete**: responsabile della consegna dei pacchetti da l'host sorgente a quello di destinazione.
4. **Livello di trasporto**: responsabile della consegna del messaggio da un capo all'altro.

5. **Livello di sessione**: responsabile del controllo e della sincronizzazione del dialogo tra richiesta e risposta
6. **Layer di presentazione**: responsabile della rappresentazione del dato, traduzione, compressione, crittografia ecc.
7. **Livello dell'applicazione**: responsabile di fornire i servizi all'utente

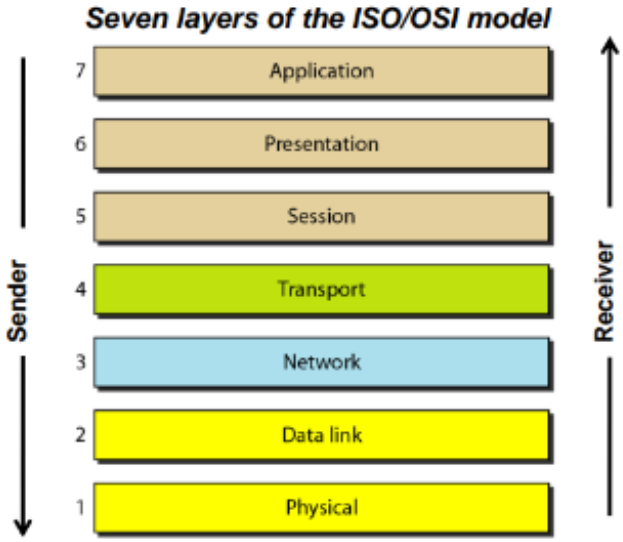


Figure 12: Modello ISO/OSI

Non sempre tutti i layer sono implementati, ad esempio i router implementano fino al 3° livello della pila. La fase di invio subisce l'**Encapsulation**, dove ogni livello del mittente aggiunge un campo specifico al messaggio in forma di header o trailer. Nella ricezione invece avviene la **Decapsulation**, dove tali campi vengono rimossi ed interpretati dal corrispettivo livello.

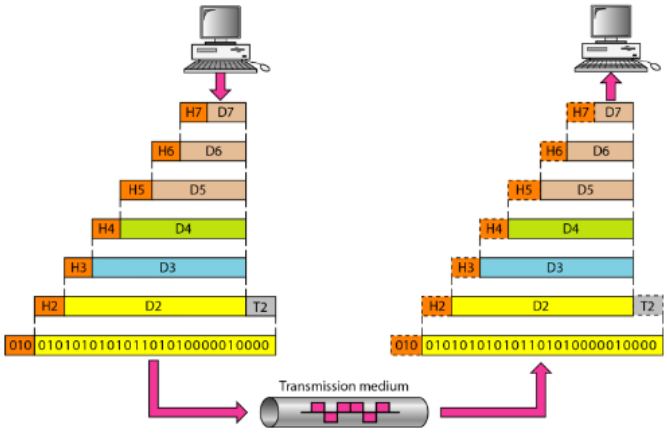


Figure 13: Encapsulation e Decapsulation

Il modello ISO/OSI è lo standard de iure per le reti, ma è complesso e dettagliato. I Protocolli TCP/IP e UDP/IP sono dei protocolli di comunicazione usati de facto su internet e nelle reti locali.

- TCP: Transmission Control Protocol
- UDP: User Datagram Protocol
- IP: Internet Protocol

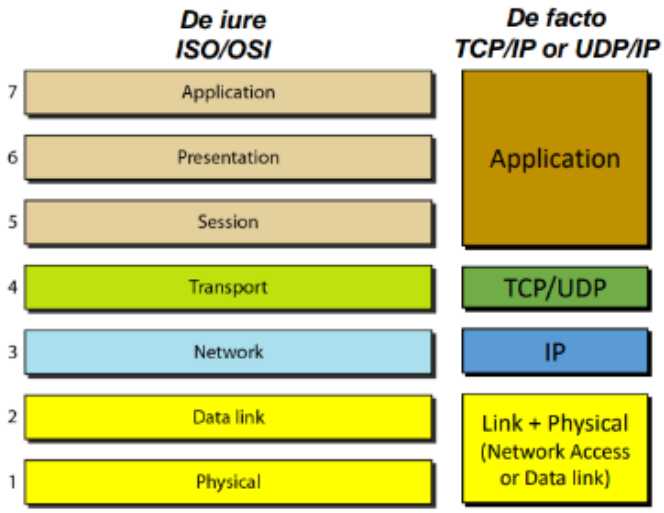


Figure 14: TCP/IP e UDP/IP models

2 Application Layer

2.1 Applicazioni di rete

Un'applicazione di rete non è altro che un programma che gira su dispositivi diversi, che possono essere nella rete locale o su host remoti. Ad esempio un applicazione web è composta da due programmi fondamentali:

- Un **programma browser** eseguito sugli host degli utenti
- Un **programma Web server** eseguito sul Web server

Quindi creare un'applicazione web significa scrivere programmi che **vengono eseguiti su differenti end systems** (che potrebbero usare diversi linguaggi o sistemi operativi) e **comunicare sulla rete** (ad esempio tramite socket) seguendo uno specifico protocollo. Non c'è bisogno però di scrivere software per dispositivi network-core dato che non eseguono applicazioni utente e che dal punto di vista dell'applicazione, la rete è idealmente una black-box (ci sono specifiche librerie che implementano le funzionalità di rete).
Una **architettura di applicazione di rete** è progettata dallo sviluppatore e dett come l'applicazione è strutturata sui vari end systems. Distinguiamo l'architettura **Client-server** e **Peer-to-peer**(P2P)

2.2 Applicazioni Client-server

Nell'architettura client-server i nodi sono eterogenei, c'è un host sempre acceso (server) che fornisce servizi che vengono richiesti da vari host (clients). In quest'architettura i client non comunicano direttamente. Invece i server hanno un **indirizzo fisso e ben noto** (hostname o indirizzo IP) in moddo tale che un client possa sempre contattarlo inviando un pacchetto (richiesta).
In un'architettura client-server multipli server sono spesso coinvolti per soddisfare le richieste dai client. Il client, dal canto suo, è ignaro di ciò e li percepisce come un unico server. I server multipli possono essere:

- Raggruppati in **Data centers** che contengono un gran numero di server in un'unica zona. Questi server devono essere alimentati, manutenuti e connessi tra di loro. Tale strategia è ottima localmente ma potrebbe causare rallentamenti se arriva una richiesta da una zona molto remota.
- Sparpagliata in **server distribuiti** per tutto il mondo. È essenziale in questo caso che tali server siano interconnessi e sincronizzati.
- Oranizzati in **data center distribuiti** (strategia ortogonale alle precedenti). Strategia molto costosa che eredita pregi e difetti delle 2 precedenti. Adottata da grandi compagnie come ad esempio Amazon e Google

Un esempio tipico è un'applicazione web dove c'è un server Web sempre attivo che riceve richieste dai browser dei client. Quando un server riceve una richiesta per un oggetto da un client risponde inviando l'oggetto richiesto al client. Il web server è sempre raggiungibile dagli host.

2.3 Applicazioni Peer-to-Peer

Spesso usate da applicazioni con un elevato traffico. Gli host sono idealmente omogenei. In quest'architettura non esiste un singolo server con un indirizzo fisso ma tutti i client hanno il loro indirizzo, perciò la comunicazione è diretta. Le applicazioni P2P pure sono molto rare, spesso sono utilizzate architetture ibride (ad esempio nelle applicazioni di messaggistica). Le architetture P2P hanno il vantaggio di essere **scalabili** e **distribuite**. Inoltre sono **economiche** ma hanno **problemi di performance, sicurezza e affidabilità**.

2.4 Comunicazione tra processi

. In un'applicazione di rete ci sono **processi** eseguiti su macchine differenti (potenzialmente con diversi sistemi operativi) che comunicano tramite la rete. Tra un paio di processi comunicanti ci sono tipicamente un **processo client** e un **processo server**.

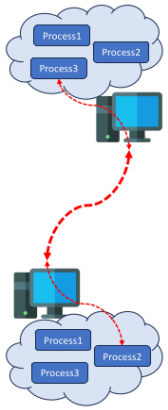


Figure 15: Processi comunicanti su rete

Quindi la maggior parte delle applicazioni di rete consistono in coppie di processi comunicanti che inviano/ricevono messaggi tra di loro attraverso la rete sottostante. Seguendo il modello a livelli, una **socket** è il software d'interfaccia tra livello applicazione e livello di trasporto che consente ai processi di inviare/ricevere messaggi sulla rete. Le socket sono usate come **API** (Application Programming Interface) tra le applicazioni e la rete. Lo sviluppatore dell'applicazione ha il pieno controllo dell'application-layer ma ha un controllo limitato sul lato del **livello di trasporto** della socket. Le uniche scelte da fare sono il protocollo di trasporto (TCP/UDP) e alcuni parametri relativi al livello di trasporto. Dato che diverse applicazioni di rete possono essere eseguite su un singolo host, per identificare il processo ricevente, c'è bisogno di 2 informazioni:

1. Un **indirizzo** dell'host
2. Un **identificativo** del processo ricevente.

L'indirizzo lavora al livello di rete mentre l'identificativo lavora al livello di trasporto.

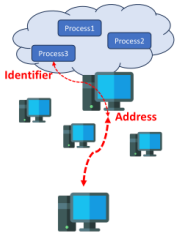


Figure 16: Identificativo e indirizzo

Nelle reti, l'host è indificato da un **indirizzo IP** mentre il processo è identificato da un **numero di porta**. Applicazioni comuni sono state assegnate per convenzione a specifici numeri di porta (ad esempio un mail server process è identificato dalla porta 25).

2.5 Quality of Service del livello di trasporto

Il livello di trasporto, sottostante a quello di applicazione offre protocolli per implementare su richiesta alcune proprietà, ovvero il **Quality of Service QoS**:

- **Reliability** (affidabilità) del trasferimento di dati: i dati inviati dall'applicazione sono consegnati correttamente e completamente all'altro capo.
- **Throughput**: indica il tasso (in bit/sec) a cui il processo mittente riesce a spedire bit al processo ricevente.
- **Timing**: tutti i bit che il mittente invia nella socket arrivano alla socket del ricevente in un determinato
- **Sicurezza**: crittografia e decifratura dei messaggi

TCP e UDP garantiscono tali proprietà in modo diverso:

- TCP include una handshake tra i processi e un trasferimento dei dati più affidabile (ma è più pesante)
- UDP è più leggero dato che non garantisce che il messaggio sia ricevuto.

2.6 Application-layer protocol

Un protocollo del livello applicazione definisce come i processi dell'applicazione di rete, eseguiti su differenti end system, scambiano messaggi ad altri processi.

- Come è strutturato il messaggio? Qual è il significato dei vari campi nel messaggio? Quando il processo invia il messaggio?

Nello specifico, il protocollo del livello applicazione definisce:

1. I **tipi** di messaggi scambiati.
2. La **sintassi** dei vari tipi di messaggio, come ad esempio i campi nel messaggio e come i messaggi sono delineati.
3. La **semantica** dei campi, come il significato dell'informazione nei campi
4. Le **regole** per determinare quando e come un processo invia messaggi e risponde ai messaggi

Ci sono diversi protocolli del livello applicazione che sono comunemente usati nelle reti, alcuni esempi:

Application	Description
DHCP	Dynamic Host Configuration Protocol, assigns IP addresses
DNS	Domain Name System, translate website names to IP addresses
HTTP/HTTPS	HyperText Transfer Protocol (Secure), transfer web pages
SMTP/SMTPS	Simple Mail Transfer Protocol (Secure), sends email messages
SNMP	Simple Network Management Protocol, manages network devices
Telnet/SSH	Teletype Network (Secure SHell), allows command-line interfacing with remote hosts
FTP/FTPS	File Transfer Protocol (Secure), used to transfer files

Figure 17: Protocolli livello applicazione

2.7 HTTP

Fino agli anni '90 Internet era usato principalmente da ricercatori, accademici e studenti universitari per accedere a host remoti (Telnet), per trasferire file (FTP) per ricevere e inviare notizie e e-mail. Nei primi anni '90 il **World Wide Web** fu la prima applicazione internet che catturò l'attenzione pubblica. Il World Wide Web è una collezione di informazioni di vario tipo che possono essere accedute su Internet seguendo uno specifico protocollo chiamato **HyperText Transfer Protocol (HTTP)**. Le comunicazioni basate sul protocollo HTTP sono tipicamente client-server:

- un programma lato client che traduce le richieste utente in messaggi HTTP. Ciò è implementato tramite i web browser
- un programma lato server che esegue le richieste HTTP e restituisce le risposte HTTP.

Il protocollo HTTP definisce la **struttura del messaggio** e come client e server scambiano tale messaggio. Il web e i suoi protocolli funzionano come una piattaforma per le applicazioni web based come Youtube, Gmail e altre applicazioni come i social network.
Le informazioni sul web sono chiamate **risorse** o **oggetti** e sono identificate da un **Uniform Resource Locato (URL)** che è una stringa composta da:

[protocol]://[usrinfo@][host][:port][/path][?query][#fragment]

Dove:

- **protocollo**: protocollo usato durante l'accesso alla risorsa (HTTP, HTTPS, FTP, etc.)
- **usrinfo**: (opzionale) sono le informazioni dell'utente come username e password seguite da @. Ormai è deprecato per problemi di sicurezza.
- **host**: è il nome o l'IP del server.
- **porta**: (opzionale) è la porta da usare (spesso dedotta dal protocollo)
- **path**: è il percorso della risorsa nel server
- **query**: preceduta da ? e le possibili richieste specifiche.
- **fragment**: preceduto da # che identifica un elemento nella risorsa.

HTTP definisce come i client eseguono richieste di oggetti dai server web e come i server trasferiscono tali oggetti ai client. HTTP è in continua evoluzione per incontrare i requisiti del networking moderno

Figure 18: Evoluzione di HTTP nel corso degli anni

Come protocollo di trasferimento sottostante il più utilizzato è TCP (affidabile e connection-oriented) ma dal 2022, in seguito all'aggiornamento ad HTTP 3.0, è stato incentivato l'uso di UDP (in realtà una sua versione aggiornata come vedremo in seguito) che è più performante.
Il client inizialmente stabilisce una connessione col server, dopodiché i due comunicheranno tramite le loro interfacce socket.
Come già detto, storicamente HTTP usava TCP come protocollo di trasporto perché offre diversi servizi utili, primo tra tutti l'affidabilità del data transfer, infatti garantisce che le richieste/risposte HTTP arrivino intatte a destinazione.
Tuttavia TCP potrebbe rallentare la comunicazione, quindi HTTP 3.0 usa anche UDP, che è stimato essere circa il 30% più veloce di TCP. Come già detto anche UDP è stato aggiornato in modo da implementare l'affidabilità necessaria. Il protocollo risultante è chiamato **QUIC** (Quick UDP Internet Connection).

2.7.1 Statelessness

La **semplicità** è molto importante per i server HTTP che affrontano un alto numero di richieste al secondo (in media 1000, ma ad esempio l'intera infrastruttura di Google ne accoglie 100000 al secondo). Perciò le applicazioni HTTP sono tipicamente **stateless**: il server non salva nessuna informazione riguardo l'interazione con il client. Non tutte le applicazioni sono stateless, molte infatti sono dette **stateful**

2.7.2 Connessioni persistenti e non persistenti

In diverse applicazioni il client e il server possono comunicare per un periodo di tempo molto esteso. Usufruendo di un protocollo connection-oriented possiamo avere:

- **Connessione persistente**: tutte le richieste e risposte sono inviate sulla stessa connessione
- **Connessione non persistente**: per ogni coppia richiesta-risposta viene stabilita una nuova connessione

La prima versione di HTTP utilizzava connessioni non persistenti, ma questa strategia fu subito accantonata perché troppo dispendiosa in favore di connessioni persistenti.
Dato che è molto difficile stimare precisamente i tempi su Internet, possiamo stimare l'overhead necessario per finalizzare una richiesta HTML in termini di **round-trip times** RTT. Assumendo una connessione basata su TCP, per stabilire la connessione viene eseguita una **three-way handshake**. Essa è eseguita seguendo 3 step:

- Il client invia un piccolo segmento TCP al server, per segnalare che una connessione è richiesta.
- Il server la riconosce e risponde con un altro piccolo segmento TCP.
- Avviene infine un altro riconoscimento da parte del client verso il server

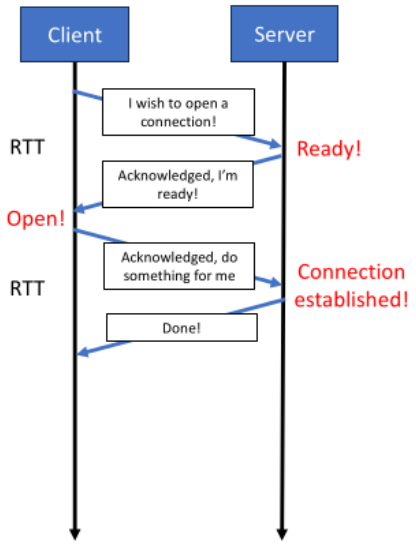


Figure 19: Funzionamento di una three-way handshake

Sono necessarie quindi 2 RTT per stabilire una connessione TCP. Da notare il fatto che un handshake simile avviene anche quando la connessione è chiusa.
Per poter ridurre il numero di RTT in una connessione non persistente il **messaggio di richiesta** viene inviato in concomitanza della terza parte del three-way handshake (il riconoscimento). Così una volta che il messaggio arriva il server invia il file HTML.
Nelle connessioni persistenti il server lascia aperto la connessione TCP dopo aver inviato una risposta. In questo modo pagine web multiple residenti sullo stesso server possono essere inviate dal server allo stesso client sulla singola connessione TCP persistente.
Un ulteriore miglioramento in termini di RTT è la strategia del **pipelining** dove richieste per gli oggetti possono essere fatte una dopo l'altra senza attendere le risposte delle richieste in corso. Dall'HTTP/2 possono essere effettuate multiple richieste e risposte possono essere interlacciate nella stessa connessione. Esistono anche meccanismi per privilegiare le richieste e risposte HTTP all'interno di questa connessione. Le connessioni persistenti con pipelining sono attualmente la **modalità default** in HTTP

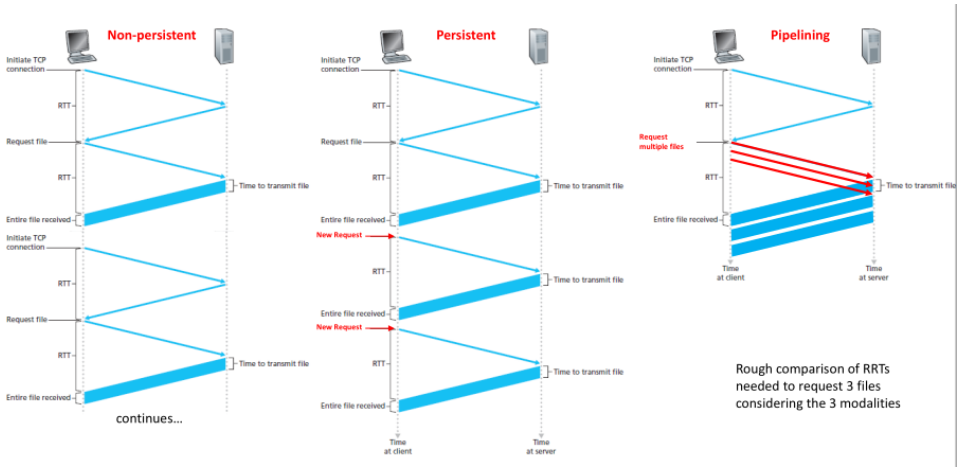


Figure 20: Confronto tra le varie modalità per la richiesta di 3 file

Quindi le connessioni persistenti hanno come pro l'essere **più veloci** (specialmente usando il pipelining), c'è bisogno di **meno memoria**. D'altro canto sono **più complesse** da implementare. Tipicamente un server HTTP chiude questo tipo di connessione quando non è usata per un certo periodo di tempo (timeout).
Le connessioni non persistenti invece sono **più semplici** da implementare e si adattano molto bene alla statelessness di HTTP. Necessitano però di **più risorse** dato che per ogni connessione i buffer TCP devono essere allocati e le variabili TCP devono essere presenti sia nel client che nel server. Inoltre sono **più lente** richiedendo 1-2 RTT aggiuntivi per ogni richiesta.

2.7.3 Formato del messaggio HTTP

In HTTP abbiamo due diversi formati per il messaggio di richiesta e per quello risposta. Il messaggio di **richiesta** specifica il comando (method) che il server HTTP deve eseguire (per esempio: dammi una pagina web). Il messaggio di **risposta** restituisce l'output del comando e la possibile informazione (ad esempio la pagina web richiesta). Entrambi i messaggi sono scritti in codice ASCII ordinario, in modo da essere facilmente leggibili dall'uomo.

2.7.4 Formato del messaggio di richiesta

Il formato del messaggio di richiesta comprende i seguenti campi:

- Il **metodo**: specifica il comando richiesto che deve eseguire il server.
- L'**URL**: usato per identificare l'oggetto su cui si vuole operare
- La **Versione**: specifica la versione di HTTP
- Le **header lines**: contengono i parametri della richiesta. Il numero e il tipo di queste linee non sono fissati. Ogni linea include il nome e il valore del parametro. Ad esempio qui possiamo specificare se vogliamo usare una connessione persistente o non persistente
- Il **body**: è specifico per il metodo e contiene dati che sono potenzialmente associati al comando.

I campi metodo, URL e versione formano la **request line** e sono separati dai seguenti caratteri speciali: sp (spazio), cr (carriage return \r) e lf (line feed \n)

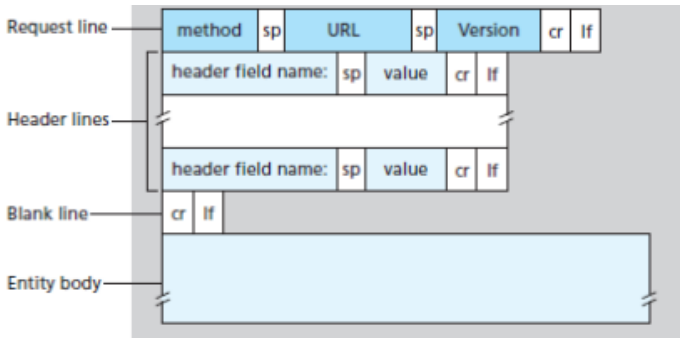


Figure 21: Formato del messaggio di richiesta

Alcuni metodi usati nelle richieste HTTP sono:

- **GET**: metodo usato per recuperare oggetti (risorse) dal server. È uno dei metodi più usati dato che ogni volta che navighiamo in una nuova pagina web il file associato deve essere recuperato. Per questo metodo il body è vuoto.
- **POST**: metodo usato per impostare informazioni all'interno di una risorsa del server. Il body contiene le informazioni da postare.
- **HEAD**: metodo simile al metodo GET solo che quando un server riceve una richiesta col metodo HEAD, esso risponde con un HTTP message, ma l'oggetto non viene restituito. Spesso usato dagli sviluppatori per il debugging.
- **PUT**: metodo che consente ad un utente di caricare un oggetto ad uno specifico path su uno specifico server Web.
- **DELETE**: metodo che consente ad un utente di eliminare un oggetto da un server Web.

2.7.5 Formato del messaggio di risposta

Simile al formato di quello di richiesta ma con la differenza che al posto della request line, è presente la **status line** che riporta l'output del comando contenente:

- La **versione**: indica la versione di HTTP della risposta del server.
- Lo **status code**: un codice che specifica l'output del comando.
- La **frase**: contiene il risultato della richiesta.

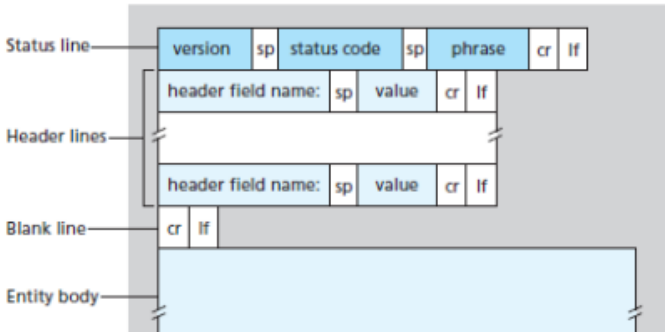


Figure 22: Formato del messaggio di risposta

Gli **status code** sono divisi in classi:

- Da 100 a 199 **informativi**.
- Da 200 a 299 **successo**.
- Da 300 a 399 **redirection**, ci sono azioni aggiuntive necessarie da parte del client.
- Da 400 a 499 **client-error**.
- Da 500 a 599 **server-error**.

2.7.6 Cookies

Un server HTTP come già detto è tipicamente **stateless**, ciò semplifica la progettazione del server riducendo l'uso di risorse e consentendo ai server di gestire migliaia di connessioni TCP contemporaneamente. Una statelessness è anch'essa una forte limitazione dato che diverse funzionalità web sono **client-specific** (ad esempio il catalogo Amazon è "addestrato" in base alle preferenze del cliente). A tal proposito il protocollo HTTP usa i **cookie**, ossia dei token digitali (id alfanumerici) usati ai server per identificare specifici client. I cookie permettono al sito web di tener traccia degli utenti.
Un cookie è creato dal server e consegnato al client. I cookie coinvolgono 4 principali componenti:

1. Una **"Set-cookie" header line** nel messaggio di risposta HTTP
2. Un **"Cookie" header line** nel messaggio di richiesta HTTP.
3. Un **file** sul client system (gestito dal browser dell'utente).
4. Un **back-end database** sul server.

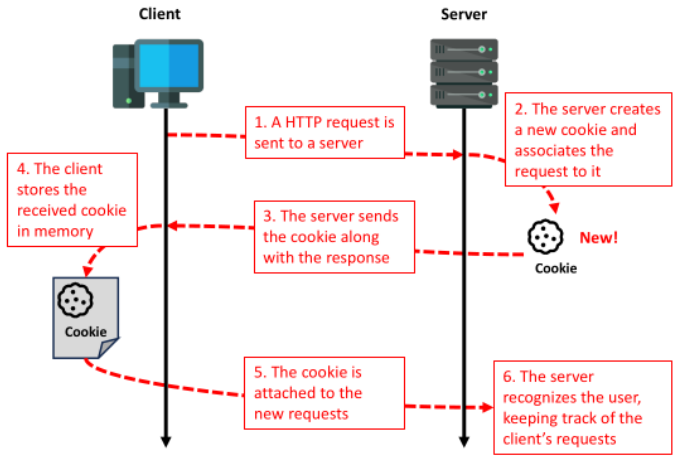


Figure 23: Creazione di un cookie

Un sito come Amazon può usare i cookies per fornire diversi servizi:

- **Registrazione dell'utente** associando le informazioni dell'utente al cookie nel database in modo tale da non doverle re-inserire ogni volta.
- **Suggerimenti di prodotti**, basate sulle pagine web visitate.

Benché spesso i cookie semplifichino l'esperienza su Internet dell'utente, essi sono ambigui perché possono essere considerati una violazione della privacy. Un sito web potrebbe addirittura vendere le informazioni a terzi.

2.7.7 Web caching

Una **Web cache** è un'entità di rete che soddisfa richieste HTTP per conto di un server web d'origine. Tali entità sono chiamate **server proxy**. Una web cache ha un suo storage dove sono contenute copie degli oggetti recentemente richiesti. Un browser può essere configurato in modo tale che tutte le richieste HTTP siano prime direzionate alla web cache per verificare se una copia dell'oggetto richiesto sia disponibile. Per esempio assumiamo che un browser invii una richiesta di una gif ad un server passando per una web cache (proxy):

1. Il browser **stabilisce una connessione TCP** con la web cache e invia ad essa una richiesta HTTP per la gif.
2. La web cache **verifica se ha una copia della gif** memorizzata localmente. Se così fosse, la gif viene fornita dalla web cache al client browser mediante una risposta HTTP. Altrimenti prosegue al punto 3.

3. **Se la web cache non ha l'oggetto**, apre una connessione col server d'origine e invia una richiesta HTTP per la gif.

4. Quando la web cache **riceve la gif, la memorizza e ne invia una copia**, in un messaggio di risposta HTTP, al client.

Da notare che una cache è sia un server che un client allo stesso tempo. Il web caching consente di **ridurre il tempo di risposta per una richiesta client** e di **ridurre il traffico**.
Le web cache sono **spesso installate nelle reti locali di compagnie/istituzioni** per velocizzarle e ridurre il traffico.
Un **hit** avviene quando una cache fornisce una risorsa senza contattare il server d'origine. L'**hit rate**, ovvero la frazione di richieste soddisfatte dalla cache, tipicamente varia da 0.2 a 0.7 (tale valore aumenta quando più client usano la cache).
Il web caching introduce un nuovo problema: la copia di un oggetto residente nella cache **potrebbe essere non aggiornata**. Per evitare tale problema, HTTP ha un meccanismo che consente alla cache di verificare l'oggetto memorizzato.
Il **conditional GET** è una richiesta HTTP che include un metodo GET e un **"If-Modified-Since:" header**. La cache verifica che l'oggetto sia ancora valido e in tal caso la versione salvata localmente è restituita all'host.
Il caching è **eseguito localmente anche dai browser**. Il principio è lo stesso dei server, il browser immagazzina localmente gli oggetti in modo tale che non debbano essere ricavati dal server. È una tecnica comune nei browser moderni che **migliora drasticamente le performance**, ma che potrebbe generare errori nel caso in cui l'oggetto **non sia aggiornato**.

2.8 E-Mail e SMTP

La posta elettronica è una delle più vecchie ed importanti applicazioni di internet. È tipicamente implementata in un'architettura client-server, dove ci sono 2 tipi di host:

- **User agent:** applicazione che permette la gestione delle mail lato client
- **Mail Server:** un server che immagazzina mail e gestisce le mailbox degli utenti

Gli utenti **non si scambiano mail direttamente**. Vengono utilizzati server mail che sono più **affidabili** e **specializzati**. Il **Simple Mail Transfer Protocol** SMTP è il principale protocollo a livello applicazione che viene usato tra server mail. Consente ai server di scambiarsi mail, ma non viene utilizzato dagli user agent. Una che una mail arriva al server viene memorizzata nella mailbox dell'utente, in attesa di essere scaricata in locale dall'user agent. Nel protocollo SMTP, nonostante la comunicazione avvenga tra server, esiste un **client-side** (mittente) e un **server-side** (ricevente) che viene eseguito sui server mail, entrambi usando il protocollo di trasporto TCP.

Esempio Ipotizziamo di avere un host A che invia un'e-mail ad un host B, vengono coinvolti 4 elementi:

- User Agent di A
- Server Mail di A
- User Agent di B
- Server Mail di B

I passi eseguiti sono i seguenti:

1. A **invoca il suo user agent** per un'email, fornendo l'indirizzo mail di B, compone il messaggio e invia il messaggio

2. Lo user agent di A **invia il messaggio al server mail di A** dove viene posizionato in una coda di messaggi

3. Sul server di A viene eseguito il client-side di SMTP che vede il messaggio nella coda di messaggi. **Apre quindi una connessione TCP verso un server SMTP** (sulla porta 25), eseguito sul server mail di B.

4. Dopo alcuni handshake iniziali, il client SMTP **invia il messaggio** di A **nella connessione TCP**.
5. Sul server di B, il server-side di SMTP riceve il messaggio. A questo punto il server posiziona il messaggio nella mailbox di B
6. B **invoca il suo user agent** per leggere il messaggio quando vuole.

Se il server ricevente è down, il client riprova nuovamente. Quando la connessione è stabilita avviene la fase di handshaking dove client e server si scambiano alcune informazioni come l'indirizzo e-mail del mittente e quello del ricevente. Il protocollo SMTP **sfrutta l'affidabilità di TCP** per inviare messaggi senza errori.

2.9 Accesso alle mail

L'utilizzo di server SMTP è più affidabile rispetto ad una comunicazione diretta tra utenti perché:

- I server, per definizione, sono costantemente accesi e visibili a tutti i dispositivi della rete.
- I server sono certificati.
- Il server può continuare a provare a inviare nuovamente le mail se la consegna fallisce (e ciò è computazionalmente costoso).

Per accedere alle mail dal server è necessaria un'altra applicazione client-server, ed un ulteriore protocollo. I più popolari sono **POP 3** (Post Office Protocol-Version 3), **IMAP** (Internet Mail Access Protocol) e HTTP.

2.9.1 POP3

Si tratta di un protocollo di accesso alle mail estremamente semplice. Il client apre una connessione TCP verso il server mail sulla porta 110. A questo punto POP3 progredisce attraverso 3 fasi:

1. **Autorizzazione**: tramite username e password per autenticare l'utente
2. **Transazione**: l'user agent può reperire i messaggi, marcare o rimuovere marcature per l'eliminazione dei messaggi, ottenere statistiche sulle mail.
3. **Aggiornamento**: dopo che il client esegue il comando di uscita, terminando la sessione POP3, il server mail elimina i messaggi marcati.

Quindi possono essere eseguite solo operazioni di download ed eliminazione.

2.9.2 IMAP

Permette ai server operazioni addizionali come:

- Gestire e creare cartelle (folders)
- Effettuare **ricerche** nelle cartelle attraverso criteri specifici.
- Permette agli user-agent di ottenere anche solo parti dei messaggi
- Consente a **client multipli** di essere connessi allo stesso server.

2.9.3 HTTP

Sempre più utenti inviano e accedono alle proprie mail tramite servizi Web. L'accesso basato sul web fu introdotto da Hotmail a metà degli anni '90 ed è ora l'approccio più comune usato anche da tutte le università e corporazioni. Con questo servizio lo user agent è un semplice browser web, e comunicano con le proprie mailbox tramite HTTP. Invece i server mail continuano ad affidarsi allo standard SMTP.

2.10 DNS

Come già visto ci sono due modi per identificare un host: con il suo **hostname** e con il suo **indirizzo IP**. Gli esseri umani preferiscono i più mnemonici hostname, mentre i dispositivi sulla rete preferiscono gli IP di lunghezza fissata e gerarchicamente strutturati. Il **Domain Name System** è un protocollo a livello applicazione che gestisce la traduzione da hostname a indirizzi IP. È un protocollo client-server in cui un client chiede ad un server DNS per una specifica traduzione. I server DNS sono solitamente **macchine UNIX** che eseguono il software Berkeley Internet Name Domain (BIND), usando tipicamente connessioni UDP alla porta 53. Il DNS di internet è **distribuito e organizzato gerarchicamente**. Tale approccio è preferito ad uno centralizzato per diversi motivi:

- **Si evita un singolo point of failure**
- **Il volume di traffico viene regolarizzato e distribuito**
- **Maggior raggiungibilità** (un server non può essere fisicamente vicino a tutti i client)
- **Facile manutenzione e aggiornamento** senza impattare sull'intero Internet.

2.10.1 Gerarchia dei server DNS

Ci sono 3 classi di server DNS:

- **Root DNS servers**: ce ne sono oltre 400 sparpagliati per il mondo e gestiti da 13 organizzazioni diverse. Tali server forniscono gli indirizzi IP dei server TLD.
- **Server Top-Level domain**: ce ne sono uno o più per ogni dominio top-lever (come .com, .org, .it, ecc.). Questi server forniscono gli indirizzi IP degli Authoritative DNS server.
- **Authoritative DNS servers** che possono essere direttamente di proprietà di organizzazioni o offerti da terze parti. Questi server forniscono finalmente l'hostname.

2.10.2 Come funziona una DNS query

Quando un DNS client vuole determinare l'indirizzo IP da un hostname, avviene il seguente procedimento:

1. Il client prima di tutto **contatta uno dei server root**, che ritorna l'indirizzo IP per il server TLD per il dominio top-level che ci interessa (es. ".com").
2. Il client quindi **contatta uno dei server TLD** che restituirà l'indirizzo IP dell'authoritative server.
3. Infine, **il client contatta uno degli autorithative server** che restituirà l'indirizzo IP dell'hostname di partenza.

Questo è ciò che *de iure* dovrebbe accadere, ma non è molto efficiente. Nella realtà dei fatti c'è un altro importante tipo di DNS server chiamata **server DNS locale** oppure **DNS resolver**. Quest'ultimo non appartiene strettamente alla gerarchia dei server ed è tipicamente gestito dagli Internet Service Provider. Quando un host si connette ad un ISP, esso fornirà all'host gli indirizzi IP di uno o più dei server DNS locali. Quando un host crea una DNS query, essa viene inviata al resolver che agisce come un **proxy**, inoltrando la richiesta alla gerarchia dei server DNS.

2.10.3 Aliasing

Il protocollo DNS fornisce un servizio di **aliasing** per i server dove nomi lunghi e complicati (anche detti canonici) sono associati a nomi più semplici e brevi (anche detti alias). Host con nomi complicati possono essere associati a degli alias più mnemonici. Il DNS può essere invocato anche per ottenere l'hostname canonico partendo da un alias.

2.10.4 Distribuzione del carico

Il DNS può essere usato per eseguire una **distribuzione del carico** tra server replicati (aka **round-robin DNS**). Tipicamente, siti molto trafficati sono replicati su server multipli, dove ogni server è eseguito su un differente end system avente un differente indirizzo IP (più indirizzi IP associati con un unico hostname canonico). Il database DNS contiene questo insieme di indirizzi IP.

2.11 Peer-to-Peer

A differenza di quella client-server, l'architettura Peer-to-Peer fa un uso minimo (se non nullo) dei server. Infatti abbiamo coppie di host connessi ad intermittenza che comunicano direttamente tra di loro. I peer **non sono di proprietà dei service provider** ma sono dispositvi controllati dagli utenti. Un'applicazione naturale del P2P è il **file sharing** dove i peer che ricevono un file possono anche condividerlo con altri peer. In qualche modo i peer svolgono funzioni di client e server allo stesso tempo.
Tipicamente nel file sharing l'approccio P2P si adatta meglio rispetto a quello client-server. D'altro canto l'architettura P2P è più difficile da implementare: ci potrebbero essere problemi di sicurezza avendo una grande mole di client in comunicazione diretta.

2.11.1 BitTorrent

Un famoso protocollo P2P per il file sharing è **BitTorrent** (i cui utenti stimati nel 2023 sono 150-170 milioni). Un **torrent** è la collezione di tutti i peer che partecipano nella distribuzione di un file specifico. I peer in un torrent scaricano tra di loro un **chunk** di egual misura del file (la misura tipica è 256 kbytes).
Quando un peer accede ad un torrent:

- Comincia ad accumulare chunks
- Mentre scarica chunks, carica anche altri chunks per altri peer
- Una volta che il peer ottiene l'intero file, potrebbe (egoisticamente) **lasciare il torrent**, oppure (altruisticamente) **restare nel torrent** e continuare a caricare chunks per gli altri peers.

Tutti i torrent hanno un nodo nell'infrastruttura chiamato **tracker** (dei server) che tiene traccia di tutti i peer che partecipano al torrent. Quando un nuovo peer accede al torrent si registra al tracker e **periodicamente** informa il tracker del suo stato. Il tracker fornisce al nuovo peer gli indirizzi IP di un sottoinsieme casuale di peer connessi al torrent. A questo punto il nuovo host cerca di stabilire delle connessioni TCP concorrenti con i peer di questa lista (chiamati vicini). Ogni peer avrà un diverso sottoinsieme di chunks del file. Periodicamente, un host chiederà a ciascuno dei peer connessi la lista dei chunks che posseggono.
Il client decide quale chunks richiedere e a chi seguendo un principio di rarety-first:

- I chunk con meno copie ripetute tra tutti i vicini hanno la priorità. In questo modo i viene equilibrato il numero di copie nel torrent.

Il client che deve caricare decide quale richiesta deve essere servita seguendo due principi interconnessi:

- **Trading**: gli host danno la priorità ai **4 migliori vicini** che in quel momento stanno fornendo dati ad un **rate più alto**. Questo check è eseguito periodicamente (ogni 10 secondi) e la lista dei 4 migliori vicini viene aggiornanta.
- **Random selection**: ogni 30 secondi, un host sceglie casualmente un **ulteriore vicino** e gli invia dei chunk. Se questo scambio casuale è buono, i 2 host entreranno nelle rispettive "**best list**". Questo processo consente ai peer con un upload rate compatibile di trovarsi l'un l'altro.

2.12 Creare applicazioni di rete

La maggior parte delle applicazioni di rete includono un programma **lato client** e un programma **lato server** che comunicano attraverso la rete. Quando questi due programmi vengono eseguiti, **un processo client e un processo server vengono creati**, e tali processi comunicano l'un l'altro leggendo e scrivendo dalle/sulle **socket**. Quindi il compito principale dello sviluppatore è scrivere codice sia per il lato client che per il lato server.

2.12.1 Protocolli

Ci sono due tipi di applicazioni di rete:

- Applicazioni che si affidano a **protocolli standard**. Ci sono diversi protocolli "aperti" le cui regole sono note. In questo caso, i programmi client e server devono essere conformi alle regole e compatibili col protocollo.

- Applicazioni che si affidano a **protocolli proprietari**. In questo caso client e server impiegano un protocollo customizzato. In questo caso il team di sviluppo crea entrambi i programmi client e server.

Ci sono due protocolli di trasporto che possono essere usati:

- Il protocollo **TCP** che è **connection oriented** e fornisce un canale byte-stream affidabile lungo il quale i dati fluiscono attraverso i due end-system.
- il protocollo **UDP** che è **connectionless** e invia pacchetti di dati indipendenti da un end-system all'altro, senza alcuna garanzia sulla consegna

2.12.2 Le socket

Le **socket** sono un elemento fondamentale per la creazione di applicazioni di rete. Tipicamente gestiscono le funzionalità dei livelli di trasporto e di rete. Da notare che ci sono diverse applicazioni (middleware) che possono racchiudere le socket semplificando la loro creazioni e fornendo funzionalità più complesse. Tuttavia le **API di base** sono ancora largamente utilizzate

2.12.3 Connection-oriented vs Connectionless

Nelle applicazioni connectionless, e quindi che utilizzano UDP, il programma lato client, una volta creata la socket, parte direttamente con l'invio del messaggio. Dal canto suo invece il server cercherà sempre di leggere il messaggio, anche se non è presente. Se riceve un messaggio invierà poi una risposta al client che verrà letta da quest'ultimo ed infine la socket verrà chiusa.
Nelle applicazioni connection-oriented, che usano TCP, una volta creata la socket, il client cercherà prima di tutto di stabilire una connessione con un server che è in attesa di un client. A questo punto partirà lo scambio di messaggi. In questo caso è compito del server chiudere la socket del client (chiusura che avviene anche nel codice del client stesso).

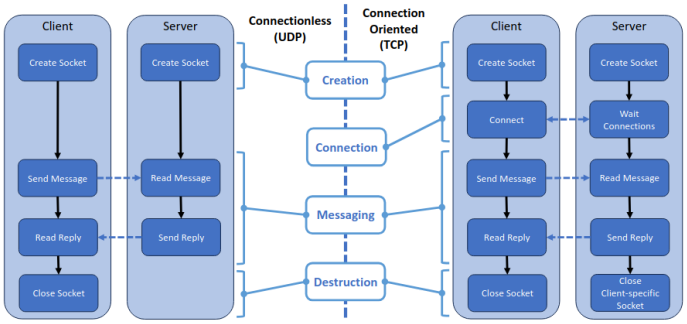


Figure 24: Procedure di comunicazione connectionless e connection-oriented

Le API di base che forniscono socket UDP e TCP sono tipicamente disponibili in tutti i linguaggi di programmazione e sistemi operativi. L'implementazione sottostante e l'utilizzo possono essere più o meno differenti a seconda della configurazione della macchina. Nel dominio UNIX vengono usate le **Berkeley sockets** per entrambe le connessioni TCP e UDP.

2.12.4 Definizione delle socket

Le socket in C fanno affidamento su alcune structure per definire gli indirizzi e le porte degli host mittenti e riceventi:

```
#include <netinet/in.h>
#include <arpa/inet.h> //per inet_addr()
#include <sys/types.h> //alcuni tipi personalizzati sono definiti qui dentro

struct sockaddr_in{
    short   sin_family; //famiglia dell'indirizzo, tipicamente settata a AF_INET (IPv4)
    unsigned short sin_port; //numero della porta
    struct in_addr sin_addr;
    char sin_zero[8]; //vettore di 0 in modo tale che questa struttura abbia la stessa
  dimensione di sockaddr e quindi castabile
};

struct in_addr{
    unsigned long s_addr; //indirizzo IP, puo' essere caricato con inet_addr() o
impostato a INADDR_ANY per localhost
};
```

Listing 1: definizione delle socket in C

Le socket vengono create in C usando la funzione socket():

```
#include <sys/socket.h>

int sockfd= socket(int domain, int type, int protocol);
```

Listing 2: funzione socket()

Dove:

- domain: specifica la famiglia come nella struttura precedente (AF_INET)
- type: specifica il tipo di connessione da creare, SOCK_STREAM per socket TCP, SOCK_DGRAM per socket UDP.
- protocol: specifica un particolare protocollo da usare all'interno della socket (spesso è 0, cioè nessun protocollo specifico)
- sockfd: è il file descriptor che identifica la socket appena creata. (le socket seguono la filosofia UNIX dove tutto è un file)

Possiamo associare una socket ad un indirizzo locale e ad una porta usando la funzione bind():

```
#include <sys/socket.h>
int val=bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Listing 3: funzione bind()

Dove:

- socket: è il file descriptor della socket che vogliamo legare.
- address: puntatore ad una struct sockaddr specificando su quale porta e indirizzo legare la socket. address.sin_addr spesso è impostato a INADDR_ANY in modo tale che le connessioni da tutti gli indirizzi vengano ascoltate.
- address_len: specifica la lunghezza della struttura sockaddr puntata dall'argomento address (ottenibile con sizeof()).
- val: valore di ritorno che è 0 in caso di successo, -1 altrimenti.

Importante notare che questa funzione è usata nei server poiché dobbiamo associare la socket ad una specifica porta. È opzionale invece nei client (il sistema operativo assegna una porta libera automaticamente).

Le operazioni di invio o ricezione dei messaggi in UDP sono eseguite dalle funzioni sendto() e recvfrom():

```
#include <sys/socket.h>
int ob = sendto(int osock, const void *obuf, size_t olen, int oflags, const
     struct sockaddr *oaddr, socklen_t oaddr_len);
```

Listing 4: funzione sento()

```
#include <sys/socket.h>
int ib=recvfrom (int isock, void *ibuf, size_t ilen, int iflags, struct sockaddr
       *iaddr, socklen_t *iaddr_len);
```

Listing 5: funzione recvfrom()

Dove:

- osock/isock: sono i file descriptor su cui inviare,ricevere un messaggio
- obuf/ibuf: sono i puntatori ai buffer contenenti un messaggi da inviare, ricevere
- olen/ilen: le lunghezze dei messaggi (in bytes)
- oflags/iflags: flag specifiche (valori tipici sono 0 o MSG_WAITALL per aspettare finché tutti gli olen/ilen byte sono inviati/ricevuti).
- oaddr/inaddr: puntatori alle struct sockaddr contenenti gli indirizzi riceventi/mittenti.
- oaddr_len/iaddr_len: lunghezza in byte della struttura sockaddr
- ob/ib: numero di byte che sono stati realmente inviati/ricevuti

Le socket possono essere chiuse usando la funzione close():

```
#include <unistd.h>
int val=close(int socket);
```

Listing 6: funzione close()

2.12.5 Esempio in C di applicazione UDP

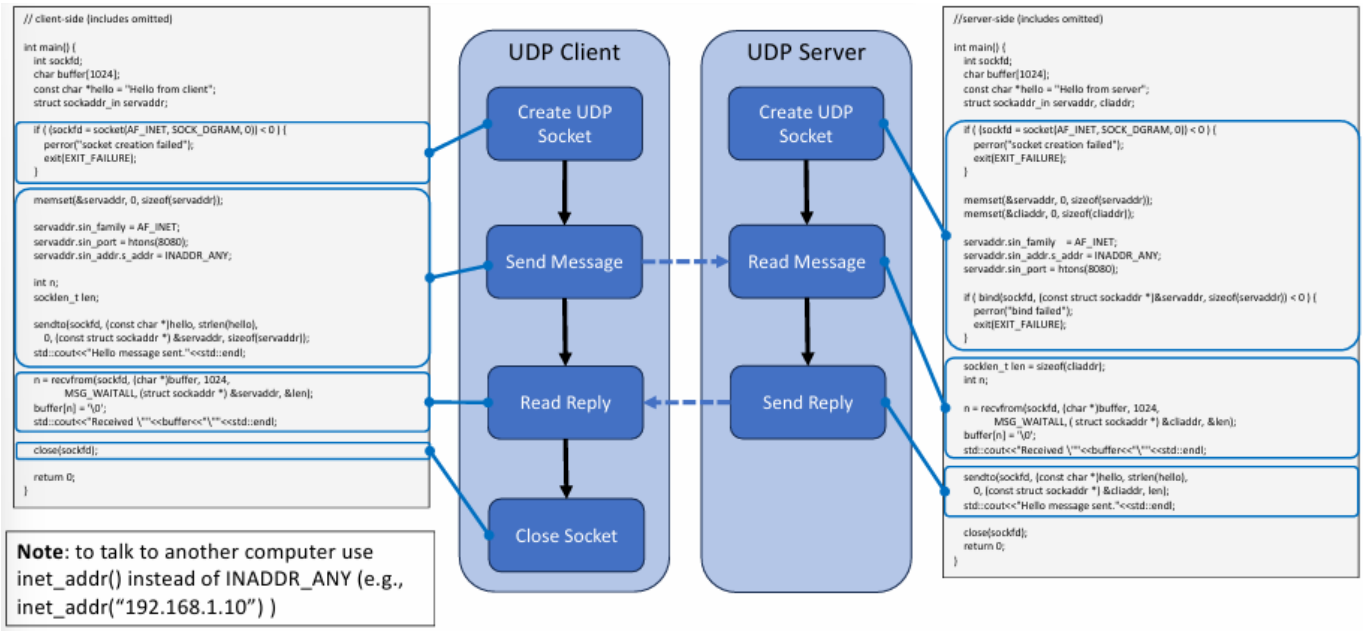


Figure 25: Applicazione client server con protocollo UDP

2.12.6 Da UDP a TCP

Come si può vedere nell'esempio precedente (figura 25) manca completamente la connessione tra client e server. Per quanto riguarda il TCP, invece, client e server devono effettuare un **handshake** prima che i messaggi possano essere trasmessi. Per far ciò usiamo **due socket** nel programma lato server:

- Una "**socket di benvenuto**", che è sempre attiva, e consente i client di eseguire una handshake col server.
- Una socket **client-specific**, creata dopo l'handshake e usata da client e server per comunicare.

La three-way handsake, che viene eseguita al livello di trasporto, è completamente **trasparente** ai programmi client e server. Una volta che la connessione è accettata dal server, la comunicazione passa alla socket appena creata.

2.12.7 Connessione (TCP)

La connessione lato client è eseguita usando la funzione connect():

```
#include <sys/socket.h>

int val=connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

Listing 7: funzione connect()

Dove:

- socket: file descriptor della socket che usiamo per la connessione.
- address: puntatore alla struct sockaddr contenente l'indirizzo del server.
- address_len: specifica la lunghezza della struct sockaddr puntata dal parametro address (si può usare sizeof() per ottenerla)
- val: valore di ritorno che è 0 in caso di successo, e -1 altrimenti.

Il server esegue una wait-for-connection che è eseguita mediante due funzioni: **listen()** e **accept()**, che lavora in combinazione con la connect() del client. La listen() apre una coda dove le connessioni in arrivo vengono "memorizzate". L'accept() apre la socket client-specific.

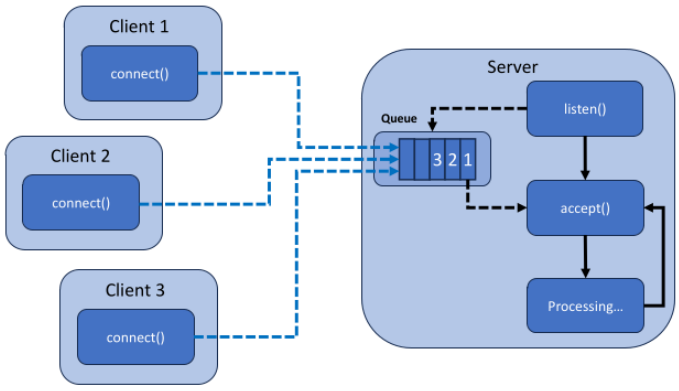


Figure 26: wait-for-connection

Di seguito la definizione di listen() e accept():

```
#include <sys/socket.h>

int val = listen(int socket, int backlog);
int new_sockfd = accept(int socket, struct sockaddr *address, socklen_t *address_len);
```

Listing 8: funzioni listen() e accept()

Dove:

- socket: file descriptor della socket su cui aspettiamo le connessioni.
- backlog: lunghezza massima della coda di connessioni in attesa.
- address: può essere un null pointer, o un puntatore alla struct sockaddr dove l'indirizzo della socket in connessione deve essere restituito.
- address_len: lunghezza dell'indirizzo.
- new_sockfd: file descriptor della nuova socke su cui client e server comunicheranno.
- val: valore di ritorno che è 0 in caso di successo, -1 altrimenti.

L'invio e la ricezione di messaggi in TCP vengono eseguiti dalle funzioni send() e read() (più semplici di quelle in UDP):

```
1    #include <sys/socket.h>
2    int ob=send(int osock, const void *obuf, size_t olen,  int flags);
3    int ib=read(int isock, void *ibuf, size_t ilen);
```

Listing 9: funzioni send() e read()

Dove:

- osock/isock: sono i file descriptor su cui inviare/ricevere un messaggio.
- obuf/ibuf: puntatori ai buffer contenenti un messaggio da inviare/ricevere.
- olen/ilen: lunghezza del messaggio (in byte).
- flags: specificano il tipo di trasmissione del messaggio (dipende dal protocollo, tipicamente 0).
- ob/ib: numero di byte che sono stati realmente inviati/ricevuti.

Da notare che abbiamo già specificato indirizzi client e server durante la fase di connessione, non c'è bisogno di farlo ora (a differenza di sendto/recvfrom)

2.12.8 Esempio in C di applicazione TCP

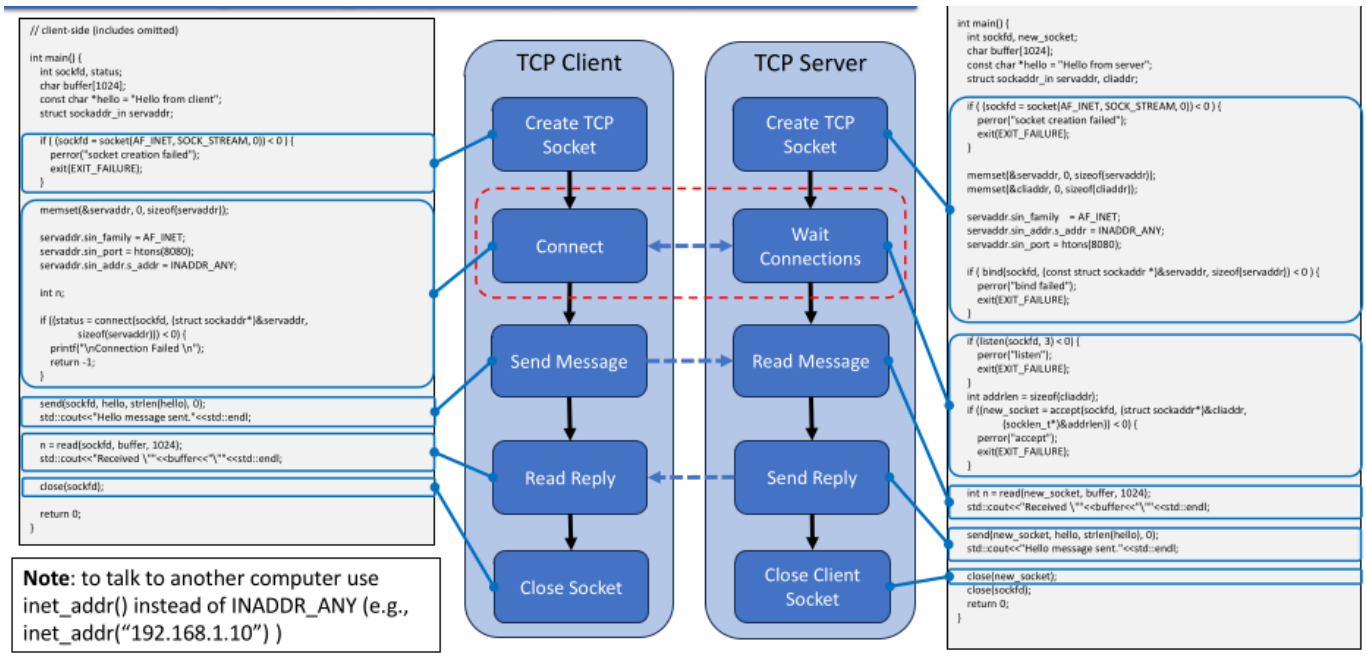


Figure 27: esempio di applicazione TCP

2.12.9 DNS Translation (C/C++)

Come abbiamo appena visto, possiamo creare socket tra due host, ma è necessario conoscere l'indirizzo IP e la porta del server. Su internet gli host sono comunemente identificati tramite il loro **hostname** invece che con i loro **indirizzi IP**.Le Berkeley sockets lavorano unicamente tramite indirizzi IP. Quindi se siamo connessi ad un host tramite hostname, necessitiamo di una traduzione DNS e ottenere l'indirizzo IP corrispondente da passare alla socket.
C e C++ offrono alcune funzioni (ad esempio *gethostbyname()*) e strutture che eseguono tale traduzione, usate per **contattare i server DNS** e ricavare gli indirizzi.
La funzione gethostbyname() non restituisce solo l'indirizzo IP, ma una struttura **hostent** che contiene alcune informazioni sull'host:

- Nome canonico.
- Possibili alias.
- Uno o più indirizzi.

La struttura hostent è così definita:

```
#include <netdb.h>
struct hostent{
    char * h_name; //nome canonico dell'host
    char ** h_aliases; // lista degli alias(terminata da un nullpointer)
    int h_addrtype; //famiglia degli indirizzi (solitamente AF_INET)
    int h_length; //lunghezza dell'indirizzo (di solito 4 byte)
    char ** h_addr_list; //lista degli indirizzi (terminata con un nullpointer)
};
//per ragioni di compatibilita'
#define h_addr h_addr_list[0]
```

Listing 10: definizione della struttura hostent

Un DNS può fornire una lista di indirizzi (ordinata randomicamente se la query è inviata a dei server DNS in round-robin). Possiamo ottenere l'indirizzo del primo elemento dell'array e castarlo nella struttura in_addr usata dalle socket.
La funzione gethostbyname() è invece definita come segue:

```
#include <netdb.h>
struct hostent * host_info = gethostbyname(const char *name)
```

Listing 11: definizione della funzione gethostbyname()

Dove:

- nome: stringa contenente l'hostname da tradurre
- host_info: puntatore alla struttura hostent contenente le informazioni sull'host. (è null pointer in caso di errore)

C'è una funzione duale chiamata gethostbyaddr() che ritorna l'hostent a partire da uno specifico indirizzo IP.

```
//includes omitted
int main(int argc, char **argv){
    //inizializzazione
    struct hostent *server_info;
    char *server_name;
    //ottenere come argomento l'hostname
    if(argc < 2){
        std::cout<<"No hostname specified"<<std::endl;
        return 0;
    }
    else {
        server_name = argv[1];
    }
    //invocare il DNS e verificare se una risposta e' stata ricevuta
    server_info = gethostbyname(server_name);
```

```
    if (server_name == NULL){
        std::cout << "could Not resolve hostname " << server_name << " :(" << std::endl;
        return 0;
    }
    //stampare i vari output (nome canonico, lista degli alias e lista degli indirizzi).
    std::cout<<"Canonical name:"<<std::endl;
    std::cout<<"\t"<<server_info->h_name<<std::endl;
    std::cout<<"Aliases:"<<std::endl;
    int i = 0;
    while(server_info->h_aliases[i] != NULL){
        std::cout<<"\t"<<server_info->h_aliases[i]<<std::endl;
        i++;
    }
    std::cout<<"IPs:"<<std::endl;
    i = 0;
    while(server_info->h_addr_list[i] != NULL){
        std::cout<<"\t"<< inet_ntoa( *( (struct in_addr *) server_info->h_addr_list[i] ) ) <<
    std::endl;
        i++;
    }
    return 0;
}
```

Listing 12: traduzione DNS

2.12.10 Esempio socket HTTP

3 Transport Layer

3.1 Dal livello applicazione al livello di trasporto

Un protocollo al livello di trasporto fornisce per lo più una comunicazione logica tra i processi che vengono eseguiti su differenti host. Dalla prospettiva dell'applicazione, gli host che eseguono **i processi sembrano connessi direttamente** come se fossero sulla stessa macchina.
Il livello di trasporto **converte i messaggi del livello applicazione** in uno o più pacchetti del livello di trasporto, chiamati **segmenti** o **datagram** (il secondo è principalmente usato per i pacchetti UDP):

- Se i messaggi del livello applicazione sono divisi in pezzetti più piccoli (segmenti/datagram), **ogni pezzo è fornito con un header del livello di trasporto**
- I segmenti sono **passati uno ad uno al livello di rete** per essere trasmessi.

3.2 Responsabilità del livello di trasporto

I protocolli del livello di trasporto (UDP e TCP) hanno quattro responsabilità principali:

1. **Consegna process-to-process**: i messaggi sono consegnati indipendentemente da dove tali processi siano. (Ciò è differente dalla consegna host-to-host che è responsabilità del livello di rete).
2. **Verifica dell'integrità**: includendo campi per il rilevamento degli errori negli header dei segmenti.
3. **Trasferimento dati affidabile**: assicurano che il dato sia consegnato dal processo mittente al ricevente correttamente e in ordine.
4. **Controllo del flusso/congestione**: prevengono le connessioni da dispositivi di rete con un eccessivo traffico. Questo servizio è utile per incrementare le prestazioni ed è molto vantaggioso per l'intera rete.

In particolare, UDP (più veloce ma più semplice) fornisce solo i primi 2 servizi, mentre TCP li fornisce tutti.

3.2.1 Consegna process-to-process

Al livello applicazione, diversi processi possono accedere alla rete nello stesso momento. Ed un processo può essere connesso a processi multipli allo stesso tempo. Per risolvere tali problemi, usiamo le socket per la consegna processo-to-process:

- Invece di inviare i messaggi direttamente a processi specifici, usiamo le socket come end-point da cui i processi ottengono i messaggi.
- Questo approccio disaccoppia il numero dei processi dal numero delle connessioni.

Poiché ci sono multiple socket nell'host ricevente, necessitiamo di un **identificatore unico** per ogni socket (**porta**) che deve essere collegato ai segmenti ricevuti. Inoltre abbiamo bisogno anche di un processo di **multiplexing/demultiplexing** sul mittente/ricevente.
Il demultiplexing è il processo di reindirizzamento di un segmento alla socket associata all'identificatore. Il multiplexing, invece, è il processo di creazione dei segmenti da processi differenti, assegnando loro l'identificatore corretto.

3.2.2 Porte

Gli identificatori delle socket sono chiamate porte sorgenti e porte di destinazione. Una porta è un numero a 16bit (numero di porta) che varia da 0 a 65535. In particolare le porte che vanno da 0 a 1023 sono chiamate **well-known port numbers** (numeri di porta ben noti) e sono **riservate** ai protocolli di applicazione. La lista delle porte ben note è aggiornata dalla **IANA**.
Quando sviluppiamo un'applicazione di rete dobbiamo assegnare opportunamente numeri di porta alle socket.
Per verificare l'utilizzo delle prote su un dispositivo Linux possiamo usare il comando nmap. Nmap è una **network scanning utility** creata per mappare gli host e i servizi su una rete.

3.3 Multiplexing e Demultiplexing

Assumiamo di avere un processo nell'host A (porta 19157) che vuole inviare un messaggio ad un processo (porta 46428) nell'host B.

- Multiplexing:
  - Il livello di trasporto nell'**host A crea un segmento/datagramma** che include il dato, la porta sorgente (19157), la porta di destinazione (46428).
  - Il livello di trasporto quindi **passa il segmento/datagramma risultante al livello di rete** che lo incapsula in un datagramma IP, che farà un tentativo di consegna del segmento all'host ricevente
- Demultiplexing:
  - Se il segmento/datagramma arriva all'host B, **il livello di trasporto lo riceve e lo "decapsula"**.
  - Il livello di trasporto quindi **passa il segmento/datagramma alla socket giusta** esaminando il numero della porta di destinazione.

3.3.1 Multiplexing e Demultiplexing in UDP

In un client UDP C++, una socket è creata come segue:

```
1    sockfd=socket(AF\_INET, SOCK\_DGRAM,0);
```

La socket tipicamente non è legata a nessuna porta specifica, è il S.O. che ne sceglie una libera. La porta/indirizzo di destinazione è settata passando una struct sockaddr_in alla funzione sendto:

```
1    servaddr.sin_port=htons(19157);
2    servaddr.sin_addr.s_addr=inet_addr( address_of_B);
3    ...
4    sendto(sockfd, (const char*) msg, strlen(msg),0,(const struct sockaddr*) & destaddr,
     sizeof(destaddr));
```

In un server UDP C++, la socket è creata e legata in questo modo:

```
sockfd=socket(AF_INET, SOCK_DGRAM,0);
...
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_addr.sin_port=htons(19157);
...
binf(sockfd, (const struct sockaddr *)&servaddr,sizeof(servaddr));
```

La porta/indirizzo di destinazione è ricavata attraverso la struttura sockaddr_in dalla funzione recvfrom e usata nel messaggio di risposta:

```
recvfrom(sockfd, (char *) msg, 1024,0,(const struct sockaddr *)&cliaddr, &len);
...
sendto(sockfd,(const char *) msg, strlen(msg),0,(const struct sockaddr *)&cliaddr, sizeof(
cliaddr);
```

Possiamo dunque dire che una socket UDP può essere identificata da 2 elementi: indirizzo IP e porta di destinazione. Infatti, possiamo usare la stessa socket per inviare un messaggio di ritorno a più porte/indirizzi sorgenti.

3.3.2 Multiplexing/Demultiplexing in TCP

Nelle comunicazioni TCP l'applicazione server ha una **socket di benvenuto**, che aspetta le richieste di connessione dai client su uno specifico numero di porta. Il client TCP crea una socket e invia un segmento di richiesta di connessione all'host specificato nella struttura sockaddr_in (nel caso seguente servaddr):

```
sockfd = socket(AF_INET,SOCK_STREAM,0);
...
servaddr.sin_port = htons(19157);
servaddr.sin_addr.s_addr = inet_addr(address_of_B);
...
connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
```

Una richiesta di creazione di connessione è solo un segmento TCP con un numero di porta di destinazione e un bit speciale nell'header TCP (SYN=1).
Quando il server riceve il segmento di richiesta di connessione, **individua il processo server che sta aspettando** di accettare una connessione. Viene quindi creata una nuova socket:

```
new_socket = accept(sockfd, (struct sockaddr*) &cliaddr, (socklen_t*) &addrlen);
```

Il livello di trasporto del lato server riempie la struttura cliaddr **utilizzando numeri di porta e indirizzi dal segmento in arrivo** e crea una nuova socket. Tutti i futuri segmenti, aventi queste specifiche caratteristiche saranno reindirizzate (demultiplexed) alla new_socket.
Quindi possiamo dire che una socket TCP può essere identificata da 4 elementi: IP e porta sorgenti, IP e porta di destinazione.
Solo una sorgente e una destinazione useranno questa socket, se il lato client o server della comunicazione viene interrotto, la socket viene chiusa da entrambi i lati (cosa che non accade in UDP).

3.4 UDP

Come già spiegato, il protocollo UDP è **molto utile per semplici e rapide connessioni**. Esso principalmente esegue 2 compiti: consegna process-to-process e controllo degli errori.
UDP è connectionless: non c'è un handshaking prima di inviare un segmento tra le entità mittenti e riceventi al livello di trasporto. Non c'è garanzia sulla consegna dei messaggi e non c'è controllo sulla congestione.
Perché usare UDP invece di TCP?

- **Controllo al livello applicazione**: UDP è più diretto, quindi l'applicazione ha più controllo sulla trasmissione. Ciò può essere molto utile per istanze di applicazioni real-time, dove il sending rate è cruciale mentre la perdita di dati può essere tollerata.
- **Creazione di connessione veloce**: non c'è un handshake, quindi meno delay per stabilire la connessione.
- **No connection state**: non ci sono parametri per il controllo della congestione.
- **Overhead per i pacchetti minimo**: Il segmento TCP aggiunge 20 byte di header in ogni segmento, invece UDP ne aggiunge solo 8.

3.4.1 Esempio di applicazione UDP: DNS

Un esempio di applicazione che usa le connessioni UDP è il DNS. Il DNS client lavora come segue:

- Nel **livello applicazione**: quando un host vuole creare una query ad un DNS server, costruisce una query DNS e passa il messaggio a UDP.
- Nel **livello di trasporto**: senza eseguire nessun handshake con il server DNS, UDP aggiunge i campi header al messaggio e passa il segmento risultante al livello di rete.
- Nel **livello di rete**: il segmento UDP è incapsulato in un datagramma IP e inviato al DNS server (attraverso il livello di accesso).
- Il client quindi aspetta una risposta alla sua ricchiesta. Se la **risposta non viene ricevuta**, può tentare differenti approcci:
  - Reinviare la query.
  - Inviare la query ad un altro server.
  - Informare l'applicazione che non è stata ricevuta nessuna risposta.

3.4.2 Utilizzi di UDP

Le applicazioni che necessitano di un **trasferimento dati affidabile**, come le email, utilizzano TCP (o protocolli basati su UDP ma più affidabili come QUIC). In questi casi infatti non possiamo permetterci perdita di pacchetti.
SNMP usa UDP per svolgere la gestione dei dispositivi (device management). Le applicazioni di network management spesso devono essere eseguite quando la rete è in uno stato stressato.
Le **applicazioni multimediali** come quelle per le videoconferenze o streaming, spesso usano sia UDP che TCP, perché una piccola quantità di pacchetti persi può essere tollerata. In generale, le applicazioni real-time reagiscono molto scarsamente al controllo della congestione di TCP. Dall'altra parte, **eseguire applicazioni multimediali con UDP è controverso** perché non viene eseguito nessun controllo di congestione. Se tutti avviassero uno streaming ad alto bit rate senza utilizzare nessun controllo di congestione, **i dispositivi di rete potrebbero causare un overflow** causando la perdita di più pacchetti UDP. Un alto loss rate indotto da mittenti UDP incontrollati potrebbero causare un drammatico calo del rate anche ai mittenti TCP.

3.4.3 Formato del segmento UDP

Il **datagramma UDP** è composto da 5 elementi:

- **UDP header** (64 bit) che include informazioni relative al protocollo UDP ed è composto da 4 elementi da 16 bit ciascuno:
  - **Source port**: numero di porta del processo mittente.
  - **Destination port:** numero di porta del processo ricevente.
  - **Lunghezza** dell'intero datagramma (header+dato)
  - **Checksum**: usato dal ricevente per verificare che il messaggio sia intatto.
- **Data**(N bit) che è il messaggio ottenuto dal livello applicazione.

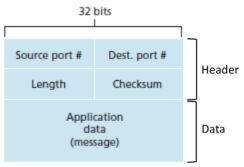


Figure 28: Formato del datagramma UDP

Checksum La verifica dell'integrità è eseguita utilizzando il checksum. Il checksum è un campo a 16bit usato per la rilevazione di errori in questo modo:

- UDP dal lato mittente esegue **il complemento a 1 della somma di tutti le parole a 16bit** nel datagramma, con l'eventuale overflow che viene anch'esso sommato.
- Il **risultato è messo nel campo checksum** del datagramma UDP.
- Quando il **ricevente somma tutti i bit nel suo messaggio (incluso il checksum)** il risultato deve essere 1111111111111111 (16 volte uno). Se anche un solo bit è 0 allora sappiamo che **è stato introdotto un errore** nel pacchetto.

3.5 Reliable Data Transfer

3.5.1 Il problema del Reliable Data Transfer

Per affidabilità (**reliability**) intendiamo la garanzia che il messaggio arrivi a destinazione senza essere alterato.
Un messaggio, diviso in parole, potrebbe essere alterato in vari modi: le parole potrebbero essere **perse**, **alterate**(di significato),**scambiate** o **duplicate**. Può capitare che si riesca a capire che il messaggio sia sbagliato, ma ad esempio nel caso di parole alterate o scambiate, il messaggio potrebbe essere inteso in modo errato.
L'error control (ad esempio mediante checksum) consente al ricevente di capire almeno **se il messaggio è alterato**, ma non è abbastanza. Il reliable data transfer è ancora uno dei principali problemi del networking. Tale problema non è assegnato solo al livello di trasporto, ma anche al livello di link e al livello applicazione.
In un canale affidabile (reliable) devono essere garantiti 3 elementi:

1. Nessun bit trasmesso deve essere **corrotto**
2. Nessun bit trasmesso deve essere **perso o ripetuto**
3. Tutti i bit consegnati devono essere **nello stesso esatto ordine** con cui sono stati inviati

In generale, dobbiamo assumere che il livello sottostante della pila (il livello di rete) sia **inaffidabile**. Questa assunzione è particolarmente realistica, ad esempio TCP è un protocollo affidabile implementato su un protocollo del livello di rete inaffidabile (IP).

3.5.2 Corruzione dei pacchetti e Stop-and-wait

Assumiamo di avere un canale in cui i bit possono unicamente essere corrotti (e non persi). La corruzione dei bit è un problema tipico, imputabile principalmente alle componenti fisiche della rete.
Un primo approccio di risoluzione è lo **Stop-and-wait** che richiede che ogni messaggio debba essere **acknowledged** (riconosciuto) prima di inviarne uno nuovo:

- **Acknowledgment Positivo** (ACK) significa che il messaggio sia stato ricevuto inalterato.
- **Acknowledgment Negativo** (NCK) significa che c'è stato un errore, quindi il messaggio deve essere ripetuto.

In una rete, i protocolli basati sulla ritrasmissione sono anche noti come protocolli **ARQ** (Automatic Repeat reQuest).
Ci sono un paio di problemi con questo approccio:

- Cosa succede se un **messaggio ACK/NCK è a sua volta corrotto**?
- L'host B come può essere sicuro che il messaggio sia una ripetizione piuttosto che un nuovo messaggio?

Viene quindi aggiunto un nuovo campo negli header dei pacchetti, il **Sequence number**, che specifica l'ordine in cui il pacchetto dovrebbe essere ricevuto. Dato che in un approccio stop-and-wait c'è sempre **un messaggio** nel canale, non c'è bisogno di specificare l'intera sequenza dei pacchetti, **abbiamo semplicemente bisogno di distinguere il pacchetto corrente da quello precedente**. Pertanto, in questo caso, necessitiamo di aggiungere un solo bit (s=0/1) agli header dei pacchetti.

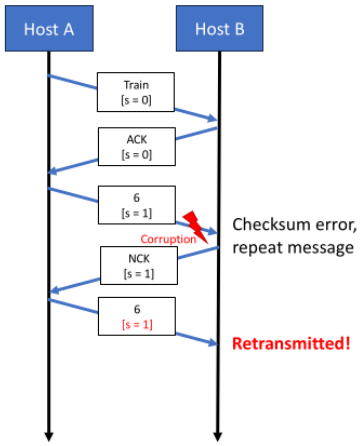


Figure 29: Esempio di approccio Stop-And-Wait

3.5.3 Pacchetti persi e Stop-and-wait

Ora assumiamo di avere un **canale dove i pacchetti possono anche essere persi**. È abbastanza ragionevole dato che i dispositivi di rete possono avere un overflow del buffer in caso di intenso traffico. In questo caso abbiamo un loop nell'approccio stop-and-wait. Se un messaggio viene perso, gli host non ne manderanno mai uno nuovo.
Una soluzione molto semplice è il **timeout** che viene aggiunto nel lato del mittente. Una volta scaduto, l'host ritenterà nuovamente ad inviare il pacchetto. Questa soluzione funziona **sia nel caso di un pacchetto perso che nel caso di un acknowledgment perso**. Nel caso di una duplicazione il ricevente deve solo cancellare il pacchetto.
La sfida è ora decidere **come stimare un timeout adatto**. Possiamo dire che un timeout ragionevole dovrebbe essere in qualche modo più lungo dell'RTT. Il timeout evita la presenza di loop in un approccio stop-and-wait, ma allo stesso tempo **peggiora le prestazioni**.

Esempio idraulico Possiamo vedere il trasferimento dati **come un problema idraulico**:

- C'è un **tempo t** che indica quanto ci vuole per spostare l'acqua da un serbatoio al tubo. Mentre l'acqua entra, allo stesso tempo si sta muovendo all'interno del tubo.
- Dopo il tempo t **tutta l'acqua del serbatoio si trova ora nel tubo** (sia il serbatoio di partenza che quello di destinazione sono vuoti)
- L'acqua attraversa tutto il tubo in un tempo $RTT/2$
- Infine, **l'acqua arriva al serbatoio di destinazione** e impiega un tempo t per riversarsi fuori dal tubo. In totale $\mathbf{RTT/2 + t}$.

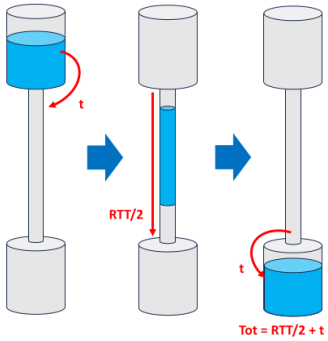


Figure 30: Esempio idraulico

3.5.4 Performance dell'approccio stop-and-wait

Consideriamo un caso ideale di due host che si trovano sulle due coste degli Stati Uniti. La speed-of-light RTT tra questi due host è approssimativamente 30 millisecondi. Ora assumiamo di avere:

- Un canale con un trasmission rate (R) di 1 Gbps (10^9 bit per secondo).
- La dimensione del pacchetto (L) di 1000 bytes (8000 bit)

Il tempo t necessario per trasmettere il pacchetto al canale è:

$$t = \frac{L}{R} = \frac{8000}{10^9} = 0.000008s$$

In un protocollo **stop-and-wait**:

- Il **mittente inizia l'invio** del pacchetto a t_0
- L'**ultimo bit entra nel canale** a $t_0 + 0.000008s$
- Il **pacchetto impiega** $0.015s$ **per raggiungere il ricevente**
- L'**ultimo bit è ricevuto** a $t = RTT/2 + L/R = 0.0150008s$
- Assumendo che i pacchetti ACK siano estremamente piccoli (il loro tempo di trasmissione può essere omesso), l'ACK torna al mittente a $t = RTT + L/R = 0.030008s$

Sui 0.030008 s di trasmission time, il mittente era in attesa per il 99,973% del tempo.

3.5.5 Pipelining e le sue performance

A differenza dello stop-and-wait, nel **pipelining** il mittente **può inviare pacchetti multipli** senza aspettare l'ACK. Negli approcci pipelining (con ad esempio 3 pacchetti):

- Il **mittente inizia l'invio dei 3 pacchetti** a t_0.
- L'**ultimo bit dell'ultimo pacchetto entra nel canale** a $t_0 + 0.000024s$
- I **pacchetti impiegano 0.015s per raggiungere il ricevente.**
- L'**ultimo bit viene ricevuto** a $t = RTT/2 + L/R = 0.15024s$
- Omettendo nuovamente il trasmission time dei pacchetti ACK, essi ritornano al mittente a $t = RTT + L/R = 0.030024s$

Sul totale del tempo, il ricevente era in attesa lo 0.035% in meno (99,920%).
Negli approcci pipelining:

- Il **range dei numeri di sequenza deve essere incrementato** dato che in ogni pacchetto in transito deve esserci un numero di sequenza univoco e potrebbero esserci molteplici pacchetti in transito.
- C'è **bisogno di buffer per memorizzare i pacchetti in arrivo** dato che non sappiamo se i pacchetti sono corretti o ci sono "buchi nella trasmissione.

Ci sono 2 protocolli di base che usano il pipelining: **Go-Back-N** e **Selective Repeat**.

Go-Back-N (GBN) In questo protocollo il mittente può inviare pacchetti multipli **senza aspettare un ACK** ma è limitato ad avere **non più di N pacchetti unacknowledged**

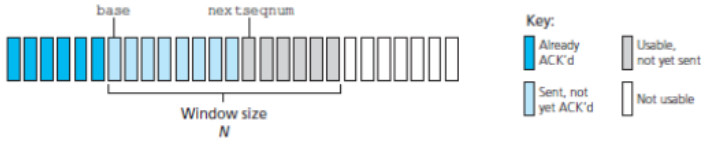


Figure 31: Protocollo Go-Back-N

- Base: è il **numero di sequenza del pacchetto unacknowledged più vecchio**.
- Nextseqnum: è il **numero di sequenza inutilizzato più piccolo** (il prossimo che sarà spedito)
- Abbiamo che:
  - I pacchetti da 0 a base-1 sono stati **trasmessi e riconosciuti.**
  - I pacchetti da base a nextseqnum-1 sono stati **inviati ma non ancora riconosciuti**
  - I pacchetti da nextseqnum a base+N-1 **possono essere inviati**
  - I pacchetti da base+N a $+\infty$ **non possono essere usati** finché un nuovo acknowledgment non viene ricevuto.

Nel protocollo GBN il **ricevente è abbastanza semplice**. Esso deve solamente **scartare i pacchetti out-of-order** (che siano danneggiati o meno) e **deve inviare al livello superiore (applicazione) solo i pacchetti in order**. I pacchetti scartati possono eventualmente essere ritrasmessi dal mittente.
Ovviamente, lo svantaggio di buttare un pacchetto correttamente ricevuto è che **dobbiamo inviarlo di nuovo**. Ciò è una **reazione a catena**, se stiamo perdendo pacchetti a causa di difficoltà della rete, molti pacchetti correttamente ricevuti ma out-of-border verranno scartati e ritrasmessi poi dal mittente. **A sua volta la ritrasmissione può non andare a buon fine** e così saranno richiesta ulteriori ritrasmissioni.

Selective Repeat (SR) In questo tipo di protocolli il mittente ritrasmette **solo i pacchetti che sono stati ricevuti con un errore** (persi o corrotti):

- Come nel GBN **i singoli pacchetti sono acknowledged**.
- Viene usata nuovamente una dimensione della finestra **N** per limitare il numero di pacchetti non riconosciuti.
- A differenza del GBN, i **pacchetti out-of-order vengono memorizzati nel buffer** e riconosciuti dal ricevente

Un punto critico del SR è che la dimensione della finestra è legata al numero di sequenza, e quest'ultimo è **finito**. Nel seguente esempio (figura 32) abbiamo 4 pacchetti, un numero di sequenza massimo di 3 e una dimensione della finestra di 3.
Assumiamo che i pacchetti da 0 a 2 siano correttamente ricevuti e il loro ACK sia stato inviato, ma non ancora ricevuto. La finestra del ricevente si sposta sul 6° pacchetto, anche se gli ACK non sono stati ricevuti per certo. Possiamo avere due casi

- **Caso a**: gli ACK dei primi tre pacchetti sono stati persi e il mittente ritrasmette questi pacchetti. *Il pacchetto 0 è nuovo o vecchio?*
- **Caso b**: gli ACK sono stati ricevuti, i pacchetti 3 e 0 sono stati inviati ma il pacchetto 3 è stato perso. *Il pacchetto 0 è nuovo o vecchio?*

Per evitare questo problema, il **numero di sequenza deve essere almeno 2 volte la dimensione della finestra**.

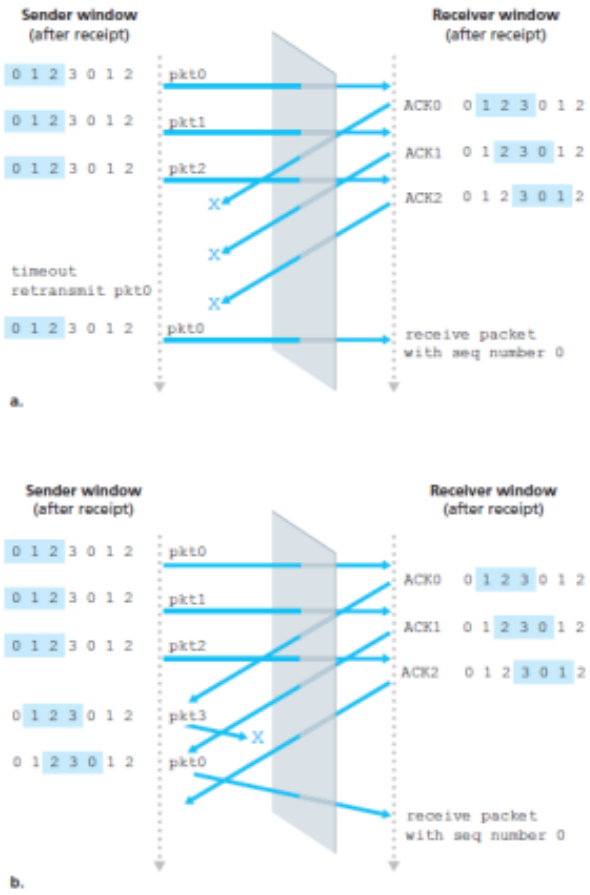


Figure 32: Esempio Selective Repeat

3.6 TCP

TCP, a differenza di UDP, è:

- **Connection-oriented**: c'è una fase di handshake prima della trasmissione che assicura la connessione tra il processo mittente e quello ricevente.
- **Affidabile** (reliable): rilevamento degli errori, ristrasmissione, acknowledgment, timer e i numeri di sequenza vengono implementati.
- **Congestion and flow aware**: viene applicata una regolazione del transmission rate in base alle prestazioni del ricevente (**flow**) e le prestazioni della rete (**congestion**).

La connection orientation rende il TCP full-duplex e point-to-point:

- **Full duplex**: se host A è connesso con l'host B, allora B è connesso con A.
- **Point-to-point**: singolo mittente e singolo ricevente (il trasferimento dei dati da un mittente a più riceventi non è possibile).

TCP è anche orientato sui sending/receiving **data streams**:

- Molteplici segmenti TCP potrebbero essere parte di un più grande data stream (sequenza ordinata di dati)
- Datagrammi UDP singoli sono considerati per lo più disaccoppiati

3.6.1 Buffer TCP

Nelle connessioni TCP le operazioni di invio e di ricezione fanno forte **affidamento sui buffer** per essere eseguite. I buffer consentono un **parziale disaccoppiamento** dei tempi di trasmissione da **latenze dell'Applicazione**, del **sistema operativo** e della **rete**. Inoltre, c'è un limite al transmission rate, in questo modo messaggi lunghi possono essere **spezzati in segmenti più piccoli** che possono essere raccolti nei buffer prima di essere disassemblati o riassemblati.

3.6.2 TCP Maximum Segment Size

La quantità di dati all'interno di un segmento è limitata dal **Maximum Segment Size** (MSS)

$$MSS = MTU - header\_size$$

Dove:

- MTU sta per **Maximum Transmission Unit** che rappresenta la massima lunghezza di un frame accettabile dal livello di link
- Header_size rappresenta la dimensione combinata degli header TCP e IP (tipicamente 20+20 byte)

Nella fase di invio i segmenti sono creati a partire dai dati dell'applicazione e passati giù al livello di rete, dove vengono separatamente incapsulati nei datagrammi IP per essere spediti.
Nella fase di ricezione i dati sono piazzati nel receive-buffer, l'applicazione preleva i dati dal buffer quando è pronto.

3.6.3 Segmento TCP

Il segmento TCP è costituito da **campi header** e **campi di dati**, quest'ultimo contiene dei dati del livello applicazione la cui dimensione è al massimo uguale all'MSS.

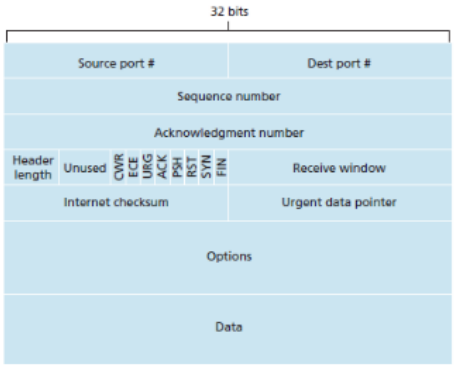


Figure 33: segmento TCP

L'header contiene diversi campi:

- **Numero di sequenza** (32 bit) per trasferimento dati affidabile.
- **Acknowledgment number** (32 bit) per trasferimento dati affidabile.
- **Receive window** (16 bit) usata per indicare il numero di byte che un ricevente è disposto ad accettare (per il flow control)
- **Lunghezza dell'header** (4 bit): specifica il numero delle parole da 32 bit contenute nell'header. (L'header TCP è **variabile a causa del campo opzioni**)
- **Campo opzioni** (K-bit) usato per processi opzionali come negoziazione dell'MSS, time-stamping, etc.
- **Flags** (6-12 bit) che include:
  - ACK bit indica che **questo segmento è un pacchetto ACK**.
  - RST, SYN e FIN sono usati per la configurazione e teardown della connessione
  - CWR e ECE sono usati nella notifica esplicita di congestione (opzionali)
  - PSH indica al ricevente di passare il dato all'applicazione immediatamente
  - URG indica che il segmento contiene dati urgenti
- **Checksum** (16 bit): per il controllo di integrità.
- **Urgent data pointer field** (16bit) indica la locazione dell'ultimo byte della parte urgente del dato.

3.6.4 Numero di Sequenza e Numero di Acknowledgment

Il numero di sequenza è usato non solo per il trasferimento dati affidabile, ma anche per **gestire la segmentazione**. Se viene spedito un data stream grande, dobbiamo romperlo in "pezzi" più piccoli in base all'MSS. Ciascun pezzo dello stream è messo in un segmento TCP.
I numeri di sequenza e di acknowledge sono **strettamente correlati**. Infatti:

- Il **numero di sequenza** di un segmento è convenzionalmente **la posizione in cui si trova il primo byte del dato del segmento nel data stream**
- Il **numero di acknowledgment** è convenzionalmente il **pezzo successivo del data stream**.

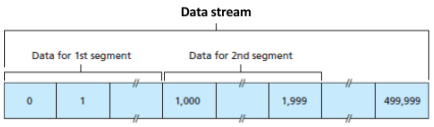


Figure 34: Esempio segmentazione data stream

Nell'esempio (figura 34), assumendo un data stream di 500kB, e un MSS di 1000 byte, il TCP costruisce 500 segmenti dove il primo segmento ha il numero di sequenza uguale a 0, il secondo segmento uguale a 1000, il terzo 2000 e così via.
Assumiamo ora che il data stream deve essere inviato da host A a host B. Per semplicità, ignoriamo la precedente interazione, e quindi partiamo dal numero di sequenza 0:

- Il ricevente otterrà il primo segmento di 1000 byte cioè **dal numero di sequenza 0 al numero di sequenza 999**.
- Il ricevente dovrà quindi riconoscere la trasmissione settando un **numero di acknowledgment uguale a 1000** (il prossimo byte previsto nello stream)
- Il ricevente otterrà il prossimo segmento di 1000 byte, quindi dal numero di sequenza uguale a 1000 fino al numero di sequenza uguale a 1999 (e così via...)

Cosa succede se il ricevente sta inviando dati in ritorno? (ricordiamo che la comunicazione è full-duplex e entrambi i flussi sono considerati affidabili).
In questo caso possiamo usare i numeri di sequenza e di acknowledgment allo stesso tempo:

- I segmenti da A hanno numeri di **sequenza correlati allo stream A-to-B** e di **acknowledge correlati allo stream B-to-A**
- I segmenti da B hanno numeri di **sequenza correlati allo stream B-to-A** e di **acknowledge correlati allo stream A-to-B**

3.6.5 Timeout di Ritrasmissione in TCP

Il trasferimento dati affidabile di TCP **fa un uso estensivo dei timeout.** L'approccio di default nelle trasmissioni TCP basate sul pipelining è di **ritrasmettere i segmenti solo se l'ACK associato non viene ricevuto prima del timeout**. Perciò, **la stima del timeout è una scelta critica** dato che può impattare pesantemente le prestazioni della trasmissione. Scegliere un timeout troppo lungo causerà il rallentamento della comunicazione. Invece sceglierne uno troppo corto causerà un overlap dei pacchetti.

Stima dell'RTT Per impostare un timeout adeguato necessitiamo di **una stima del round-trip time (RTT)**. Il timeout dovrebbe essere più grande dell'RTT altrimenti potrebbero essere inviate ritrasmissioni non necessarie.
Il TCP stima questo valore **campionando l'RTT dei segmenti riconosciuti con successo** che non sono stati ritrasmessi. Dato che il tempo di un RTT campionato (samRTT) **potrebbe variare** (in base al traffico e altri fattori) la stima dell'RTT (estRTT) è data dall**exponential weighted moving average (EWMA)**:

$$estRTT_t = (1 - \alpha)estRTT_{t-1} + \alpha samRTT_t$$

Dove α è un parametro che di solito è 0.125.
Possiamo anche considerare la variabilità dell'RTT nel calcolo del timeout. La **variabilità dell'RTT** (devRTT) può essere data dalla differenza tra il campione (samRTT) e la stima (estRTT):

$$devRTT_t = (1 - \beta)devRTT_{t-1} + \beta|samRTT_t - estRTT_t|$$

Dove β è un parametro che di solito è 0.25.

Avendo queste stime, è possibile definire un timeout di ritrasmissione per TCP che non è né più basso né più alto dell'RTT stimato:

$$timeout_t = estRTT_t + 4devRTT_t$$

La variabilità dell'RTT (devRTT) è usata per impostare un margine ragionevole e che si adatta alle oscillazioni dell'RTT. In questo modo la finestra di timeout è più larga quando non siamo certi del valore dell'RTT attuale.
Di default, il valore iniziale dell'RTT (a t=0) è di 1 secondo.

3.6.6 Fast Restransmit

La ritrasmissione innescata dal timeout è abbastanza efficace, ma il **periodo di timeout può essere relativamente lungo**.
Il **fast retransmit** è un approccio in cui il ricevente segnala il mittente che un pacchetto potrebbe essere stato perso inviando **ACK duplicati**.

- Quando il TCP receiver riceve un segmento con un **numero di sequenza che è maggiore di quello previsto** significa che un segmento è stato mancato.
- Il receiver **rinvia il vecchio ACK** (ACK duplicato) avente come numero ACK quello del segmento previsto.
- Se il **mittente riceve N duplicati** (tipicamente 3 duplicati), esso assume che il segmento precedente sia stato perso quindi viene ritrasmesso **prima della deadline** (ovvero prima che il timeout scada)

In questo esempio (figura 35) abbiamo un segmento perso con numero di sequenza uguale a 100.

- Ogni volta che B riceve un segmento out-of-order esso **invia indietro un ACK duplicato** (ack=100).
- Quando l'host A **riceve 3 duplicati**, esso assume che il segmento 100 è stato perso e lo **invia di nuovo** (fast retransmit).
- Quando l'host B riceve il segmento perso, **invierà l'ACK del prossimo segmento previsto** in base alla sua policy (Go-Back-N o selective repeat).

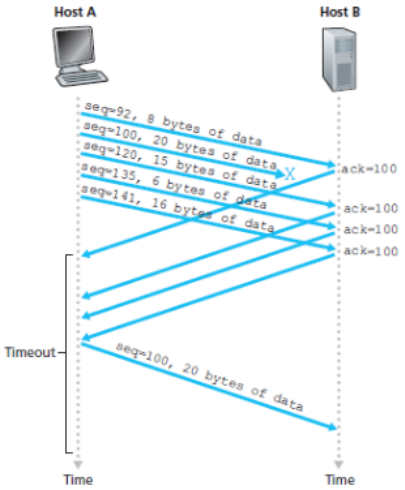


Figure 35: Esempio di fast retransmit

Ma perché non ritrasmettiamo i segmenti direttamente quando il primo ACK duplicato è ricevuto? Perché **potremmo ricevere l'ACK sbagliato per diverse ragioni** (ad esempio se 2 segmenti vengono inviati nell'ordine sbagliato).

3.6.7 Problemi nella gestione delle connessioni TCP

Dato che TCP è connection-oriented **due host devono accordarsi** sia nella procedura di **apertura** che di **chiusura**. Sapendo che i messaggi potrebbero essere **persi o danneggiati** sulla rete, **è abbastanza difficile** per due host raggiungere tale accordo. Se i messaggi sono persi durante la trasmissione **uno dei due capi della comunicazione potrebbero aprire o chiudere la connessione mentre l'altro capo no**. Per evitare questo problema TCP implementa una procedura chiamata **three-way handshake** dove le richieste di apertura/chiusura **devono essere riconosciute** dagli host prima che una connessione possa essere stabilita/rilasciata.

Esempio: problema delle due armate Immaginiamo di avere un **armata bianca accampata in una valle** e, su entrambe le colline a lato, ci sono **2 armate blu nemiche**. **L'armata bianca è più grande delle due blu prese singolarmente**, ma insieme le blu sono più grandi e risulterebbero vincitrici solo se attaccano simultaneamente. Per sincronizzare i loro attacchi, **le armate blu devono inviare messaggeri attraverso la valle** dove potrebbero essere catturati (comunicazione non affidabile). Esiste un protocollo che consente ai blu di vincere?

Supponiamo che il comandante dell'**armata blu #1** invii un messaggio dove propone l'attacco. Ora supponiamo che il **messaggio arrivi**, e che la sua **risposta arrivi in modo sicuro** all'armata #1.
L'attacco avverrà? Probabilmente no perché il **comandante #2 non sa se la sua risposta è arrivata.** Se non è arrivata, l'armata #1 non attaccherà.
Per effettuare una **three-way handshake** ora il primo comandante **deve riconoscere** la risposta. Assumendo che **nessun messaggio sia perso**, l'armata #2 otterrà l'acknowledgment, ma il comandante dell'armata #1 ora esiterà (non sa se il suo acknowledgement sia arrivato).
A questo punto potremmo trasformarlo in un four-way handshake, ma ciò non aiuterà. Infatti, **può essere provato che non esiste un protocollo che funzioni**. Il three-way handshake non è perfetto ma di solito è adeguato.

3.6.8 Stabilimento di una Connessione TCP

In TCP per stabilire una connessione viene eseguita una three-way handshake:

- Il client invia una **richiesta di connessione** (segmento SYN) al server.
- Il server risponde con un **acknowledgment speciale** (segmento SYN-ACK).
- Il client invia un **ultimo acknowledgment** (segmento ACK).

Stabilire una connessione è una **procedura delicata** che può anche aggiungere ritardi significativi. Tipicamente c'è un **timeout** (30-60 secondi) per completare l'handshake, dopo cui **la procedura viene abortita**. Alcuni attacchi (come il SYN flood) avvengono qui.
Nel dettaglio, la procedura del three-way handshake avviene in questo modo:

1. Il client invia un **segmento SYN** avente:
   - Nessun dato del livello applicazione (payload).
   - Il bit SYN settato a 1.
   - Un numero di sequenza iniziale casuale (client_isn) come numero di sequenza.
2. Quando il segmento SYN arriva, il **server alloca buffer e variabili TCP** e invia indietro un **segmento SYNACK** avente:
   - Nessun dato del livello applicazione (payload)
   - I bit SYN e ACK settati a 1.
   - Il numero di acknowledgment settato a client_isn+1.
   - Un numero di sequenza iniziale casuale (server_isn) come numero di sequenza.
3. Quando il segmento SYNACK arriva, il **anche il client alloca buffer e variabili** alla connessione e invia al server un ultimo **segmento ACK** avente:
   - Possibili dati del livello applicazione.
   - Il bit SYN settato a 0 (la connessione è stabilita) e un bit ACK settato a 1.
   - Il numero di acknowledgment impostato a server_isn+1.

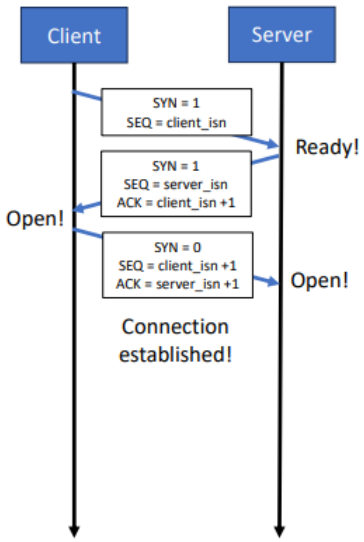


Figure 36: Stabilimento di una connessione TCP

3.6.9 Rilascio di una connessione TCP

Il rilascio di una connessione TCP (aka **teardown**) può essere eseguita da entrambi gli host. Ipotizziamo che il client stia chiudendo la connessione, la **three-way handshake** è eseguita come segue:

1. Il client invia un **segmento speciale di shutdown** (segmento FIN) al server impostando il bit di FIN a 1.
2. Quando il server riceve questo segmento, invia come risposta **un segmento di acknowledgment/shutdown**, impostando i bit ACK e FIN a 1.
3. Infine il client **riconosce** il segmento di shutdown del server

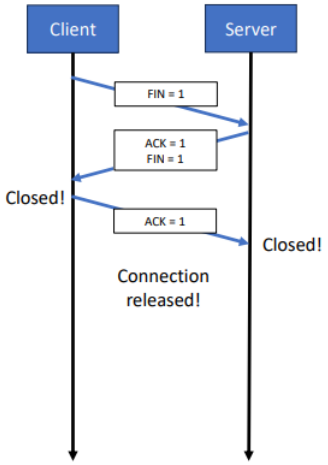


Figure 37: Rilascio di una connessione TCP

Dato che la three-way handshake non è perfetta,in caso di pacchetti persi TCP si affida ai timer per chiudere le connessioni.

3.6.10 Problema del Congestion and Flow control

Ulteriori funzionalità di TCP rispetto ad UDP sono i **congestion and flow control**. Abbiamo anticipato che i pacchetti persi a volte sono **risultato dell'overflow dei buffer**:

- Dal buffer ricevente, se **l'applicazione ricevente non è abbastanza veloce nella lettura dei dati**.
- Dai dispositivi di rete, se i **nodi sono congestionati**

. Seguendo la nostra analogia del flusso d'acqua usata precedentemente, **possiamo vedere i buffer come secchi intermedi** che traboccano in caso di acqua eccessiva. Se i pacchetti sono persi, **gli host sono forzati ad affidarsi alla ritrasmissione e al timeout** che impattano pesantemente sulle prestazioni della rete.

3.6.11 Flow Control

Quando la connessione TCP riceve dei byte che sono corretti e in sequenza, essa piazza i dati nel buffer di ricezione. L'applicazione ricevente **leggerà i dati da questo buffer**, ma **non necessariamente nell'istante in cui i dati arrivano** (l'applicazione potrebbe essere impegnata). Se l'applicazione è relativamente **lenta nel leggere i dati**, il mittente può facilmente causare un **overflow del buffer di ricezione** inviando troppi dati troppo velocemente.
Il TCP **flow control è un servizio di speed-matching**: tenta di eguagliare (riducendo) il rate a cui il mittente invia rispetto al rate a cui l'applicazione ricevente riesce a leggere.
In questo caso ci affidiamo ad una **finestra di ricezione** per informare il mittente di quanti segmenti può ricevere senza produrre overflow. Ricordiamo che il segmento TCP consente al **ricevente di dire al mittente** quanto spazio del buffer resta tramite **il campo receive window**

Assumiamo per semplicità che il **TCP receiver scarti i segmenti out-of-order**, quindi tutti i segmenti nel buffer sono ordinati. Dal lato del receiver abbiamo:

- s_{buff}: dimensione del buffer del receiver (in byte)
- i_{read}: l'ultimo byte dello stream estratto dall'applicazione
- i_{rec}: l'ultimo byte dello stream ad essere ricevuto.
- Possiamo definire la dimensione della **receive window** (rwnd) come:

$$rwnd = s_{buff} - (i_{rec} - iread)$$

Dal lato del mittente abbiamo:

- i_{sent}: l'ultimo byte dello stream ad essere inviato
- i_{ackd}: l'ultimo byte dello stream che è stato riconosciuto dal ricevente
- Conoscendo la receive window, il mittente **garantisce in qualsiasi momento** che:

$$i_{sent} - i_{ackd} \leq rwnd$$

3.6.12 Congestion Control

Il problema del congestion flow è simile al flow control, ma è collegato all'infrastruttura di rete. Ad esempio: due host (A e B) hanno una connessione che **condivide un singolo router** ed un singolo link in uscita di capacità R tra sorgente e destinazione. Il **router ha dei buffer** che consentono di immagazzinare pacchetti in arrivo quando il rate di pacchetti in arrivo supera la capacità del link in uscita.

Assumiamo per semplicità che entrambe le applicazioni in A e B stiano inviando dati nella connessione allo stesso **rate medio** di λ_{in} byte/sec. Se $\lambda_{in} \leq R/2$, tutto ciò che è inviato dal mittente viene ricevuto dal ricevente con **un delay finito**. Se invece $\lambda_{in} > R/2$, il **link raggiunge la sua piena capacità**(R), e i pacchetti eccedenti vengono memorizzati nel buffer.
Dato che il **buffer è finito**, i **pacchetti accumulati infine verranno scartati**, e gli host saranno forzati a inviarli di nuovo (ancora più traffico).
Neanche un buffer infinito funzionerebbe, i pacchetti non saranno persi ma il **delay crescerà costantemente**: se superiamo la capacità per sempre, il delay raggiungerà l'infinito.
Ci sono **principalmente due approcci** per il congestion control:

1. **End-to-end congestion control**: questo è l'approccio standard di TCP dove la **congestione è dedotta dagli end system** basandosi solo sul comportamento osservato della rete (ad esempio, i pacchetti persi e il delay). I segmenti TCP persi sono presi come un indicatore della congestione della rete, e TCP decrementa il sending rate di conseguenza.
2. **Network-assisted congestion control**: è un approccio recente (opzionale) dove il livello di trasporto lavora in sinergia con il livello di rete. **I router forniscono feedback espliciti** al sender e/o al receiver riguardo lo stato della congestione della rete. A questo scopo vengono utilizzate le flag CWR e ECE.

Il TCP **si affida per lo più all'end-to-end congestion control**. L'approccio è che **ogni mittente adatti il suo sending rate** in base alla **congestione percepita della rete**. Ci sono 3 problemi principali da considerare:

- **Rate regulation**: come fa un sender TCP a regolare il suo sending rate?
- **Congestion detection**: come fa un sender TCP a percepire la congestione sul percorso tra se stesso e la destinazione?
- **Rate adjustment**: quale algoritmo dovrebbe usare il sender per cambire il suo send rate come funzione della congestione percepita?

Rate regulation Nel flow control di TCP il sending rate è regolato considerando lo spazio del buffer sul lato ricevente (receiver window). Nel congestion control di TCP **anche il sender tiene traccia di una congestion window** (cwnd):

$$i_{sent} - i_{ackd} \leq min(rwnd, cwnd)$$

Quindi, il rate è regolato incrementando/riducendo cwnd

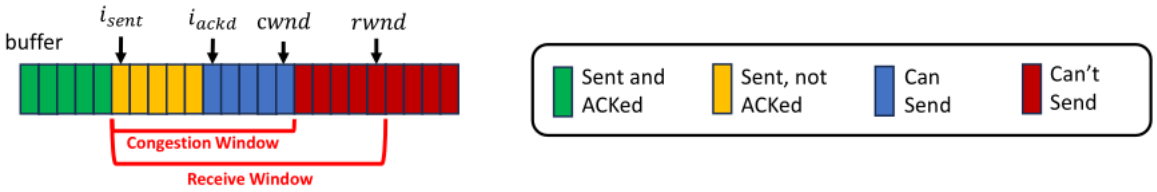


Figure 38: Rate regulation

Congestion detection Un sender TCP percepisce che c'è una congestione sul percorso tra se stesso e la destinazione controllando i **loss events**.
Un loss event si verifica **quando un viene raggiunto un timeout oppure vengono ricevute 3 ACK duplicate/errate**. Entrambi gli eventi stanno a significare che un pacchetto precedente **non è arrivato a destinazione**, probabilmente perché c'è stato un overflow nei dispositivi di rete.

Se non si verificano loss events, ovvero gli ACK dei segmenti arrivano come previsto, allora la **finestra di congestione può essere incrementata**. Altrimenti, se **avviene un loss event**, la **finestra di congestione deve essere decrementata**.

Rate adjustment Viene eseguito tramite il **bandwidth probing**: il rate è incrementato finché gli ACK arrivano correttamente (sondando la rete), quando avviene un loss event il rate viene decrementato. Questo processo è ripetuto in continuazione. TCP usa **l'algoritmo di congestion-control di Jacobson per regolare il rate** che include 3 fasi:

1. **Partenza lenta**: parte da 1 MSS/RTT incrementando il rate esponenzialmente.
2. **Incremento additivo**: incrementa il rate linearmente
3. **Ripristino veloce**: dimezza il rate invece di partire di nuovo lentamente e procede con un incremento additivo

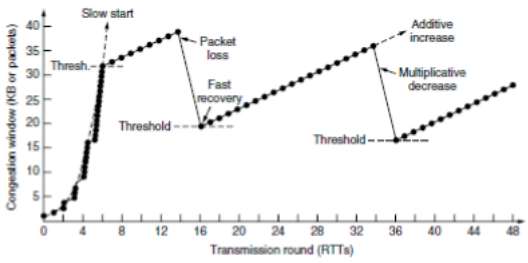


Figure 39: Comportamento tipico a dente di sega dell'algoritmo di Jacobsen

4 Network Layer

4.1 Dal livello di Trasporto a quello di Rete

Il compito del livello di rete è principalmente quello di consentire la comunicazione tra due host remoti:

- Consente all'**host mittente**: di prendere i segmenti dal livello di trasporto, di **incapsulare ciascuno di loro in un datagramma**, e di inviare quest'ultimo nella rete.
- Consente all'**host ricevente**: di ricevere i datagrammi dalla rete, **estrarne i segmenti del livello di trasporto**, e spedirli al livello di trasporto.

A questo livello, alcuni dei più importanti protocolli sono IP, DHCP, NAT ecc...
All'interno della rete ci sono i router, che sono **nodi che inoltrano i datagrammi al nodo adiacente** fino all'host di destinazione.
I router non eseguono né le applicazioni né i protocolli di rete. (implementano uno stack di protocollo "troncato").
L'atto di inoltrare un datagramma eseguito dai router è anche chiamato **hop**.

4.2 Il livello di rete in Internet

Il livello di rete è responabile principalmente della **consegna host-to-host**.
Quali **servizi potrebbero essere offerti** insieme alla consegna:

- **Consegna garantita**: un pacchetto inviato da un host sorgente arriverà all'host di destinazione.
- **Consegna garantita con un ritardo limitato:** ii pacchetti saranno consegnati entro uno specifico ritardo.
- **Consegna dei pacchetti in ordine**: i pacchetti arriveranno a destinazione nell'ordine in cui sono stati inviati.
- **Bandwidth minima garantita**: possibilità di specificare un bit rate minimo in modo che, se il rate dell'host mittente rientra in esso, allora tutti i pacchetti vengono infine consegnati all'host di destinazione.
- **Sicurezza**: encryption/decryption di tutti i datagrammi del sorgente/destinazione

In realtà, **nessuno di questi servizi** è generalmente offerto dalle reti. Al contrario, il livello di rete di Internet fornisce **solo un servizio**, il così detto **servizio best-effort**.
Il **servizio best-effort** (o consegna best-effort): la rete **fa del suo meglio** per consegnare un pacchetto dalla sorgente alla destinazione. I pacchetti non hanno la garanzia né di essere consegnati in ordine, né di essere congegnati. Non c'è garanzia sui ritardi o sulla bandwidth minima.
Nonostante la sua semplicità, il servizio best-effort, combinata ad una buona bandwidth **ha dimostrato di essere adeguato**.

4.3 I Router

Il ruolo principale del livello di rete è di eseguire la consegna host-to-host, cioè muovere pacchetti dall'host mittente all'host ricevente. Questo processo è eseguito dai **router** (nodi di rete) che sono **nodi speciali che hanno multipli link d'entrata e d'uscita**. I router forniscono due funzioni del livello di rete:

- **Forwarding:** quando un pacchetto arriva al link di input del router, esso **deve spostare il pacchetto al link d'uscita appropriato**. È anche possibile:
  - **Bloccare un pacchetto** e non farlo uscire dal router
  - **Scartare un pacchetto** se è expired (scaduto?)
  - **Duplicare un pacchetto** e inviarlo su multipli link d'uscita
  - **Modificare un pacchetto**
- **Routing**: per **decidere la strada o il path da seguire** per i pacchetti quando passano dal mittente al ricevente (attraverso **algoritmi di routing**)

4.3.1 Tipologie di router

I router possono essere molto **differenti** tra loro in base all'uso e alla tecnologia:

- Uso domestico o uso commerciale.
- Connessioni wired o wireless

Un'importante distinzione è fra edge e core router:

- **Edge router**: è un router che distribuisce pacchetti di dati all'interno di una o più reti (ad esempio connettere la rete con l'ISP). Questa è la tipologia più comune.
- **Core router**: usato per distribuire pacchetti all'interno della stessa rete piuttosto che su più reti. Questa tipologia è anche usata **sulle backbone di Internet** e il suo lavoro è di eseguire un pesante trasferimento di dati.

4.3.2 Data and Control planes

Il lavoro dei router è tipicamente implementato in un **processo a 2 step (o two-planes)**:

- **Forwarding**: l'azione **locale del router** di trasferire un pacchetto da un link di input al link di output appropriato. Questa è un'**operazione veloce** (tipicamente alcuni nanosecondi), e pertanto implementato nell'hardware.
- **Routing**: il processo **sulla rete** che determina il percorso che il pacchetto deve intraprendere per raggiungere la destinazione. Questo è un **processo più lento** (nell'ordine dei secondi), ed è spesso implementato via software

Al data plane abbiamo la **forwarding table** (tabella di inoltro): una tabella che associa link di output con possibili indirizzi di destinazione. Un router inoltra un pacchetto **esaminando il valore di uno o più campi** nell'header del pacchetto. Il valore memorizzato nella forwarding table indica **l'interfaccia del link di output** a cui il pacchetto verrà inoltrato.

Dato che più router possono essere attraversati prima di raggiungere la destinazione, il **contenuto della forwarding table** deve essere determinato **collezionando informazioni da diversi router**. Questo processo è implementato usando uno o più **tabelle di routing** (control plane) che eventualmente portano alla creazione della **forwarding table** (data plane)

La creazione delle tabelle di routing e forwarding può essere eseguita in due modi:

- **Decentralizzata** (o distribuita): possiamo avere ogni router dotato di un componente di routing che comunica con lo stesso componente di altri router (questo era l'approccio più usato).
- **Centralizzato**: possiamo avere un controller fisicamente separato dai router che distribuisce le tabelle di forwarding da utilizzare dai router.

Nel secondo caso, il controller remoto potrebbe implementare in un **data center remoto** (con alta affidabilità e ridondanza) e potrebbe essere gestito dall'ISP o qualcuno di terze parti.

Questo approccio è alla base del **software-defined networking** (SDN), dove la rete è "software-defined" perché il controller che elabora le tabelle di forwarding e interagisce coi router è un software centralizzato.

4.3.3 Componenti principali

Le principali componenti di un router sono:

- **Porte di input** (differenti dalle porte del livello di trasporto): sono le **interfacce fisiche di input** che operano con il livello sottostante (livello di collegamento) del link connesso, il loro ruolo è:
  - Consultare la tabella di forwarding (aka **lookup**) e di **preparare la switching fabric** per la porta di output da scegliere.
  - Di **inoltrare i pacchetti di controllo** al routing processor.
  - Il numero di porte di input **può variare tra le decine e le centinaia**
- **Switching fabric**: connette le porte di input alle porte di output.
- **Porte di output**: memorizzano i pacchetti ricevuti dallo switching fabric e trasmettono questi pacchetti sul link di uscita. Le porte possono anche essere **bidirezionali**
- **Routing processor**: esegue i protocolli di routing, conserva le informazioni di stato dei link, ed calcola la tabella di forwarding.

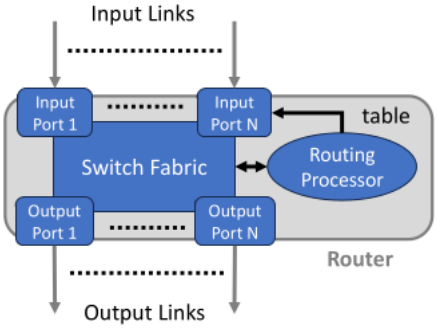


Figure 40: Componenti principali di un router

4.3.4 Forwarding

Il **ruolo della tabella di forwarding è di associare gli indirizzi IP alle porte di output**, in modo tale che i pacchetti possano essere inoltrati al link d'uscita giusto per essere trasmessi al nodo successivo.
Un indirizzo IP è un numero di 4 byte (32 bit) che usualmente vediamo in notazione decimale, ma può anche essere visto nella sua forma binaria.
All'interno di una forwarding table, gli IP sono associati ai numeri delle porte di output.

Quando un pacchetto è ricevuto dal link in una porta di input un'**operazione di lookup** viene eseguita. Ogni **porta ha una copia della tabella di forwarding** dal routing processor per evitare colli di bottiglia dovuti a continue invocazioni. Più **indirizzi potrebbero essere associati alla stessa porta**.

Example of a 4-ports table

Prefix of IP addresses	Link Interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

11001000 00010111 00010000 00000000
TO
11001000 00010111 00010111 11111111

11001000 00010111 00011000 00000000
TO
11001000 00010111 00011000 11111111

11001000 00010111 00011001 00000000
TO
11001000 00010111 00011111 11111111

Figure 41: Esempio di una tabella a 4 porte

Si noti che le entry potrebbero non essere mutualmente esclusive. Prendiamo come esempio le righe evidenziate in rosso nella tabella 41 e l'indirizzo ip 11001000 00010111 00011000 10101010, il router inoltrerà il

pacchetto all'entry con il match più lungo (**longest prefix matching rule**), in questo caso il link 1 "vince". Questa **procedura di lookup è tipicamente eseguita direttamente nell'hardware** per essere eseguita il più velocemente possibile.
Una volta che la porta di output di un pacchetto è stata determinata (dopo il lookup), **il pacchetto può essere inviato nello switching fabric**. In alcuni dispositivi, i pacchetti potrebbero anche essere temporaneamente messi in coda se altre porte di input stanno usando il fabric.

Quest'operazione a due step per **cercare un indirizzo IP di destinazione (match)** e **forwarding (action)** è chiamata match-plus-action ed è eseguita in diversi device di rete come:

- **Switch**: azioni simile al router
- **Firewall**: dove l'azione è **filtrare specifici pacchetti in arrivo**.
- **Network address translator** (NAT): dove l'azione è di **riscrivere il numero di porta** di uno specifico pacchetto in arrivo prima del forwarding.

4.3.5 Switching Fabric

In uno switching fabric, lo switching può essere effettuato in 3 modi:

1. **Switching via memory**: le porte sono **considerate come dispositivi I/O** che scrivono i pacchetti sulle celle di memoria. Dopodiché **il routing processor copia il messaggio** sulla porta di output come specificato dalla tabella di forwarding. Questo approccio è abbastanza lento (necessita di accessi in memoria) ed era più comune nei router iniziali (che erano dei computer standard).
2. **Switching via bus**: le porte di input trasferiscono il pacchetto direttamente alle porte di output su un **bus condiviso**, senza l'intervento del routing processor. Solo una porta alla volta può essere servita, ma questo approccio è spesso sufficiente per i router che operano in una piccola area locale.
3. **Switching via interconnection network**: le porte di input/output sono **connesse da una rete** avente dei cross-points che possono essere aperti e chiusi per reindirizzare i pacchetti. In questo approccio possono essere inoltrati più pacchetti in parallelo (metodo usato dai router moderni).

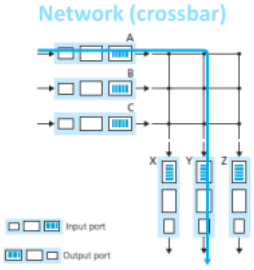


Figure 42: Esempio di switching con una rete interconnessa

4.3.6 Porte

Dato che lo switching richiede del tempo, le porte di input e output **hanno delle code per memorizzare temporaneamente i pacchetti**. L'**estensione di queste code non è fissata**, può dipendere dal traffico, la velocità dello switching fabric, ecc...

In generale, **i pacchetti possono essere ricevuti/inviati più velocemente/lentamente dello switching**, quindi possono accumularsi nei buffer di input/output che potrebbero quindi andare incontro ad overflow.

Ad esempio: assumiamo di avere **un router con N porte di input ed N di output**, che **ogni porta sta ricevendo pacchetti** allo stesso tempo e che tutte le linee di input e output **hanno la stessa velocità** di R_{line} pacchetti al secondo. Consideriamo ora lo **scenario peggiore** dove **tutti i pacchetti devono essere inoltrati sulla stessa porta**. Se lo **switching fabric** ha un rate uguale a R_{switch}, abbiamo:

- Se $R_{switch} \cong NR_{line}$, **l'accodamento sulle porte di input è trascurabile**, dato che tutti i pacchetti vengono inoltrati in tempo dal fabric, ma **l'accodamento sulle porte di output è significativo** dato che i pacchetti in arrivo sono N volte maggiori del rate R_{line} del link di output
- Se $R_{line} < R_{switch} < NR_{line}$, c'è un **accodamento su entrambe le porte** visto che le porte di input devono aspettare lo switching fabric mentre le porte di input devono aspettare il link.
- Se $R_{switch} \cong R_{line}$, l'**accodamento sull'output è trascurabile**, ma **l'accodamento sulle porte di input è significante**.

4.3.7 Scheduling dei pacchetti

È ragionevole avere **più pacchetti** (potenzialmente da più porte di input) che devono essere inoltrati **ad una singola porta di output**.
Ci sono 3 approcci più famosi:

- **Firt-come-first-served** (FCFS, oppure **first-in-first-out**, FIFO), che è un **semplice approccio basato sul tempo**
- **Priority queuing**, che è basato sull'importanza dei pacchetti
- **Round-robin queueing**, dove i pacchetti sono divisi in classi (basate sulla priorità) ed ogni classe è servita a turni.

FIFO Se il link di **output è occupato** i pacchetti in arrivo alle porte di output devono essere **salvate nel buffer**.
Se **non c'è abbastanza spazio nel buffer** per conservare i pacchetti in arrivo, dobbiamo affidarci sulla **politica di packet-discarding**. Una politica tipica è di **scartare i pacchetti arrivati recentemente (drop-tail)**, ma in altri approcci più sofisticati **anche i pacchetti già salvati nel buffer possono essere rimossi** per fare spazio a quelli in arrivo.
Un pacchetto viene rimosso dalla coda **solo se è stato completamente trasmesso** sul link d'uscita.

Priorità I pacchetti che arrivano al link di output sono classificati (ad esempio tramite i numeri di porta TCP/UDP) in classi di priorità al momento dell'arrivo nella coda. Un **operatore di rete può configurare una coda** in modo tale che specifici pacchetti possano ricevere la priorità su altri pacchetti. **Ogni classe di priorità potrebbe avere la sua coda**. La scelta tra i pacchetti della stessa classe di priorità è tipicamente fatta con un approccio FIFO.

Round-robin In questo approccio i pacchetti vengono ordinati in classi che vengono **alternate** e non selezionate in base alla priorità. Un'implementazione comune è chiamata **weighted fair queuing** (WFQ) dove i pacchetti in arrivo sono classificati e accodati nelle rispettive aree di attesa.
Ciascuna classe è quindi associata ad uno specifico peso che detta il **grado con cui la classe viene selezionata** invece delle altre.

4.4 Internet Protocol (IP)

L'**Internet Protocol (IP)** è il protocollo del livello di rete usato per garantire la consegna host-to-host. Ci sono di versioni di IP attualmente in uso che sono la **versione 4 (IPv4)**, la più usata e la più comune, e la **versione 6 (IPv6)**,la versione più nuova, proposta per rimpiazzare IPv4. La **più importante funzionalità** fornita dal protocollo è quella di **identificare gli host** su una rete.
L'**IP addressing** è il processo di assegnamento di indirizzi IP ai dispositivi. È un'operazione **cruciale** per le funzionalità della rete ed è parecchio **complessa** in Internet.

4.4.1 Indirizzi IPv4

Gli indirizzi IP sono tipicamente scritti nella così detta **notazione decimale puntata**, dove ogni byte dell'indirizzo è scritto nella sua forma decimale e separata da un punto dagli altri byte dell'indirizzo.
Ogni dispositivo in Internet deve avere un **indirizzo IP univoco globalmente** Dato che ogni indirizzo è di 32 bit, c'è un totale di 2^{32} possibili indirizzi IP.

Gli host e i dispositivi di rete **sono connessi tramite link** (wired o wireless). Un host tipicamente ha un singolo link nella rete, mentre un dispositivo di rete, come un router, hanno più link. Il confine tra host e il link fisico è chiamato **interfaccia**. Poiché ogni dispositivo è capace di inviare e ricevere datagrammi IP, IP richiede che **ogni interfaccia abbia il proprio indirizzo IP**. Quindi, un **indirizzo IP è tecnicamente associato all'interfaccia** e non all'host o router che contiene tale interfaccia.

4.4.2 Subnetting

Assegnare indirizzi IP a differenti interfacce non è banale. Non sarebbe saggio assegnarli in modo casuale per diverse ragioni:

- Dato che IP non da indicazioni sulla locazione, **non sapremmo dove trovare gli host**.
- Potremmo aver bisogno di **tabelle di forwarding enormi all'interno dei router**.

L'indirizzamento della rete richiama quello della telefonia standard: le **reti sono gerarchicamente divise in sottoreti** (subnet) dotati di differenti prefissi.
Un indirizzo IP è quindi diviso in 2 parti:

- La **prima porzione** (quella più a sinistra) identifica la sottorete a cui il nodo è connesso.
- La **seconda porzione** (quella più a destra) identifica la singola interfaccia.

Il **numero di bit appartenenti ad ogni porzione non è fissato.**

4.4.3 Subnet Mask

Nello specifico, per distinguere la parte della subnet dalla parte dell'interfaccia, un indirizzo IP è associato ad una **subnet mask** che specifica quali bit dell'indirizzo appartiene alla parte della subnet. Una rappresen-

	Dotted-decimal	Binary
IP address	193.32.216.9	11000001 00100000 11011000 00001001
Subnet Mask	255.255.255.0	11111111 11111111 11111111 00000000

Figure 43: Esempio di subnet mask

tazione comune per rappresentare le mask è la **slashed notation**, ad esempio: 193.32.216.0/24 rappresenta l'IP della sottorete, dove /24 indica che i 24 bit più a sinistra sono dedicati alla sottorete.
Per capire se 2 host fanno parte della stessa subnet, dobbiamo semplicemente vedere se **i loro prefissi di sottorete sono uguali**.

Gli host **appartenenti a diverse sottoreti possono comunque comunicare**, ma l'amministratore

di rete potrebbe decidere di trattare i messaggi che entrano o escono da determinate sottoreti in modo diverso.
Il dispositivo che interconnette diverse sottoreti è il router. Per definizione, un **router ha più porte fisiche**, queste potrebbero essere correlate a **differenti interfacce di rete** che hanno differenti indirizzi IP.
I router si trovano spesso ai margini di multiple sottoreti e gestiscono i messaggi che entrano e escono (**gateway**). Grazie alla loro posizione privilegiata, i **router spesso forniscono diversi servizi** oltre che il semplice routing:

- Mascheramento IP (NAT)
- Assegnamento IP dinamico (DHCP)
- Protezione (Firewall)

4.4.4 Ifconfig

Il comando per visualizzare il proprio indirizzo su Linux è Ifconfig (l'equivalente per windows è ipconfig). Inoltre fornisce la **lista di tutte le interfacce di rete della macchina insieme alle loro configurazioni di rete**

4.4.5 Internet Addressing

L'idea di **spezzare grandi reti in reti più piccole è particolarmente importante** su Internet dove miliardi di dispositivi devono essere connessi. Su internet gli **indirizzi devono essere accuratamente assegnati** per evitare problemi come:

- La **tabella di forwarding** potrebbe diventare troppo grande.
- Potremmo avere interfacce diverse con lo **stesso indirizzo**.
- Potremmo **ritrovarci a corto di indirizzi**.

L'approccio è di dividere gli indirizzi di Internet **fornendo sottoreti alle organizzazioni** (come ISP, aziende, istituzioni, ecc.). Ci sono 2 modi:

- **Classful addressing** (più vecchio, non più usato)
- **Classless addressing**(attualmente utilizzato)

4.4.6 Classful Addressing

Nel **classful addressing** Internet viene diviso in classi in base ad una specifica divisione: Se ad un'organizzazione

	Format	Example	IPs per network
Class A	a.b.c.d/8	10.X.X.X	> 16 million
Class B	a.b.c.d/16	10.10.X.X	65535
Class C	a.b.c.d/24	10.10.10.X	254

Figure 44: Classi nel classful addressing

servono 300 IP, **gli viene assegnata la classe B**. Però verrebbero così sprecati 65235 indirizzi IP.

4.4.7 Classless Addressing

L'approccio moderno è più flessibile ed è chiamato **Classless InterDomain Routing (CIDR)**. Qui, ad un organizzazione verrebbero assegnati indirizzi di rete di qualsiasi forma:

a.b.c.d/X

Ora, se servissero 300 indirizzi IP verrebbe assegnato ad esempio 241.115.2.0/23 che **fornisce 512 indirizzi IP e verrebbero sprecati solo 212**.

Negli indirizzi che usano CIDR abbiamo:

- La parte di rete dell'indirizzi chiamato **prefisso**.
- Il set di indirizzi IP riservati all'organizzazione è chiamato **blocco**.

C'è anche la possibilità di creare **sottoreti interne** in un blocco CIDR con il seguente risultato:

Figure 45: Esempio di sottorete in un blocco CIDR

4.4.8 Address Aggregation

L'indirizzamento basato sui prefissi è molto utile per i dispositivi che connettono diversi prefissi. È possibile per un router ricordare solamente i prefissi. Quando un messaggio con uno specifico prefisso, viene inoltrato ad un router più specifico e così via. Questo approccio è chiamato **address aggregation**.

4.4.9 Ottenere un blocco

Per ottenere un blocco di indirizzi IP da usare in un'organizzazione ci sono due modi:

- Si può **contattare un ISP**, che fornirà indirizzi da un blocco più largo di indirizzi che è stato già allocato
- Si può chiedere all'ICANN, che è un'organizzazione globale non a scopo di lucro che ha la responabilità di gestire lo spazio degli indirizzi IP.

4.4.10 Datagramma IPv4

Il pacchetto al livello di rete è chiamato **datagramma**. I campi chiave di un datagramma IPv4 sono:

- **Numero di versione** (4bit) specifica **la versione dell'IP** (ad esempio IPv4 o IPv6)
- **Header length** (4bit): **dimensione dell'header** (non è fissa, le opzioni sono di dimensione variabile), usata per conoscere dove parte il payload (nessuna opzione sta a significare header di 20 byte).
- **Type of service** (TOS, 8bit): identifica le **proprietà** specifiche del datagramma (come real-time/non-real-time). Alcuni tipi sono **definiti dall'amministratore di rete**.
- **Datagram length** (16 bit): lunghezza in **byte del datagramma** (header+dati). I datagrammi raramente sono più grandi di 1500 byte (con un massimo di 65535).
- **Identifier** (16 bit): un numero progressivo che identifica unicamente il datagramma. (usato nella frammentazione)
- **Flags and fragmentation offset** (3+13 bit): proprietà e offset del frammento. (usato nella frammentazione)

- **Time-to-live** (TTL, 8 bit): garantisce che i datagrammi non **circolino per sempre nella rete**. Questo campo viene decrementato di uno ogni volta che il datagramma è processato da un router. Se il campo TTL raggiunge 0, il router scarta quel datagramma.
- **Upper-layer Protocol** (8 bit) indica il **protocollo del livello di trasporto** a cui il payload di questo datagramma IP dovrebbe essere passato.
- **Header checksum** (16 bit): per l'error detection. Qui il checksum è calcolato come complemento a 1 della somma dei 2 byte dell'header. **I router verificano questo valore** e **i datagrammi errati vengono scartati**. Si noti che il checksum deve essere **ricalcolato e salvato di nuovo a ogni router** a causa dei campi TTL e option che possono cambiare.
- **Source and destination IP addresses** (32+32 bit).
- **Options** (non fissati): consentono all'header IP di essere esteso (**funzionalità aggiuntive**). Il campo opzioni fornisce un certo grado di complessità (dimensione sconosciuta), esso non è presente in IPv6.
- **Data** (non fissata) contiene il **messaggio attuale (payload)**, tipicamente nella forma di un segmento TCP/UDP.

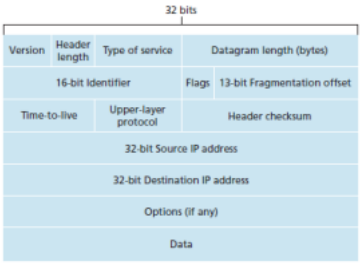


Figure 46: Struttura del datagramma IP

4.4.11 Frammentazione di IPv4

Il primo problema con il datagramma IP è che esso può essere **frammentato al livello link**. Alcuni protocolli del livello link trasportano datagrammi di **differenti dimensioni**.
I datagrammi IP sono tipicamente incapsulati in frame del livello link per essere trasportato da nodo a nodo. La massima quantità di dati che un frame può trasportare è il **maximum transmission unit** (MTU).
Un router che interconnette 2 link aventi differenti MTU **potrebbe ricevere un datagramma ip dal link di input che non si adatta al link di output**.
Il **router deve dividere il payload** del datagramma IP in 2 o più datagrammi IP più piccoli (frammenti). **I frammenti sono quindi incapsulati in frame del livello di link** e inoltrati al link d'uscita.

Quando un router **frammenta un datagramma**:

- **Copia lo stesso identificatore, indirizzo sorgente e di destinazione** in un nuovo frammento
- Imposta il campo **fragment offset** di tutti i segmenti con **numeri progressivi**.
- Imposta la **flag dell'ultimo segmento a 1** (per segnalare che i frammenti sono finiti).

Chiaramente, i **frammenti devono essere riassemblati** prima di raggiungere il livello di trasporto alla destinazione, dato che sia TCP che UDP si aspettano di ricevere segmenti completi dal livello di rete.
I designer di IPv4 hanno pensato che **riassemblare i datagrammi nei router** avrebbe potuto introdurre complicazioni importanti nel protocollo e **impattare sulle prestazioni dei router**. In IPv4 quindi il datagramma **viene riassemblato negli end system**:

- Se più datagrammi che hanno lo **stesso indirizzo e identificatore** vengono ricevuti, significa che il datagramma originale è stato frammentato
- L'host deve **ricreare il datagramma originale** dai frammenti.

4.4.12 Assegnazione degli indirizzi IP

In un blocco di indirizzi, **gli indirizzi IP devono essere assegnati** alle interfacce. L'**amministratore di sistema** può configurare gli indirizzi IP in 2 modi:

- **Manualmente**: assegnando gli indirizzi IP host per host.
- **Automaticamente**: la rete autonomamente assegna gli indirizzi IP liberi agli host in arrivo.

Il secondo approccio (il più comune) può essere eseguito usando il **Dynamic Host Configuration Protocol (DHCP)**. In aggiunta, **DHCP fornisce all'host le informazioni necessarie per entrare nella rete**, come la subnet mask, l'indirizzo del default gateway, e l'indirizzo del server DNS locale. Il **DHCP è plug-and-play**, ed è tipicamente usato nelle nostre case.

4.4.13 DHCP

Il DHCP è un **protocollo client-server**: un nuovo host in arrivo (client) si connette al server DHCP per ricevere informazioni della rete. **Il servizio DHCP può essere fornito da un computer o dal router stesso**. Formalmente il DHCP è un **protocollo del livello applicazione**. (I router, dato che implementano una versione ridotta della pila, non dovrebbero poter fornire questo servizio ma nel pratico, grazie alla loro posizione centrale nella rete, possono fornirlo)

Aggiungere un nuovo host è un processo di 4 passi:

1. **DHCP server discovery**: il nuovo host invia in **broadcast un messaggio di DHCP discovery (UDP sulla porta 67)** per trovare il server DHCP.
   - IP di destinazione: 255.255.255.255 (broadcast)
   - IP sorgente: 0.0.0.0 (questo host)
   - Transaction ID casuale
2. **DHCP server offer**: dato che ci possono essere più server DHCP nella rete, quando un messaggio di discovery viene ricevuto, un **server risponde con un messaggio di broadcast (sulla porta 68) offrendo una possibile configuazione di rete** (presentando l'yiaddr ovvero Your Internet ADDRess).
   - IP di destinazione: 255.255.255.255 (broadcast)
   - IP sorgente: ip del DHCP server (ad esempio 223.1.2.5)
3. **DHCP request**: il nuovo client seleziona l'offerta accettata rispondendo con **l'echo dell'offer message**.
4. **DHCP ACK**: **messaggio ACK** finale che conferma la transazione.

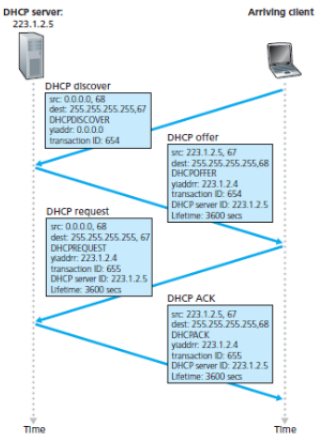


Figure 47: Aggiunta di un nuovo host

In questo caso tutto funziona senza problemi perché il server DHCP e l'host in arrivo **sono sulla stessa subnet**. Se un server si trova in una subnet diversa, abbiamo bisogno di un **DHCP relay agent** per inoltrare i messaggi DHCP (ruolo eseguibile dal router).
In linux si può usare il comando ifconfig in combinazione col comando route per vedere la configurazione della rete fornita dal DHCP.

4.4.14 Visibilità e NAT

Su Internet ci sono miliardi di dispositivi connessi che devono essere **associati ad un univoco indirizzo IP per essere raggiunti da chiunque**.
È davvero necessario **avere tutti i dispositivi visibili a chiunque?**. Ci sono **evidenti problemi nell'avere tutti i dispositivi raggiungibili** dall'esterno delle reti locali:

- Gli indirizzi IP **sono finiti**.
- Se un dispositivo è localmente connesso **è spesso non desiderabile esporli tutti su internet**
- Gli**amministratori locali dovrebbe essere pienamente consapevole del blocco IP sovrastante** per assegnare indirizzi non utilizzati.

Il servizio di **Network Address Translation (NAT)** consente di **rimappare gli indirizzi IP** dei pacchetti in una rete in altri indirizzi (**mascheramento della rete**). Viene eseguito usando una **tabella di traduzione** associando **più indirizzi IP/porte locali (della lan) in un indirizzo IP/porta globale (della WAN)**. Questo servizio può essere offerto dai **router**.
La rete locale mascherata è anche chiamata **rete privata o regno con indirizzi privati (?)**. Gli indirizzi IP privati sono validi sono all'interno della rete privata.

4.4.15 Ping e nmap

Per verificare se un host è raggiungibile su una rete, possiamo usare il comando ping (Packet Internet Groper) è una utility basata sul protocollo di trasporto **ICMP (Internet Control Message Protocol)** che invia un pacchetto "echo request" ad un host che automaticamente risponde con un messaggio "echo reply". Il comando Ping fornisce anche il **tempo trascorso tra richiesta e risposta**, misurando l'RTT.

Oltre alla scansione delle porte, il comando nmap può anche essere usato per **mappare/scansionare i dispositivi sulla rete**. Si **affida al ping** per scoprire i dispositivi nella rete.

4.4.16 Indirizzi IP riservati

Dato che **gli amministratori locali dovrebbero essere liberi di organizzare una rete privata** senza interferenze, per convenzione ci sono alcuni **blocchi IP riservati** che l'ISP o l'ICANN non può assegnare a nessuno. Alcuni esempi più noti sono:

- 10.0.0.0-10.255.255.255 (riservato alle **reti private**)
- 192.168.0.0-192.168.255.255 (riservato alle **reti private)**
- 127.0.0.0-127.255.255.255 (indicano la macchina attuale, per il **loopback**, tipicamente è usato 127.0.0.1)
- 0.0.0.0 (indica **la rete corrente**)
- 255.255.255.255 (indica il **broadcast**)

La necessità di riservare indirizzi IP è data dal fatto che **indirizzi privati possono essere gli stessi di quelli privati**, quindi il routing nella rete potrebbe essere ambiguo.

Se per esempio vogliamo creare una **nuova sottorete locale (LAN) in una pre-esistente rete (WAN)**, allora l'**amministratore di rete della WAN** ci deve fornire le seguenti informazioni:

- L'**indirizzo IP della WAN**: es. 150.100.0.0/16

- L'**indirizzo IP del gateway**(che porta ad una WAN esterna o internet): es. 150.100.50.1
- Un indirizzo IP **libero per la nostra rete**: es. 150.100.50.10

Ora dobbiamo configurare il nostro router e i nostri dispositivi per impostare la nuova LAN:

- Stiamo considerando un router **NAT-enabled**
- L'**indirizzo IP della nuova LAN** sarà: es. 192.168.1.0/24 (riservato alla rete locale)

Il nostro **router locale ha 2 interfacce:** una per il lato WAN e un'altra per il lato LAN. Cominciamo configurando le **impostazioni del lato WAN**. Sul router dobbiamo specificare i **seguenti parametri**:

- L'indirizzo IP del router nella WAN (può essere statico o dinamico)
- La subnet mask della WAN
- Il gateway
- Uno o più DNS

Ora configuriamo le **impostazioni del lato LAN**. Dobbiamo dare un **indirizzo IP** all'interfaccia del lato LAN del router. Se il nostro router fornisce anche **il servizio DHCP** dobbiamo allora specificare:

- La **Address Pool** (per gli host DHCP)
- Il **Lease time**: un timeout dell'assegnazione dell'indirizzo IP, dopo il quale, l'indirizzo IP può essere riutilizzato.
- Il **Gateway** e il **DNS** che deve essere comunicato agli host.

Per configurare invece la connessione su un nuovo host ci sono due differenti metodi da scegliere:

- **Automatic (DHCP)**: usa DHCP per configurare automaticamente la connessione.
- **Manuale**: imposta la connessione inserendo manualmente la configurazione

In una **configurazione manuale** abbiamo bisogno di sapere la configurazione della rete (e gli indirizzi IP disponibili). Dobbiamo specificare le informazioni principali:

- Indirizzo IP dell'host (statico/fisso)
- La subnet mask
- Il gateway
- Il DNS (uno o più)

4.4.17 IPv6

All'inizio degli anni '90, l'Internet Engineering Task Force (IETF), **cominciò lo sviluppo di un successore del protocollo IPv4** per affrontare la **carenza di indirizzi IPv4**. IPv6 fu progettato per **garantire più indirizzi disponibili**, ma fu anche un opportunità per **aggiornare alcuni aspetti di IPv4**, basandosi sull'esperienza accumulata.
Non si è certi di **quando esattamente tutti gli indirizzi IPv4 disponibili saranno esauriti**. E invece **IPv5**? Fu proposto nel 1979 a si basava sul protocollo sperimentale Stream Protocol (ST). Si affidava ancora ad indirizzi a 32 bit, quindi fu scartato principalmente per la carenza degli indirizzi.

IPv6 ha i seguenti vantaggi:

- **Capacità di indirizzamento estesa**: da 32a 128 bit, ciò assicura che non si rimarrà mai senza indirizzi. (Sono disponibili 2^128 indirizzi circa)
- **Un header di lunghezza fissata di 40 byte**: alcuni campi dell'IPv4 col tempo sono stati scartati o resi opzionali

- **Flow labeling**: i pacchetti da alcune applicazioni (come audio/video streaming) possono essere raggruppati in un unico flusso con una specifica identificazione.

Il Datagramma di IPv6 è composto dai seguenti campi:

- **Version**(4 bit): identifica il numero versione del protocollo IP.
- **Traffic class**(8bit): come il campo TOS nell'IPv4, può essere usato per dare **priorità ai datagrammi**.
- **Flow label**(20 bit): identifica un **flusso** di datagrammi.
- **Payload length**(16 bit): il **numero di byte del payload** (esclusi i 40 byte dell'header)
- **Next header**(8 bit): identifica il **protocollo del livello sovrastante** con cui il contenuto (campo data) di questo datagramma deve essere consegnato.
- **Hop limit**(8bit): specifica il **time-to-live** come nel campo TTL di IPv4 (decrementato ad ogni inoltro).
- **Indirizzi sorgente e destinazione** (128+128 bit).
- **Data** (variabile): la porzione di **payload** del datagramma.

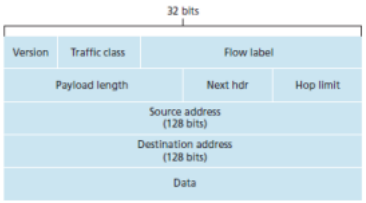


Figure 48: Struttura del datagramma IPv6

I campi che mancano dall'IPv4 sono (vedi figura 46):

- **Campi relativi alla frammentazione:** IPv6 **non consente la frammentazione e il riassemblaggio immediato nei router**, ma solo negli host. Se un datagramma IPv6 ricevuto da un router è troppo grande per essere inoltrato, il **router semplicemente scarta tale datagram e invia un errore "Packet Too Big"** al mittente. Il mittente può rinviare i dati, usando un datagramma IP più piccolo.
- **Header checksum**: impiega tempo nei router, ed è ridondante dato che **il livello di link tipicamente esegue il checksum sull'intero pacchetto**.
- **Options**: non sono più parte dell'header standard di IP, **alcune opzioni possono essere specificate nel campo next header**.

C'è un solo grande problema con IPv6, cioè che **non è retro-compatibile**. I sistemi che utilizzano IPv6 non sono in grado di gestire i datagrammi IPv4. Allora come verrà adottato IPv6 da Internet?

- **L'approccio flag day:** in una data prestabilita **tutte le macchine su Internet verranno spente e aggiornate da IPv4 a IPv6**. Un approccio simile fu usato quando fu introdotto TCP ma con la situazione odierna è una soluzione impensabile.
- **L'approccio del tunnelling**: l'uso di specifici router (i tunnel) incaricati di **mappare datagrammi IPv4 in datagrammi IPv6 e viceversa** in un modo più o meno trasparente. È l'approccio più realistico, già in uso.

L'adozione di IPv6 era inizialmente lenta, ma sta accelerando. Google riporta che circa il **45% dei client nel 2023 che accedono ai servizi Google** usano IPv6.

4.5 Routing

4.5.1 Introduzione

Un router, quando riceve un pacchetto, può **eseguire 3 azioni**:

- **Inoltrare il pacchetto** così com'è ad un adiacente
- **Inoltrare il pacchetto modificato/rimpiazzato** (e.g.,frammentazione, mascheramento, ecc.)
- **Scartare il pacchetto**

Ci sono **differenti tipi di router**, alcuni sono molto semplici e sono usati dagli amministratori locali (ad esempio per connettere LAN e WAN), altri sono usati dagli ISP per consentire ai pacchetti di essere inoltrati verso il giusto dispositivo nell'intero Internet.
Come abbiamo visto, **i router sono dotati di tabelle di forwarding** che specificano la strategia locale di inoltro. Il modo con cui queste tabelle sono create può essere centralizzato o decentralizzato. I router non sono a conoscenza, almeno all'avvio, del resto della rete.

Per decidere dove i pacchetti dovrebbero essere inoltrati, i router utilizzano gli **algoritmi di routing**, che determinano percorsi buoni (ovvero sequenze di router) dai mittenti ai destinatari.
Tipicamente, **un percorso "buono" è quello che ha il costo minimo** (percorso più corso o più veloce), nel mondo reale **ci sono anche problemi di policy** da considerare (ad esempio se delle regole specificano che un'organizzazione può parlare o meno con un'altra ecc.).
Il **problema del routing è cruciale per le reti**, ma in generale il problema di trovare "percorsi buoni" su una rete (o grafo) è particolarmente rilevante non solo nelle reti (altri esempi sono i videogiochi, guida autonoma, robotica).

4.5.2 Flooding

L'approccio più semplice è il **flooding**: ogni router inoltra tutti i pacchetti in arrivo a tutti i link ad eccezione del di quello da cui il pacchetto è stato ricevuto.

- Tutti i percorsi sono esplorati (incluso quello ottimale), quindi il **pacchetto sarà sicuramente consegnato** se un percorso esiste.
- I pacchetti "che non hanno una meta" raggiungeranno il loro **hop limit** e verranno scartati.
- È **estremamente robusto** a causa della ridondanza dei pacchetti generati.

Il flooding può essere usato in situazioni specifiche dove la ridondanza è necessaria (come nel caso del broadcast dei messaggi) ma è **impraticabile nelle reti larghe e ad alte prestazioni** (come Internet).
Per le reti tipiche sono necessari algoritmi più sofisticati.

4.5.3 Formulazione del problema

Possiamo **formalizzare** la nostra rete come un grafo $G = (N, E)$ dove:

- N è l'insieme dei **nodi** (i router della nostra rete)
- $E \subseteq N \times N$ è un insieme di **archi** (link fisici) rappresentati come coppie di nodi.

Un **arco ha anche un valore rappresentante il suo costo**, che rispecchia la lunghezza fisica del link, la sua velocità, oppure il costo monetario associato a quel link.
Possiamo formalizzare il **costo** come una funzione $c : N \times N \to \mathbb{R}$. Quindi data una coppia generica di nodi $(x, y) \in N \times N$:

- $\forall x \in N$, allora $c(x, x) = 0$
- Se $(x, y) \notin E$, allora $c(x, y) = \infty$
- Se $(x, y) \in E$, allora anche $(y, x) \in E$ (grafo non orientato) e $c(x, y) = c(y, x)$

Seguendo questa formulazione, un **percorso** π in G è una **sequenza di nodi** $\pi = (x_1, x_2, \ldots, x_n)$ in modo che tutti i **nodi adiacenti sono connessi**, formalmente:

$$\forall x_i, x_{i+1} \in \pi, (x_i, x_{i+1}) \in E$$

Il **costo totale del percorso** è la somma dei costi di tutti gli archi:

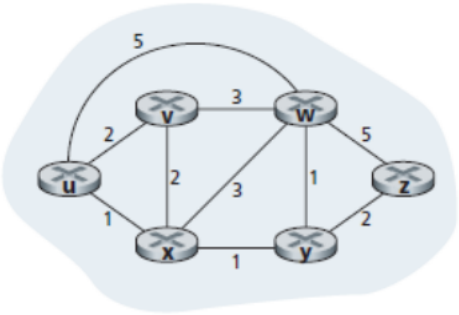
$$c(\pi) = \sum_{i=1}^{n-1} c(x_1, x_{i+1}$$

Ora chiamiamo $P(x, y)$ l'**insieme di tutti i possibili percorsi** che connettono x e y, possiamo definire il **percorso migliore** come il percorso $\pi^* \in P(x, y)$ che ha il costo minimo:

$$\forall \pi \in P(x, y), c(\pi^*) \leq c(\pi)$$

Nel seguente esempio abbiamo un grafo composto di 6 nodi:

- N=u,v,w,x,y,z
- E=(u,w),(u,v),(w,x),...
- c(u,w)=5, c(u,v)=2, c(u,x)=1, ...



Il miglior percorso π^* tra i nodi x e z è:

- $\pi^* = (x, y, z)$
- $c(x, y) = 1$, $c(y, z) = 2$
- $c(\pi^*) = 3$

Gli algoritmi capaci di trovare il percorso ottimale π^x tra 2 nodi sono chiamati **shortest path algorithms**. Nel caso delle reti, siamo interessati nel **trovare tale percorso ottimale** non solo per due nodi, ma **per tutte le possibili coppie di nodi**.
Un algoritmo di routing **converge** quando il percorso più corto per tutte le coppie di nodi nella rete viene trovato.
Nelle reti **ci sono 2 famiglie di algoritmi** che sono tipicamente usati:

- Distance-Vector
- Link-State

4.5.4 Algoritmo Distance-Vector

L'algoritmo **Distance-Vector** (DV) è un approccio decentralizzato che sfrutta la **conoscenza locale** della rete combinata con la comunicazione dei nodi per trovare i path più brevi. È basato sull'**algoritmo di Bellman-Ford** che viene usato per calcolare i percorsi.
Qui **ciascun nodo riceve alcune informazioni** da uno o più dei suoi adiacenti diretti, esegue un calcolo, e infine **distribuisce i risultati** del suo calcolo ai suoi vicini.
Questo algoritmo è **asincrono** in quanto non richiede che tutti i nodi siano coordinati con gli altri.

Chiamiamo $d^*(x, y)$ il costo (distanza) del percorso migliore dal nodo x al nodo y. Dall'**equazione di Bellman** possiamo derivare la seguente:

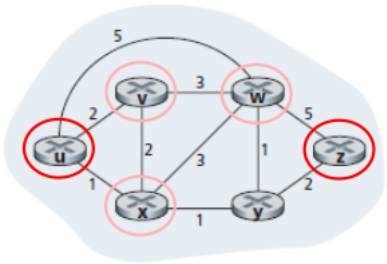
$$d^*(x, y) = \min_{(x,a)\in E}\{c(x, \alpha) + d^*(\alpha, y)\}$$

È abbastanza intuitivo, **se x è parte del percorso migliore, ci deve essere un nodo adiacente α che si trova sullo stesso percorso migliore**, ed il costo di questo percorso migliore è dato dal costo di raggiungere α più il percorso migliore tra α e y.

Se siamo in grado di trovare tale α **possiamo solo aggiungerlo nella tabella di forwarding** e inoltrarlo a tutti i pacchetti diretti ad y. Per far ciò, **dobbiamo avere una stima di $d^*(\alpha, y)$** per tutti gli adiacenti (il distance vector o vettore dele distanze).

Proviamo ora con un esempio reale. Considerando u e z, il percorso migliore è:

- $\pi^* = (u, x, y, z)$
- $d^*(u, z) = c(\pi^*) = 4$



Ora, **per tutti i nodi lungo il percorso ottimale l'equazione di Bellman deve essere mantenuta**. Vediamo cosa accade per il nodo u.

$$d^*(u, z) = \min_{(u,\alpha)\in E}\{c(u, \alpha) + d^*(\alpha, z)\}$$

In questo esempio u ha 3 adiacenti (w,v e x):

- Per $\alpha = w$ allora $c(u, w) = 5$ e $d^*(u, z) = 8$.
- Per $\alpha = v$ allora $c(u, v) = 2$ e $d^*(u, z) = 5$.
- Per $\alpha = x$ allora $c(u, x) = 1$ e $d^*(u, z) = 4$.

Il nodo x **minimizza l'equazione, ed è il prossimo sul percorso ottimale** π^*.

Possiamo **iterare questo processo sul prossimo nodo** del percorso (che è x) e potremo vedere che **l'equazione di Bellman vale**.
Quindi, se abbiamo la il distance vector d^* per tutti i nodi, **possiamo facilmente usare l'equazione di Bellman per calcolare il next hop** nel percorso (tabella di forwarding).

Di seguito lo pseudocodice dell'algoritmo DV:

```
DistanceVector(x)
 for all nodes v
  if v is a neighbor of x then
    Dx(v) = c(x,v)
  else
    Dx(v) = ∞
 for all neighbors w and destinations y
  Dw(y) = ?
 send initial Dx(·) to all neighbors

 repeat
  if cost of a neighbor updated then
   for all nodes y
    Dx(y) = minv{c(x,v) + Dv(y)}
   if Dx(y) changed for any destination y then
    send updated Dx(·) to all neighbors
 until false //forever
```

Usiamo la **struttura dati** $D_w(v)$ per memorizzare i distance vector per ogni nodo w che punta i nodi v.
Durante la fase di **inizializzazione** solo le **distanze dagli adiacenti vengono impostate**.
Durante la fase **online**:

- Gli aggiornamenti dagli adiacenti vengono ricevuti.
- I distance vector vengono **aggiornati** (equazione di Bellman).
- I cambiamenti locali sono **comunicati** ai nodi adiacenti.

Per chiarire questo processo torniamo sul percorso tra i nodi u e z (vedi 4.5.4):

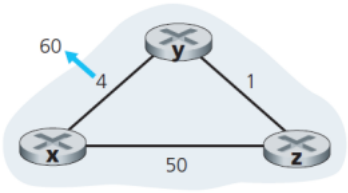
1. All'inizio, **u conosce solo i suoi vicini**. Il costo per arrivare a z è infinito ($D_u(z) = \infty$).
2. Il nodo **y si attiva** e scopre un percorso migliore per z che sarebbe il link diretto y-z con il costo di 2 ($D_y(z) = c(y, z) = 2$.Il nodo y quindi comunica la notizia a x.
3. Il nodo **x scopre che il percorso migliore per z ora passa per y** con il costo di 3 ($D_x(z) = c(x, y) + D_y(z) = 1 + 2$), e inoltra la notizia al nodo u.
4. Il nodo **u riceve le notizie**. Ora c'è un percorso verso z che costa meno di infinito che passa per x ($D_u(z) = c(u, x) + D_x(z) = 1 + 3$).

Lo **stesso procedimento** avviene per tutti i nodi e percorsi.

Per quanto riguarda la complessità computazionale, tutte le volte che il costo di un nodo viene aggiornato, tutti i nodi devono rivalutare i loro archi. Questo processo conduce ad una **complessità nel caso peggiore di** $O(|N||E|)$.

- Vantaggi:
  - L'**algoritmo è asincrono**, ciò facilita la computazione dato che i router possono adattarsi ai costi aggiornati in qualsiasi stato si trovino.
- Svantaggi
  - **Convergenza lenta**: gli aggiornamenti si diffondono lentamente in quanto tutti i nodi devono rilevare il cambiamento e inviare un aggiornamento agli adiacenti.
  - **Count-to-infinity**: mentre i percorsi migliori vengono immediatamente riconosciuti e adottati dalla rete, i fallimenti sono rilevati lentamente e potrebbero produrre cicli.

Il problema del Count-to-infinity : questo problema è causato dalla natura "locale" dell'algoritmo DV (che è anche la sua forza). Il vettore locale ($D_x(y)$) è direttamente calcolato **partendo dai vettori degli altri nodi** ($D_v(y)$).
Non c'è un **indicazione su quali nodi si trovano nei percorsi migliori** scelti dai nodi adiacenti. Consideriamo il seguente esempio in cui **il link x-y subisce un incremento del costo e solo y lo nota**.



Questa è la situazione iniziale prima dell'incremento (c(x,y) è ancora 4).

- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

Il nodo y a questo punto rileva l'incremento e aggiorna il distance-vector verso x: $D_y(x) = min\{60, 1 + D_z(x)\} = 1 + D_z(x) = 6$.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

Ma il vettore $D_y(x)$ è usato anche da z per calcolare il percorso per x, quindi anche $D_z(x)$ deve essere aggiornato.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

Ma il vettore $D_z(x)$ è esso stesso usato da y per calcolare il percorso per x, quindi anche $D_y(x)$ deve essere aggiornato di nuovo:

- $D_y(x) = 1 + D_z(x) = 8$,
- $D_y(z) = 1$,

- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

Qui nasce il loop: i 2 vettori dovranno essere continuamente aggiornati finché non viene raggiunta una configurazione stabile.
Questo problema poteva essere evitato se anche x avesse comunicato il cambio del costo del link x-y. Se x è malfunzionante o in ritardo per qualche ragione, ci vorrà tempo prima che si raggiunga la convergenza.

4.5.5 Algoritmo Link-State

L'algoritmo **Link-State** (LS) è un approccio **centralizzato** che sfrutta la piena conoscenza sulla rete per trovare il percorso migliore.Questo tipo di algoritmi **non soffrono del problema del count-to-infinity**. L'algoritmo LS è basato sull'**algoritmo di Dijkstra**.

Si noti che tale livello di conoscenza può essere raggiunto se **ciascun nodo invia in broadcast pacchetti link-state**, contenenti ID e costo dei link a cui è collegato, a tutti gli altri nodi nella rete.
La versione base dell'algoritmo calcola il **percorso più breve da un nodo di partenza verso tutti i possibili nodi di destinazione** (quindi deve essere eseguito **su tutti i nodi)**.

In LS prima della computazione del percorso più breve, ogni router deve scoprire i nodi che gli sono vicini e condividere quest'informazione con gli altri. Questa **fase di scambio di messaggi** è fatta in 4 passi:

1. **Scoperta degli adiacenti**: i router mandano uno specifico messaggio di saluto su tutti i link, gli altri router rispondo inviando i loro ID.
2. **Setup dei costi**: i router settano la distanza o il costo per ognuno dei propri adiacenti, che possono dipendere dai costi attuali o impostati dall'amministratore.
3. **Costruzione del pacchetto**: il router crea un pacchetto che riassume gli ID dei nodi adiacenti e i loro costi
4. **Scambio dei pacchetti**: il router invia questo pacchetto in broadcast e riceve pacchetti simili da tutti gli altri router.

Dopo che questo processo è eseguito, **ogni router ha la "completa" conoscenza sulla tipologia della rete**, ora l'algoritmo shortest-path (Dijkstra) può essere eseguito.

Di seguito lo pseudocodice dell'algoritmo Link-State:

```
LinkState(x):
 N' = {x}
 for all nodes v:
  if v is a neighbor of x then
    D(v) = c(x,v)
  else
    D(v) = ∞

 repeat
  find w not in N' such that D(w) is a minimum
  add w to N'
  for each neighbor v of w which is not in N':
     D(v) = min( D(v), D(w) + c(w,v) )
 until N'= N
```

Usiamo la **struttura dati D(v)** per salvare le distanze dal nodo corrente a tutti i possibili nodi di destinazione. Durante la **fase di inizializzazione** assumiamo che **tutti i costi siano noti** e solo le **distanze verso gli adiacenti vengono settate**.

Durante la **fase di costruzione:**

- Selezioniamo il nodo w **più vicino**.
- **Esploriamo i nodi adiacenti** di w, verificando se tutti i percorsi che attraversano w siano migliori di quello attuale.
- Si esce quando **tutti i nodi sono stati esplorati**.

Torniamo a considerare il precedente esempio con 6 nodi (vedi 4.5.4). Ecco come **funziona l'algoritmo LS quando invocato sul nodo u**:

- In questo caso, usiamo un ulteriore funzione **p(x) per salvare il nodo precedente rispetto a quello selezionato**.
- Se vogliamo ricavare il percorso, possiamo **semplicemente tornare indietro da tutti i predecessori** finché il nodo corrente non è raggiunto

step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Per quanto riguarda la complessità computazionale, **ad ogni iterazione verifichiamo tutti i nodi** della rete, eccetto il nodo sorgente, dopodiché una nuova sorgente viene selezionata. Ciò vuol dire che **stiamo rimuovendo un nodo ad ogni iterazione**, quindi il numero totale di iterazioni è $|N|(|N|+1)/2$. Quindi il **caso peggiore è** $O(|N|^2)$.
Questa è una versione base dell'algoritmo, la complessità può essere ridotta introducendo strutture dati come gli heap, **raggiungendo performance migliori:** $O(|N| + |E|log(|E|))$.

- Vantaggi:
  - **Convergenza veloce**: tutte le comunicazioni sono eseguite simultaneamente.
  - **Non c'è count-to-infinity**: lo stato di tutti i link viene propagato.
- Svantaggi:
  - **Algoritmo sincrono**: i nodi devono ricevere informazioni da tutta la rete prima di iniziare.

4.5.6 DV vs. LS

Gli **algoritmi DV e LS adottano approcci complementari** per il routing:

- Nel DV viene sfruttata la **conoscenza locale** sugli adiacenti.
- L'algoritmo LS richiede **informazioni globali** su tutti i nodi.

Complessità del messaggio: **LS è più complesso da implementare** data la comunicazione sincrona necessaria tra tutti i nodi. In DV, la comunicazione è necessaria solo se il percorso migliore cambia.

Velocità della convergenza: **DV richiede più tempo per convergere**, nel frattempo possiamo avere dei percorsi subottimali. L'algoritmo DV soffre anche del problema count-to-infinity.

Robustezza: **LS è considerato come più robusto** in quanto le **tabelle di forwarding vengono calcolate separatamente**. Ogni nodo riceve informazioni da tutti gli altri nodi e crea la propria tabella. In DV tutte le computazioni sono collegate, ogni nodo dipende dalle tabelle degli altri. Se una tabella è sbagliata, tutte le altre tabelle prendono quell'errore. Nota storica: Nel 1997 il malfunzionamento di un router di una piccola ISP causò una reazione a catena che inondò i router della backbone e causò problemi ad una parte di Internet per diverse ore.

In fine, non c'è un vincitore, **entrambe le soluzioni sono utilizzate in Internet**.

4.5.7 Comando Traceroute

In linux possiamo usare il comando **traceroute** per tracciare tutti i percorsi dei nostri pacchetti verso una specifica destinzazione. Il comando traceroute restituirà gli **IP di tutti i dispositivi incontrati** dall'host sorgente all'host di destinazione.

Se **proviamo diverse volte** verso un host lontano (ad esempio google.com), **potremmo vedere percorsi diversi** (a causa dell'aggiornamento del routing).

5 Livello Link e Fisico

5.1 Dal livello di Rete al livello Link e Fisico

Abbiamo visto che il livello di rete garantisce principalmente la comunicazione tra due host (dovunque essi siano).

I livelli link e fisico forniscono la comunicazione tra **2 host connessi**:

- Il **livello link** è la porzione della pila dedicato alla **trasmissione dei pacchetti sui link** (canali di trasmissione), da nodo a nodo sulla rete.
- Il **livello fisico** è la porzione della pila che regola la **struttura dei link** (mezzo di trasmissione, connettori, tipi di cavi, ecc.) e come i bit sono rappresentati/trasmessi lungo il link.

I **livelli Link e Fisico sono strettamente intrecciati** (perciò sono spesso raggruppati in un unico livello chiamato network access). Alcuni protocolli comuni (come Ethernet) copre entrambi i layer.
A differenza dei livelli precedenti, questi **due livelli sono presenti in tutti i componenti** dell'infrastruttura di rete:

- **Nodi**: host (PC, server, ecc.), router, switch, hub, WiFi access point, ecc.
- **Link**: cablati o wireless.

Per trasmettere un datagramma IP, dobbiamo trasferirlo dall'host sorgente al destinatario saltando su ognuno dei link/dispositivi lungo il percorso. Per essere trasferito su un link, **i datagrammi sono incapsulati in un frame del livello link** che varia in base allo specifico tipo di link.
Questo è l'**ultimo incapsulamento**, i frame vengono convertiti in segnali e trasferiti sul link seguendo le proprie specifiche. I segnali possono essere impulsi elettrici, luce o onde radio.

5.2 Servizi

Il **livello link può offrire fino a 4 servizi**:

1. **Framing**: per incapsulare i **datagrammi nei frame del livello link** avendo come payload i datagrammi del livello superiore e degli **specifici header/tailer**. La struttura del frame dipende dai protocolli del livello link/livello fisico
2. **Link access**: per definire la **regolamentazione dell'accesso al link** per mezzo di un protocollo **Medium Access Control** (MAC). Tali regole dipendono dall'architettura del link:
   - Per i **link point-to-point** (singolo mittente e singolo ricevente), il **protocollo MAC è semplice** (o non esistente). Il mittente può inviare il frame ogni volta che il link è inattivo.
   - Per i **link di broadcast** c'è il cosiddetto problema dell'accesso multiplo. Qui, **il protocollo MAC serve a coordinare la trasmissione dei frami** dei vari nodi.
3. **Reliable delivery**: garantisce che i frame trasmessi lungo il link sia ricevuto.
   - Un servizio di consegna affidabile al livello link è **spesso usato per link che tendono ad alti rate di errore**, come i link wireless, con l'obiettivo di correggere un errore localmente. Esso può **provocare un forte overhead**. Dato che altri protocolli al livello applicazione/trasporto sono affidabili, questp servizio **non è sempre implementato**.
4. **Error detection and correction**: l'hardware del livello link può essere afflitto dal problema del **bit flipping**. Alcuni protocolli del livello link implementano strategie di checking/correzione più sofisticati in aggiunta a quelli dei livelli superiore. Dato che non c'è un ulteriore incapsulamento, questi check coprono tutto il messaggio. I check integrati nell'hardware sono inoltre **molto più veloci**.

5.3 Implementazione e tecnologie fisiche

Nei dispositivi di rete le funzionalita del livello link sono **implementate sia sul lato software che (principalmente) sul lato hardware)**.
Nei computer ci sono adattatori di rete chiamati **Network Interface Card** (NIC) che hanno un chip specializzato (il **controller**) per implementare le funzionalità del livello di link. In passato le NIC erano **separate dalla scheda madre** (inserite negli slot PCI), ma recentemente questo approccio sta cambiando in favore di NIC già integrate nella scheda madre.

Il **protocollo del livello link**, e quindi la struttura del frame risultante, **dipende dalla tecnologia** del link fisico (livello fisico). Ecco alcune tecnologie:

- **Ethernet 802.3**: connessione cablata basata su impulsi elettrici su **4 paia di cavi di rame attorcigliati** (i più comuni) oppure su fotoni lungo un cavo in **fibra ottica**.
- **WiFi** 802.11: connessione wireless basata sulle **frequenze radio** (2.4GHz, 5GHz, 6GHz).
- **Bluetooth**: connessione wireless basata su **frequenze radio** (2.4GHz).

Le tecnologie fisiche sono in continua evoluzione, i loro limiti in termini di distanza/bandwidth vengono spesso aggiornati.

5.4 Error Detection and Correction

5.4.1 Formulazione del problema

Nel livello di link è possibile eseguire il **rilevamento e la correzzione degli errori sui bit**. Ipotizziamo di avere dei **dati D di dimensione** d, per proteggerlo da errori di bit includiamo nel messaggio alcuni bit per l'**error detection and correction: *EDC***. L'obiettivo è di **capire se il *D'* e *EDC'* ricevuti differiscono dagli originali *D* e *EDC*** e, se così fosse, di **riottenere gli originali D e EDC**.

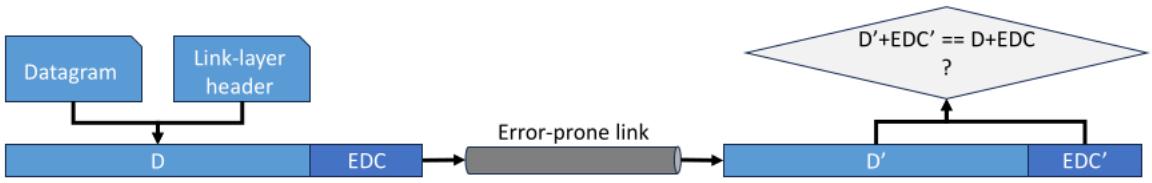


Figure 49: Error detection and correction

Si noti che anche con l'uso dei bit di error detection **possono ancora esserci bit errati non rilevati**, ed il ricevente potrebbe essere ignaro che le informazioni ricevute contengono bit errati.

5.4.2 Parity check

Il **parity check** è la forma più semplice di error detection: **includiamo solo 1 bit addizionale (parity bit)** al messaggio in modo tale che il numero totale di 1 tra i d+1 bit del messaggio sia pari (even parity scheme) o dispari (odd parity scheme).

Messaggio	Parity Bit (even scheme)	Parity bit (odd scheme)
10101100	0	1
11010000	1	0

Table 1: Esempio di parity check

Il **ricevenete deve solo contare il numero di 1** nei d+1 bit ricevuti e confrontarli con lo schema adottato. Chiaramente **questo approccio funziona solo se ci sono un numero dispari di bit errati**, ma quanto è probabile?

Se la **probabilità di bit errati è piccola** e si può supporre che gli errori si verifichino **indipendentemente da un bit al suo successivo**, la probabilità di più bit errati nel pacchetto dovrebbe essere estremamente piccola, ma non **nel caso delle reti**. In pratica, è stato osservato che **gli errori sono spesso raggruppati** insieme in "esplosioni". Sotto queste condizioni, la probabilità di errori non rilevati usando singoli bit di parità può raggiungere il 50%, quindi non è molto efficace come approccio.

5.4.3 Parity check (bidimensionale)

Un possibile modo per migliorare il parity check è usare **più parity bit**. L'approccio bidimensionale è una tecnica in cui il messaggio è diviso in n righe ed m colonne, aventi un bit di parità ciascuno. Qui, il ricevente può rivelare l'occorrenza di un errore, e anche quale bit risulta erraro, dopodiché può **tentare una correzione**. Il parity check bidimensionale può anche rilevare (ma non correggere) qualsiasi combinazione di due errori in un pacchetto.

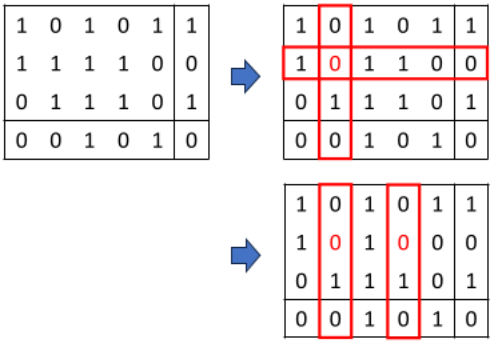


Figure 50: Parity Check bidimensionale

5.4.4 Cyclic Redundancy Check (CRC)

Il **Cyclic Redundancy Check** è una tecnica di error detection largamente usata. Ecco come funziona:

- Per inviare d bit di dati D il **il mittente e il destinatario concordano su un pattern di r+1 bit chiamato generatore** (G), che ha il bit più significativo (quello più a sinistra) a 1.
- Per un dato pezzo di dati D, il **mittente sceglierà r bit addizionali** (bit CRC) che devono essere accodati al messaggio **in modo tale da rendere D+CRC divisibile in modulo 2 da G**.
- Il **destinatario divide il messaggio D+CRC per G**, se il resto è zero, il messaggio è corretto, altrimenti c'è stato un errore.
- Quest'operazione è implementata **shiftando G verso il bit più significativo del messaggio ed eseguendo una XOR bit a bit**.

Consideriamo un esempio semplice di un messaggio D di 6 bit e 3 bit CRC dove:

- d= 6
- D= 1 0 1 1 1 0
- r= 3
- G= 1 0 0 1

Sul lato del mittente **dobbiamo creare il CRC** da D e G, questo CRC verrà aggiunto al messaggio e trasmesso al ricevente. Sul lato del ricevente **useremo D, G e il CRC ricevuto** per capire se il messaggio ricevuto è intatto.

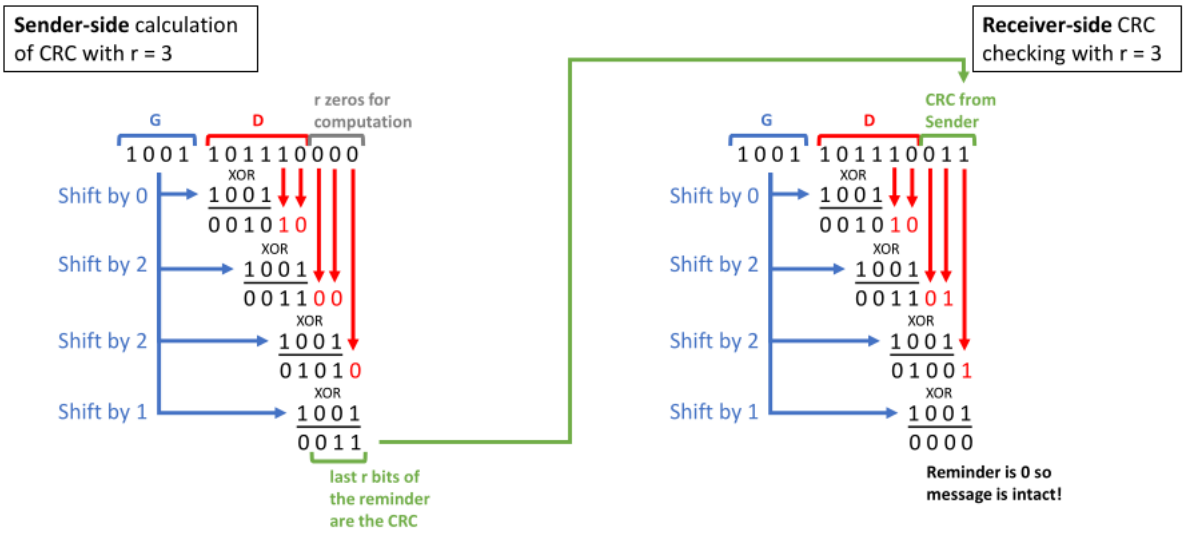


Figure 51: Esempio di Cyclic Redundancy Check

Sono stati definiti **standard internazionali** per generatori a 8, 12, 16 e 32 bit. Il **CRC-32**, che è stato adottato in diversi protocolli IEEE del livello link, usa il seguente generatore (33 bit):

$$G_{CRC-32} = 100000100110000010001110110110111$$

Ognuno di questi **standard CRC può rivelare sicuramente errori di burst inferiori a r+1 bit**, mentre per **errori maggiori di r+1 bit c'è una probabilità P** di trovare l'errore che è:

$$P = 1 - 0.5^r$$

Di conseguenza, la **probabilità di trovare l'errore cresce all'aumentare di r**.

5.5 Link Access

5.5.1 Tipi di link

Di base ci sono 2 tipi di link:

- **Link point-to-point**: costituiti da un **singolo mittente** ad un capo ed un **singolo destinatario** sull'altro capo. Il **point-to-point protocol** (PPP) è un esempio di protocollo che gestisce questi link.
- **Link Broadcast**: molteplici mittenti e destinatari connessi sullo **stesso canale condiviso**. Il termine broadcast è usato perché quando un nodo trasmette un frame esso viene ricevuto da tutti i nodi sul canale. L'accesso a questo link **deve essere coordinato** (problema dell'accesso multiplo) dato che più comunicazioni su un singolo link possono interferire tra di loro.

5.5.2 Collisioni

Il problema principale di un link broadcast è la **collisione**: se più nodi stanno simultaneamente trasmettendo frame sullo stesso canale, **tutti questi frame potrebbero sovrapporsi diventando incomprensibili**. Questi frame collisi sono poi ricevuti da tutti i nodi sul canale e **scartati come errori**. Non è stato causato nessun danno ma:

- Tutti i **frame trasmessi sono stati persi**.
- Il **time-interval viene sprecato** dato che il canale è stato usato per trasmettere dati inutili.

L'**accesso** al mezzo fisico (così come il **framing**) viene regolato dal protocollo **MAC** (Medium Access Control).

5.5.3 Protocollo di accesso multiplo

In una rete **protocolli di accesso multiplo** vengono utilizzati per **regolare le trasmissioni sui canali di broadcast**, in questo modo:

- **Vengono gestite le collisioni**.
- Ciascun nodo ha la possibilità di trasmettere, in modo tale che **i nodi non monopolizzino il link**.
- Le **connessioni stabilite non vengono interrotte**.

Tali protocolli sono necessarie per diverse tipologie di reti (sia cablate, wireless o satellitari), dove **centinaia o migliaia di nodi possono comunicare direttamente su un canale di broadcast**.

Il ruolo primario di un **protocollo di accesso multiplo** è in qualche modo di **evitare le collisioni**. I principali approcci sono:

- **Partizionamento del canale**: la bandwidth viene partizionata per diversi nodi.
- **Accesso casuale**: i nodi "scommettono" per l'accesso.
- **Taking-turns**: i nodi aspettano il proprio turno.

Ulteriori caratteristiche che un protocollo di accesso multiplo per **un canale di broadcast col rate di R bps** dovrebbe fornire sono:

- **Massimizzare l'utilizzo del canale**: se M nodi hanno dei dati da inviare, ciascuno di essi dovrebbe avere, in media, un throughput di R/M bps (se M=1 allora il throughput dovrebbe essere R).
- Essere **decentralizzato**: un nodo master può essere un single point of failure.
- Essere **semplice e leggero**: tonnellate di frame vengono spediti, non deve esserci un overhead.

5.5.4 Protocolli di partizionamento del canale: TDMA

Il **Time division multiple access**: ipotizziamo di avere un canale con N nodi che ha un rate di trasmissione di R bps, il **TDMA divide il tempo in frame di tempo** (steps) e divide ulteriormente **ogni frame in N slot di tempo**.
Ciascuno **slot di tempo è assecciato ad uno** degli N nodi. Ogni volta che un nodo ha un pacchetto da inviare, esso aspetta l'assegnazione di un time slot. Tipicamente, **la dimensione degli slot è decisa in modo tale che un intero pacchetto possa essere trasmesso** durante uno slot di tempo.

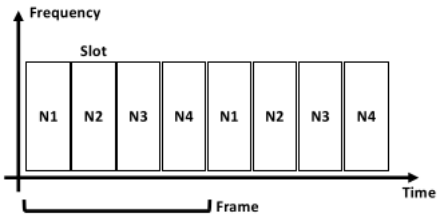


Figure 52: Esempio di TDMA

5.5.5 Protocolli di partizionamento del canale: FDMA

Il **frequency division multiple access** divide gli R bps del canale in frequenze differenti (ciascuna con una bandwith di R/N) e assegna ciascuna frequenza ad uno degli N nodi.
FDMA e TDMA condividono pro e contro:

- **Evitano le collisioni e dividono la bandwidth equamente** tra gli N nodi.
- Un **nodo è limitato da una bandwidth di R/N**, anche quando è l'unico nodo con pacchetti da inviare.

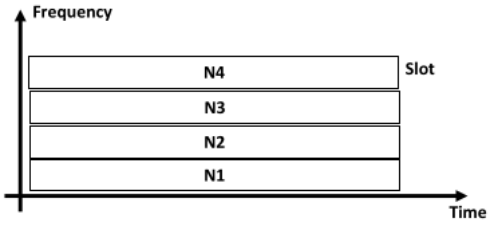


Figure 53: Esempio di FDMA

5.5.6 Protocolli di partizionamento del canale: CDMA

Il **code division multiple access** assegna un diverso codice a ciascun nodo che viene usato per codificare/decodificare i bit di dati che invia.
Se i codici sono scelti con attenzione, le **reti CDMA consentono a diversi nodi di trasmettere simultaneamente** senza causare interferenze. Il CDMA è usato **principalmente su canali wireless**. È stato usato nei sistemi militari per diverso tempo (grazie alle sue proprietà anti-jamming) e **anche nella telefonia cellulare** (come il 3G). È complesso da implementare e non scala bene col numero dei nodi.

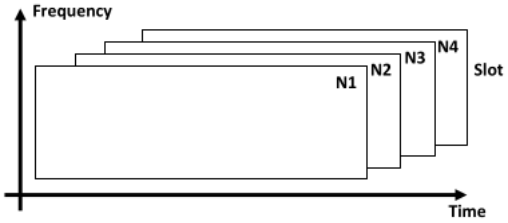


Figure 54: Esempio di CMDA

5.5.7 Protocolli ad accesso casuale

Nei protocolli ad accesso casuale, un **nodo che sta trasmettendo trasmette sempre al massimo rate** del canale (a R bps).
Se **si verifica una collisione** (cioè almeno 2 nodi stanno trasmettendo) tutti i nodi che stanno trasmettendo **aspettano un ritardo casuale** prima di tentare di nuovo la ritrasmissione.
Dato che questa **selezione è eseguita indipendentemente**, i 2 nodi possono **scegliere un ritardo che è abbastanza differente** da consentire ad uno dei due contendenti di inserire il messaggio.
Se, altrimenti, **vengono scelti ritardi simili**, si verifica una nuova collisione, ed il **processo viene iterato**.

5.5.8 Protocolli ad accesso casuale: Slotted ALOHA

Il protocollo **slotted ALOHA** funziona in questo modo:

- Il **tempo è diviso in slot** dove ciascuno slot è largo abbastanza da contenere un frame.
- I nodi sono sincronizzati, quindi **ciascun nodo trasmette i frame solo all'inizio di uno slot**.
- **Se non viene rilevata nessuna collisione** nello slot, la **comunicazione continua**.
- Altrimenti, **se viene rilevata una collisione** in uno slot, **ciascun nodo in collisione ha una probabilità** $p \in [0, 1]$ **di ritrasmettere questo frame** in ciascuno degli slot seguenti, finché il frame non viene inviato con successo.

Caratteristiche:

- **Se c'è un solo nodo, userà l'intero rate R** del canale (nessuna collisione).
- È **decentralizzato** dato che i nodi sono totalmente indipendenti dagli altri.
- È **semplice da implementare** e da eseguire.
- È **improbabile che si verifichino collisioni multiple consecutive**.

5.5.9 Protocolli ad accesso casuale: CSMA

Una **debolezza di ALOHA è che potremmo iniziare una trasmissione** (producendo una collisione) **anche se il canale è già impegnato**. Una soluzione è **monitorare il canale** e di tentare la trasmissione solo se il canale è a riposo.

I protocolli **Carrier Sense Multiple Access** (CSMA) e **CSMA with collision detection** (CSMA/CD) sono basati su 2 principi:

- **Carrier sensing**: i nodi **ascoltano il canale prima della trasmissione**. Se un frame da un altro nodo viene trasmesso in quel momento, il nodo aspetta finché nessuna trasmissione viene rilevata.
- **Collision detection**: i nodi **ascoltano il canale mentre sta trasmettendo**. Se viene rilevata una collisione, interrompe la trasmissione e aspetta un tempo casuale prima di ricominciare.

Se tutti i nodi performano il carrier sensing, **perché le collisioni si verificano collisioni in primo luogo?** A causa del **ritardo nella trasmissione del segnale**.
Anche se la propagazione dei segnali nel canale è tipicamente vicino alla velocità della luce, **impiega tempo per raggiungere tutti gli altri nodi**. Quindi, un secondo nodo rileva la trasmissione solo dopo che è iniziata.
A causa di questo ritardo tra l'inizio della trasmissione e la sua rilevazione, un **nodo può considerare come libero un canale attualmente in uso**, producendo una collisione.

Il **CSMA puro** è semplice:

1. **Verifica se il canale è impegnato**.
2. **Se il canale è a riposo, invia un frame**.

Le collisioni **non vengono rilevate** ma sono comunque possibili, capiamo che un frame è stato perso solo **perché non viene ricevuto un ACK**.

Il **CSMA/CD è più evoluto** (correntemente **implementato dal protocollo Ethernet**):

1. **Controlla se il canale è impegnato**.
2. **Se il canale è a riposo, invia un frame**.
3. **Mentre trasmette, verifica la presenza di possibili collisioni**.
4. **Quando una collisione viene rilevata**, ferma la trasmissione e **aspetta per un periodo casuale** $K \in \{0, \dots, 2^n - 1\}$ dove **n è il numero di collisioni rilevate** sul frame corrente (**binary exponential backoff**).

Dato che i periodi possibili aumentano, **la probabilità di inviare con successo un frame aumentano col numero di collisioni**.

5.5.10 Protocolli Taking-turns: Polling

Nel protocollo **Polling** c'è un **nodo master che seleziona in modo round-robin un nodo alla volta** a cui è consentito trasmettere (fino al massimo throughput). Questo processo è iterato ogni volta che la trasmissione si interrompe (come nel Bluetooth).

- Non ci sono collisioni.
- C'è un **polling delay** (per selezionare il nodo).
- L'approccio è **centralizzato**, c'è un single point of failure.

5.5.11 Protocolli Taking-turns: Token-passing

Nel protocollo **Token-passing** non c'è un **nodo master, i nodi si scambiano un frame speciale chiamato token**, se un nodo riceve il token gli viene consentita la trasmissione, quindi il token viene passato al nodo successivo.

- Non ci sono collisioni.
- L'approccio è **decentralizzato**.
- Ci sono **problemi se qualche nodo dimentica di rilasciare il token** (in questo modo finisce per monopolizzare il link).

5.6 Indirizzi MAC

Al livello link, i **dispositivi sono identificati dagli indirizzi MAC**:

- **Ciascuna interfaccia di rete ha uno specifico indirizzo MAC** (era progettato per essere fisso ma può essere cambiato).

L'indirizzo MAC (o indirizzo fisico) è **un indirizzo del livello link composto da 6 byte** (2^{48} possibili indirizzi) spesso rappresentati in notazione esadecimale:

1A:23:F9:CD:06:9B or 1A-23-F9-CD-06-9B

Gli indirzzi MAC sono locali (mentre gli IP sono globali). Tutte le interfacce sono associate ad un indirizzo MAC, ma è usato solo all'interno della LAN.
In una LAN, 2 interfacce (A e B) comunicano in questo modo:

- **A include l'indirizzo MAC di B nel frame** e lo trasmette.
- **B riceve il frame e compara il proprio MAC con il MAC di destinazione** del frame.
- **Se i 2 MAC sono uguali, il frame viene accettato**, altrimenti il frame viene scartato (il resto della pila non viene coinvolto).

C'è anche la possibilità di **inviare messaggi in broadcast** (che vengono accettati a prescindere dal MAC), per le LAN che usano indirizzi a 6 byte, **l'indirizzo di broadcast è una stringa di 48 bit a 1 consecutivi**. In una LAN è possibile che un interfaccia riceva frame diretti ad un'altra interfaccia. **Il ruolo dell'interfaccia MAC è di filtrare i frame indesiderati** senza disturbare l'host.

5.7 Switch

Gli **switch sono l'equivalente dei router nel livello link**:

- Non viene implementato un algoritmo di **routing**.
- **Solo gli indirizzi MAC vengono usati**, gli IP non vengono considerati.

Il ruolo dello switch è di **ricevere frame del livello link e inoltrarli** sul link in uscita:

- Lo **switch è trasparente** agli host e router nella sottorete.
- Uno switch ha anche dei **buffer sulle interfacce**.

Gli **switch hanno tabelle di forwarding** che associano indirizzi MAC alle interfacce. La **tabella viene aggiornata automaticamente e dinamicamente** (self-learning) ogni volta che un nuovo dispositivo viene scoperto.

5.7.1 Switched LAN

È tipico per le **LAN usare uno o più switch** per connettere molteplici dispositivi locali.
A differenza dei router, **gli switch sono più veloci e plug-and-play**:

- Non c'è un algoritmo di routing coinvolto
- Vengono considerati **solo 2 livelli della pila**.

D'altro canto, le switched LAN sono **di dimensioni limitate** e devono essere **strutturate ad albero**:

- Gli **indirizzi MAC sono difficili da raggruppare** (le tabelle di forwarding negli switch possono crescere rapidamente).
- Dato che non c'è routing, **i loop sono difficilmente evitabili**.

5.8 ARP

Dato che i protocolli del livello superiore lavorano con gli indirizzi IP, **dobbiamo tradurre gli IP in indirizzi MAC**.
L'**Address Resolution Protocol** (ARP) gestisce la conversione tra indirizzi IP e MAC. **Ciascuna interfaccia è dotata di un modulo ARP che ha una tabella ARP** che associa ciascun IP nella LAN ad un indirizzo MAC **con uno specifico valore time to live** (TTL) dopo il quale la entry è cancellata. Dato che gli indirizzi MAC sono locali, anche **ARP lavora solo su reti locali**.

Esempio Assumiamo che l'host C (222.222.222.220) **vuole inviare messaggi ad A** (222.222.222.222). Per far ciò, dobbiamo anche conoscere il MAC associato:

- Prima di inviare il messaggio, se **A non è presente nella tabella** viene inviata un pacchetto "ARP request" in **broadcast** a tutti i dispositivi della rete alla ricerca del giusto IP.
- Tutti i nodi ricevono il pacchetto ma **solo l'indirizzo IP ricercato risponde** con un messaggio diretto "ARP reply".

Un host malevole potrebbe intercettare il messaggio, rispondendo con un "forget reply" (**ARP poisoning**). L'ARP poisoning è uno degli attacchi più comuni sulle LAN.

5.8.1 ARP: Gateway

Cosa accade **se l'IP è fuori dalla rete**? Il router che connette le 2 reti **deve avere almeno 2 interfacce** (2 IP,MAC e tabelle ARP) ciascuna di esse all'interno della specifica sottorete.
I **frame diretti all'esterno della sottorete sono inviati alla prima interfaccia** del router, spostati verso la seconda interfaccia, **e inviata al giusto host usando la seconda tabella ARP**.

5.9 Frame Ethernet

Il frame del protocollo **Ethernet** è composto dai seguenti campi:

- **Campo dati** (da 46 a 1500 byte): contiene il datagramma IP. Il limite massimo è dato dal maximum transmission unit (MTU) di Ethernet, se il datagramma eccede viene frammentato.
- **Indirizzo di destinazione** (6 byte): contiene l'**indirizzo MAC del destinatario**.
- **Indirizzo sorgente** (6 byte): contiene l'**indirizzo MAC della sorgente** che trasmette il frame sulla LAN.
- **Type field** (2 byte): specifica il **protocollo del livello di rete** usato per questo frame (ci potrebbe essere un alternativa ad IP, ad esempio, i pacchetti ARP hanno un tipo specifico - 0x0806).
- **CRC** (4 byte): contiene il **numero di CRC**.
- **Preamble** (8 byte): è un **blocco di "wake-up" usato per sincronizzare i clock** degli adattatori di destinazione e sorgente.

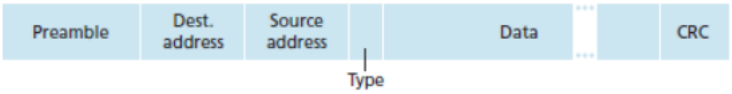


Figure 55: Frame del protocollo Ethernet

6 Panoramica generale della Pila

Come abbiamo visto, **diversi protocolli nella pila TCP/IP collaborano** per consentire ai messaggi di viaggiare in una rete.
Ciascun protocollo di un livello specifico aggiunge un pezzo di informazioni al messaggio regolando uno o più aspetti della comunicazione.
A volte è difficile avere una visuale d'insieme studiando singolarmente i singoli protocolli, faremo quindi un recap dell'intero processo proponendo uno **scenario di richiesta di una pagina web**.

6.1 Esempio di una pagina web

Ipotizziamo che un **utente (Bob) vuole accedere alla pagina web di google.com dalla rete istituzionale**. Per far ciò, Bob **connette il proprio laptop alla rete scolastica mediante un cavo Ethernet**.
In questo esempio ipotizziamo di avere una **tipica configurazione** per la rete scolastica:

- Le **prese Ethernet (a muro) sono collegate ad uno switch**, il quale è **connesso al router DHCP della scuola.**
- Il **router della scuola è connesso ad un ISP (come Comcast)** che fornisce il **servizio DNS**.

Per semplicità diamo per assunto alcune cose:

- Non c'è un **servizio NAT** dal router (IP pubblico).
- tutte le connessioni sono di tipo Ethernet.
- i pacchetti non vengono mai persi
- Il protocollo usato è http e non https.

All'inizio, l'**utente conosce poco sulla topologia della rete**, il modo in cui la LAN è connessa ad Internet, o i dispositivi coinvolti.
Ciò che è a conoscenza è:

- **C'è un sito google.com da qualche parte nell'internet** che Bob vuole raggiungere.
- **La rete locale è in qualche modo connessa ad Internet**.
- **Il laptop può essere connesso alla rete locale** tramite un cavo Ethernet.
- La **LAN ha un DHCP attivo**.

Una volta che il **laptop è connesso fisicamente** alla rete, deve:

- Accedere alla rete (ottenere informazioni dal DHCP)
- **Ottenere l'indirizzo IP del sito web**. (DNS)
- **Ottenere la pagina web di Google**. (HTTP)

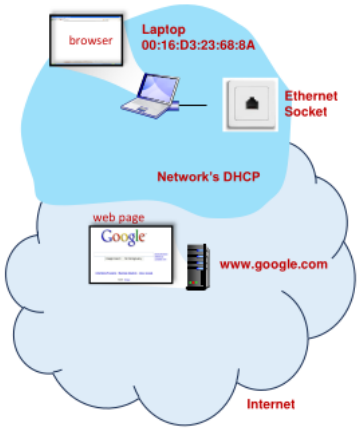


Figure 56: Conoscenza della rete all'inizio

6.1.1 DHCP

Il primo passo per il laptop è di accedere alla rete **richiedendo un host IP, il gateway e il DNS** al DHCP.

1. Il **sistema operativo sul laptop crea un messaggio DHCP discovery** e inserisce questo messaggio **in un segmento UDP** con la porta di destinazione 67 (server DHCP) e la port sorgente 68 (client DHCP).
   Il **segmento UDP viene poi inserito in un datagramma IP** con un IP di destinazione broadcast (255.255.255.255) e l'IP sorgente 0.0.0.0 (ancora nessun host IP)

2. Il **datagramma IP** contenente il messaggii DHCP discovery è poi piazzato **in un frame Ethernet** con l'indirizzo MAC di broadcast (FF:FF:FF:FF:FF:FF) come destinazione (raggiungerà tutti i dispositivi sullo switch e, si spera, il DHCP) e l'indirizzo MAC del laptop come sorgente (00:16:D3:23:68:8A).

3. Il **frame Ethernet che contiene il messaggio DHCP viene inviato allo switch**. Lo switch **invia in broadcast il frame** su tutte le porte in uscita (incluse le porte connesse al router).

4. Il **router riceve il frame Ethernet** che contiene il DHCP discovery sulla sua intergaccia (con indirizzo MAC 00:22:6B:45:1F:1B), **il messaggio viene decapsulato**:

   - Il **frame è accettato poiché ha l'indirizzo MAC di broadcast come destinazione**, quindi viene estratto il datagramma IP.
   - Il **datagramma è accettato poiché ha l'indirizzo IP di broadcast come destinazione**, quindi viene estratto il segmento UDP.
   - **Viene eseguito un demultiplexing sul segmento UDP** e il payload viene ricevuto dal processo DHCP sul router.

5. Assumiamo ora che il **router DHCP può allocare indirizzi IP nel bocco CIDR 68.85.2.0/24** (che è una sottorete fornita dall'ISP):

   - Il **server DHCP offre l'indirizzo IP** 68.85.2.101 al laptop.
   - Il **server DHCP crea un messaggio DHCP offer** che contiene:
     - L'**indirizzo IP offerto** e la mask (68.85.2.101/24)
     - L'**indirizoo IP del server DNS** (68.87.71.226).
     - L'**indirizzo IP per il gateway** (68.85.2.1).
   - Il messaggio viene **incapsulato**:
     - Viene creato un **segmento UDP** con la porta sorgente 67 e la porta di destinazione 68.
     - Il **segmento viene messo in un datagramma IP con l'indirizzo broadcast come destinazione** e 68.80.2.1 come indirizzo sorgente.

– Il **datagramma viene messo in un frame Ethernet con indirizzo MAC sorgente 00:22:6B:45:1F:1B** (quello dell'interfaccia LAN del router) e **indirizzo MAC di destinazione 00:16:D3:23:68:8A** (quello del laptop).

6. Il frame Ethernet **contenente l'offerta DHCP è inviato** (unicast) dal router allo switch, il quale lo inoltra al laptop verificando l'indirizzo MAC di destinazione.

7. Il **laptop di Bob riceve il frame Ethernet** con l'offerta DHCP e lo **decapsula**:

   - **Il frame viene accettato dato che l'indirizzo MAC è corretto**, viene quindi estratto il datagramma IP.
   - **Il datagramma IP viene accettato dato che è stato usato l'IP di broadcast**, quindi viene estratto il segmento UDP.
   - L'**offerta DHCP subisce un demultiplexing** dal processo DHCP del sistema operativo del client.
   - Il **sistema operativo accetta l'offerta e risponde con un messaggio DHCP request** (saltato per brevità).
   - Quando un **messaggio DHCP ACK viene infine ricevuto** da un nuovo segmento UDP, il **sistema operativo imposta le informazioni di rete ricevute** (IP, gateway e DNS). Ora il laptop è entrato nella rete

All'inizio, conoscevamo solo il MAC del laptop, **ora conosciamo**:

- L'**indirizzo IP del laptop**
- L'**indirizzo IP del router**(gateway)
- La **configurazione di rete**(mask)
- L'**indirizzo IP del DNS**

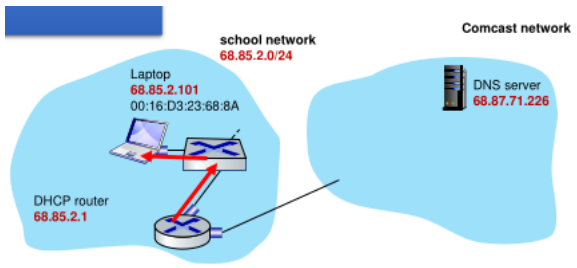


Figure 57: Conoscenza della rete a questo punto

6.1.2 DNS e ARP

Ora l'**utente decide di aprire il browser e navigare verso l'home page di Google** (google.com), quindi l'indirizzo IP del server web deve essere ricavato dal DNS:

8. Il **browser invoca una routine del sistema operativo per ottenere l'indirizzo IP del server** (come nella funzione gethostbyname):

   - Il **sistema operativo crea un messaggio DNS query** per l'hostname "www.google.com".
   - Il **messaggio DNS è encapsulato in un segmento UDP** con 53 come porta di destinazione (server DNS).
   - Il **segmento UDP viene piazzato in un datagramma IP** con 68.87.71.226 (indirizzo IP del DNS ricavato dal DHCP) come indirizzo di destinazione e 68.85.2.101 come indirizzo sorgente (se stesso).

9. Il datagramma IP che contiene la query **DNS deve essere incapsulata in un frame Ethernet**. Per far ciò **ci serve l'indirizzo MAC del gateway**, quindi deve essere creata una **query ARP**. Da notare che anche se **conosciamo l'indirizzo IP del gateway** dal DHCP, **non siamo a conoscenza dell'indirizzo MAC**.

10. Il **sistema operativo crea un frame query ARP** con l'**indirizzo IP target 68.85.2.1** (il default gateway) e broadcast come indirizzo MAC di destinazione.
Il **frame viene inviato allo switch**, il quale lo consegna a **tutti i dispositivi connessi**, incluso il **router gateway**.

11. Il **router riceve il frame** sulla sua interfaccia lato-LAN e scopre che l'indirizzo IP targe 68.85.2.1 nel **messaggio ARP coincide con l'indirizzo IP dell'interfaccia**:

    - Il **router gateway prepara una risposta ARP** indicando che l'indirizzo MAC 00:22:6B:45:1F:1B corrisponde all'indirizzo IP 68.85.2.1
    - La **risposta ARP viene messa in un frame Ethernet** con indirizzo MAC di destinazione 00:16:D3:23:68:8A (indirizzo del laptop).
    - Il **frame è inviato allo switch**, il quale lo consegna al **laptop**.

12. Il **laptop riceve il frame** contenente il messaggio ARP di risposta ed estrae l'indirizzo MAC del router gateway (00:22:6B:45:1F:1B).

13. Il **laptop può ora inviare il frame Ethernet contenente la query DNS** all'indirizzo MAC del gateway. Si noti che, in questo caso, il **datagramma IP** avrà come **indirizzo IP di destinazione 68.87.71.226** (il server DNS), ed un **indirizzo MAC di destinazione 00:22:6B:45:1F:1B** (il router gateway).

14. Il **router gateway riceve il frame Ethernet** e lo decapsula:

    - Il **router estrae il datagramma IP** dal frame e ricava l'indirizzo IP di destinazione del DNS.
    - Il **router cerca l'indirizzo IP di destinazione** (68.87.71.226) e determina dalla sua tabella di forwarding che il **datagramma dovrebbe essere inviato al primo router della rete dell'ISP** (il più a sinistra nella rete di Comcast nell'immagine)
    - Il **datagramma IP viene piazzato in un frame** appropriato per il link che connette il router della rete scolastica al router di Comcast. Il frame viene quindi inviato su questo link.

15. Il **primo router dell'ISP riceve il frame** ed estrae il datagramma IP:

    - Il **router verifica l'indirizzo di destinazione** (68.87.71.226) e ricava dalla tabella di forwarding (creata con un algoritmo LS o DV) l'interfaccia d'uscita.
    - Un **nuovo frame viene creato e inviato al prossimo router** attraverso l'interfaccia selezionata.
    - **Questo processo può essere iterato** più volte prima di raggiungere il server DNS

16. Infine il datagramma IP che contiene la **query DNS arriva al server DNS**:

    - Il server DNS estrae la query DNS, **cerca il nome www.google.com dal suo database** e cerca il suo indirizzo IP (64.233.169.105).
    Si noti che **in questo caso abbiamo assunto un cached IP**, altrimenti dovrebbero essere inviate richieste addizionali a degli authoritative servers.
    - Il **server DNS crea un messaggio DNS reply** che contiene l'indirizzo IP ricavato, e **lo piazza in un segmento UDP**.
    - Il **segmento viene messo in un datagramma IP** con l'indirizzo IP del laptop come indirizzo di destinazione. Viene messo poi in un apposito frame.
    - Questo **messaggio verrà inoltrato** attraverso la rete di Comcast al router della rete scolastica e poi, tramite lo switch, al laptop.

17. Il **sistema operativo del laptop estrae l'indirizzo IP target** dal messaggio DNS. Ora sappiamo **come raggiungere il server web di Google**.

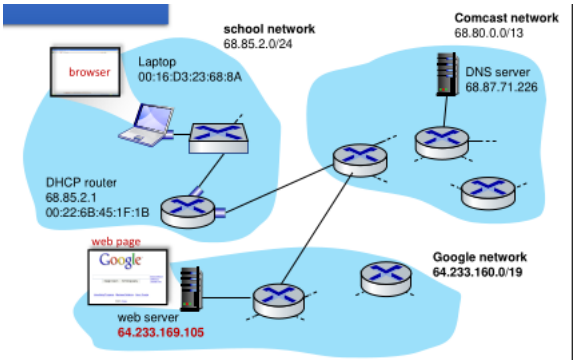


Figure 58: Conoscenza della rete a questo punto

6.1.3 HTTP

Conoscendo l'IP del server possiamo **creare una richiesta HTTP GET** per l'home page di google:

18. Il **browser crea una socket TCP**, quindi il sistema operativo **esegue un three-way handshake** con il web server di Google:
    - Il **sistema operativo crea un segmento TCP SYN** con la prota di destinazione 80
    - Il **segmento è incapsulato in un datagramma IP** con indirizzo di destinazione 64.233.169.105 (www.google.com).
    - Il **datagramma viene messo in un frame** con indirizzo MAC di destinazione di 00:22:6B:45:1F:1B (il router gateway) ed inviato allo switch.
19. Tutti **i router (della rete locale, ISP e di google) inoltrano il datagramma** al server web usando la loro tabella di forwarding.
    Si noti che i **frame possono essere modificati lungo il percorso in base ai link specifici**.
20. Il frame contenente il **segmento SYN arriva infine al server**:
    - Il **datagramma è estratto** dal frame.
    - Il **segmento viene estratto** dal datagramma e **demultiplexato** dalla socket di "welcome" associata alla porta 80.
    - Una **socket specifica per la connessione viene creata** tra il server web di Google e il laptop.
    - Un **segmento TCP SYNACK viene generato** piazzato all'interno di un datagramma avente come indirizzo IP di destinazione l'indirizzo del laptop (68.85.2.101).
    - Il **segmento è piazzato in un frame del protocollo appropriato** per viaggiare lungo il link che connette www.google.com al primo router.
21. La **TCP SYNACK infine arriva al laptop**:
    - Il **datagramma viene demultiplexato** dal sistema operativo sulla socket TCP creata nello step 18.
    - La **socket entra nello stato di "connessione stabilita"**.
22. La **socket sul laptop ora è pronta per inviare byte al server web**:
    - Il **browser crea il messaggio HTTP GET** contenente l'url che deve essere recuperato.
    - Il **messaggio viene scritto nella socket**, e la richiesta GET diventa il **payload del segmento TCP**.
    - Il **segmento TCP è piazzato in un datagramma e quindi nel fram** (incapsulazione).
    - Il **messaggio è consegnato verso www.google.com** (come negli step 18-20)
23. Il **server web di www.google.com riceve il messaggio GET** dalla socket TCP:

- Il **messaggio viene decapsulato e demultiplexato**, quindi la **richiesta GET viene interpretata**.
- Il **server crea una messaggio di risposta HTTP**, con l'**home page di Google nel body**.
- Il **messaggio viene scritto nella socket TCP**.
- Il **sistema operativo del server incapsula il messaggio in un segmento, poi in un datagramma ed infine in un frame**.

24. Il datagramma contenente **il messaggio di risposta HTTP viene inoltrato attraverso le 3 reti**, ed arriva al laptop:
    - Il **sistema operativo del laptop decapsula il messaggio**, il quale **viene demultiplexato dal browser**.
    - Il **browser legge la risposta HTTP** dalla socket.
    - Il **browser estrae il codice html dal body** della risposta HTTP.
    - Il **browser finalmente mostra l'home page di Google**

7 Network Security

7.1 Introduzione

Alcune applicazioni potrebbero scambiare **informazioni sensibili**. Quindi sulle reti pubbliche i messaggi dovrebbero essere **protetti**. Alcuni protocolli (come HTTP, SMTP, FTP) offrono una "loro versione sicura" che implementano tecniche per la protezione delle informazioni.
La **network security** è il campo che studia **possibili attacchi** verso le reti e **possibili modi per prevenirli**.
Ci sono **diversi tipi di attacchi** che hanno diversi scopi e diverse meccaniche. Gli attacchi evolvono insieme alle tecnologie e alla mole di utenti che utilizzano le reti.

7.2 Tipologie di attacchi

7.2.1 I malware

Un **malware** è un **software malevolo** che può essere trasferito ad un computer tramite la rete (scaricando file, attraverso allegati di e-mail, ecc.).
Una volta che il malware infetta il dispositivo, lo può danneggiare in diversi modi:

- Forzando un sistema a mostrare pubblicità (**adware**)
- Mostrando messaggi di allarme falsi per indurre gli utenti a scaricare malware (**scareware**)
- Cancellando (**wiper**) o criptando (**ransomware**) i file del dispositivo.
- Raccogliendo informazioni private come password, credenziali, ecc. (**spyware**). Ad esempio i **keylogger** sono software che registrano (loggano) i tasti battuti su una tastiera.
- Ottenendo privilegi di root del sistema (**rootkits**).
- Trasformando il dispositivo in uno slave o foothold (punto d'appoggio) per attaccare altri dispositivi (**zombie** o **botnet**).

I malware odierni sono spesso **self-replicating**: una volta che infettano un host, da quell'host **cerca di entrare negli altri host su Internet**, e dai nuovi host infettati, cerca di entrare in altri host ancora.
Un malware si può diffondere nella forma di un virus o di un worm:

- I **virus** sono malware che **richiedono una qualche forma di interazione con l'utente** per infettare il suo dispositivo. Ad esempio un allegato di una mail che contiene un codice eseguibile malevolo che si replica inviando mail simili ai contatti dell'utente. Sono tipicamente **mascherati come componenti software legittimi** (cavalli di Troia).
- I **worms** sono malware che **entrano in un dispositivo senza nessuna interazione esplicita dell'utente**. Ad esempio, un utente può eseguire un'**applicazione di rete vulnerabile che accetta il worm senza intervenire**. Il worm poi scansiona la rete cercando host che eseguono un'applicazione simile.

7.2.2 DoS

Gli attacchi DoS (**Denial-of-Service**) sono abbastanza comuni e sono progettati per rendere una rete, un host o altri pezzi dell'infrastruttura inutilizzabile dagli utenti legittimi.
Ci sono 3 tipi di attacchi DoS:

1. **Attacco di vulnerabilità**: inviando messaggi adatti per applicazioni vulnerabili o sistemi operativi per farli fermare o crashare.
2. **Bandwidth flooding**: inviando una grande quantità di pacchetti all'host bersaglio, impedendo ai pacchetti legittimi di raggiungere il server.
3. **Connection flooding**: stabilendo un gran numero di connessioni TCP aperte a metà o per intero con l'host bersaglio, quindi impedisce alle connessioni legittime di essere accettate.

Gli attacchi DoS possono essere **distribuiti** (DDos) cioè eseguiti da più host usando una **botnet** di zombie (o slaves).

7.2.3 Packet Sniffing

Il **packet sniffing** coinvolge un ricevitore passivo (**sniffer**) che registra una copia di pacchetti rilevanti da un host bersaglio cercando di rubare informazioni sensibili (aka **eavesdropping**).
Gli sniffer possono essere distribuiti in tutti i tipi di reti **broadcast** semplicemente copiando i pacchetti che sono indirizzati a destinazioni differenti invece di scartarli automaticamente.
Nelle reti non broadcast invece, uno sniffer può essere piazzato in un malware (spyware) e usato per infettare i dispositivi della rete (come i router) in modo tale che tutto il traffico inoltrato venga anche copiato.

Dato che gli sniffer sono **passivi** (non viene aggiunto traffico all'interno della rete) **essi sono molto difficili da rilevare**.
Per prevenire lo sniffing vengono usati approcci basati sulla **crittografia**.
Ci sono diversi sniffer disponibili gratuitamente su internet. Un esempio di packet sniffer è **Wireshark** (una volta chiamato **Ethereal**).
Wireshark è un **software per l'analisi di pacchetti o protocolli**, principalmente usato per scopi legittimi (creazione di nuovi protocolli, monitoraggio di una rete, ecc.).

7.2.4 IP Spoofing

L'**IP spoofing** è una tecnica che consente ad host malevoli di iniettare nella rete pacchetti con falsi indirizzi sorgente. Può essere usato in **combinazione con applicazioni vulnerabili** per attaccare specifici host essendo mascherati come un altro utente. Può anche essere usato per **attacchi DoS** (un'alternativa alle botnet) come messaggi da differenti IP sorgenti che sono **più difficili da filtrare**.
Lo spoofing (e lo sniffing) possono anche essere usati per attacchi **man in the middle** (MitM o MiM), dove colui che attacca si trova tra 2 host comunicanti camuffandosi come entrambi. I 2 host credono di star comunicando tra di loro, quando **in realtà stanno comunicando con l'attaccante** che si finge a turno come uno dei 2.
Per prevenire lo spoofing, possiamo usare **verifiche di integrità del messaggio** e l'**autenticazione end-point**, consentendoci di determinare se il messsaggio non è stato modificato o se il messaggio è originato dalla sorgente giusta.

7.3 Basi della Network Security

Ora che abbiamo visto i vari tipi di attacchi, possiamo definire un insieme di proprietà che una **comunicazione sicura** dovrebbe garantire:

- **Confidenzialità**: solo il mittente e il destinatario previsto dovrebbero essere in grado di capire il contenuto del messaggio trasmesso (per prevenire lo sniffing)
- **Integrità del messaggio**: il contenuto della comunicazione non deve essere alterato, né da azioni malevoli né da errori.
- **Autenticazione end-point**: sia il mittente che il destinatario dovrebbero essere in grado di confermare l'identità dell'altra parte coinvolta nella comunicazione (per prevenire lo spoofing).
- **Sicurezza operazionale**: fare affidamento ad un'infrastruttura di rete che previene l'intrusione di host malevoli all'interno della comunicazione.

Le **prime 3 proprietà sono software-based**, mentre l'ultima tipicamente si affida ad **hardware specifici** (firewall, sistemi di rilevamento di intrusioni).

7.4 Crittografia

Una **tecnica crittografica** consente ad un mittente di **mascherare i dati** in modo tale che essi diventino incomprensibili per un intruso. L'intruso **non può ottenere informazioni** dal messaggio intercettato, ma il **destinatario deve essere in grado di ripristinare il dato originale** dal dato mascherato.
La versione originale del messaggio è chiamata **plaintext** (o testo in chiaro) ed è leggibile da chiunque.
Prima di immettere il messaggio in un canale, l'host A usa un algoritmo di crittografia per trasformare il messaggio in una forma non leggibile chiamata **ciphertext** (testo cifrato) il quale deve essere decriptato

quando verrà ricevuto.
In molti sistemi crittografici moderni, inclusi quelli usati su internet, la **tecnica di cifratura è nota a chiunque** (anche l'intruso). La **parte non nota** dell'algoritmo sono le **chiavi di cifratura/decifratura**. Una **chiave** è una stringa alfanumerica che deve essere fornita all'algoritmo di crittografia al fine di cifrare/decifrare il messaggio. Le chiavi per la cifratura e decifratura possono essere **identiche** (crittografia simmetrica) o **differenti** (crittografia asimmetrica).

7.4.1 Crittografia simmetrica

Nella **crittografia simmetrica** c'è **una sola chiave** che è usata sia per la cifratura che per la decifratura.
Cifratura: il messaggio in chiaro insieme alla chiave viene passato all'algoritmo di cifratura per generare un testo cifrato che può essere inviato in modo sicuro attraverso la rete.

Decifratura (o decodifica): il messaggio cifrato insieme alla chiave viene passato all'algoritmo di decodifica per ricreare il messaggio in chiaro iniziale.

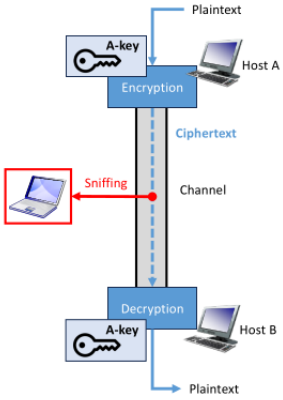


Figure 59: Esempio di crittografia simmetrica

Il **problema** principale con la crittografia simmetrica è la comunicazione della chiave segreta (**problema dello scambio della chiave**). Gli host possono usare **canali sicuri** per scambiare la chiave. Gli host possono usare alcuni **protocolli** che consentono loro di "convergere" su una chiave condivisa.
Se le due parti **non riescono a stabilire uno scambio di chiavi iniziale sicuro, non saranno in grado di comunicare in modo protetto** senza il rischio che i messaggi vengano intercettati e decifrati da un terzo che ha acquisito la chiave durante lo scambio iniziale di chiavi.

7.4.2 Crittografia asimmetrica

Nella **crittografia asimmetrica** c'è un **sistema a due chiavi** (chiavi pubbliche e private). Il messaggio che viene **cifrato con una chiave deve essere decifrato con l'altra** e viceversa.
L'idea è che la **chiave pubblica possa essere inviata su un canale non sicuro** o condivisa in pubblico, mentre la chiave privata è disponibili solo al suo possessore.

Un tipico approccio è di usare **chiavi pubbliche per la cifratura e chiavi private per la decodifica**:

- Se l'intruso "sniffa" la chiave pubblica, è ancora impossibile decifrare il messaggio.
- L'host A userà la chiave B pubblica per cifrare i messaggi che possono essere letti solo dall'host B tramite la chiave B privata che è posseduta solo da B.

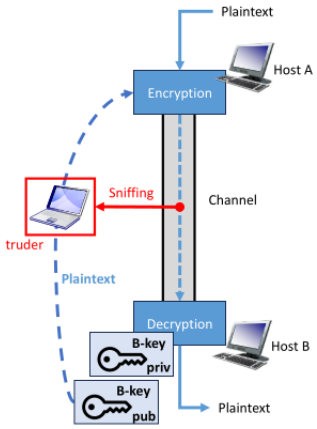


Figure 60: Crittografia asimmetrica

7.4.3 Certification Authority

Nella crittografia con chiave pubblica sarebbe molto utile **verificare se una chiave pubblica appartiene veramente all'entità** con cui vuoi comunicare. D'altro canto, **potremmo avere la chiave di qualcun altro** (attaccante) e potremmo cifrare i messaggi che sono leggibili ad entità non legittime.

L'azione di legare una chiave pubblica ad una particolare entità è tipicamente fatta da un **Certification Authority** (CA), il cui lavoro è di validare le identità e rilasciare certificati. Un CA ha i seguenti ruoli:

1. Un **CA verifica che un entità sia chi dice di essere**. Non c'è un protocollo per far ciò, **uno deve semplicemente fidarsi del fatto che il CA** abbia eseguito una rigorosa procedura di verifica dell'identità.
    - Funziona come un processo di selezione naturale: **se un CA non è affidabile, nessuno si affiderà a lui**.
    - Ci sono diversi **CA federali o statali** che forniscono una ragionevole affidabilità, ma dobbiamo ancora fidarci di loro.
2. Una volta che il CA verifica l'identità dell'entità, **il CA crea un certificato che lega la chiave pubblica dell'entità alla sua identità**. Il certificato contiene la chiave pubblica e un identificativo globale pubblico del possessore (un indirizzo IP o un hostname).

7.5 Integrità del messaggio

L'**integrità del messaggio** (anche conosciuta come **autenticazione del messaggio**) è il problema di verificare se:

1. Il messaggio **non è stato manomesso**.
2. Il messaggio è stato **effettivamente creato dall'host previsto**.

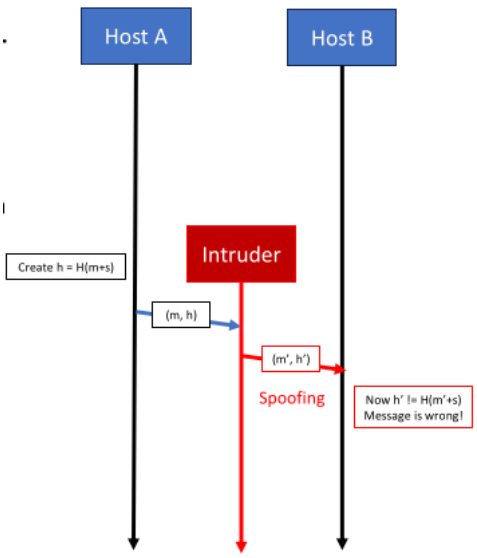
Possiamo creare un **check-item** simile al checksum o al CRC. Tipicamente, una funzione di hash è usata per creare tali item. (una **funzione di hash** è una qualsiasi funzione che può essere usata per **mappare dati di dimensione arbitraria in valori di lunghezza fissa**).
Una **funzione di hash crittografica** è una funzione H che converte un messaggio x in una stringa di lunghezza fissa H(x) tale per cui è molto difficile (computazionalmente impossibile) trovare un altro messaggio y tale che H(y)=H(x).

Come nel checksum o CRC, **una prima strategia può essere di allegare questo hash al messaggio**:

1. L'host **A crea il messaggio m e calcola l'hash h=H(m)**.
2. L'host **A aggiunge h al messaggio m**, creando un messaggio esteso (m,h), e lo **invia a B**.
3. L'host **B riceve il messaggio (m,h) e calcola H(m)**. Se H(m)=h, il messaggio è intatto.

Questo approccio è ovviamente fallace. Un intruso potrebbe **eseguire uno spoofing sull'intero messagio** (m,h), creandone **ad hoc uno nuovo** (m',h') che sia ancora consistente con la funzione di hash H.
Per evitare ciò, A e B necessitano di un **s condivisa e segreta** (come **una chiave o una password**) che è una stringa nota solo a loro. Ciò funziona come la cifratura simmetrica dove s è la chiave privata unica.
Ipotizzando che tale s esista, allora:

1. L'host **A crea un messaggio m+s** e **calcola l'hash h=H(m+s)** ovvero un **message authentication code (MAC)**.
2. L'host **A aggiunge il MAC al messaggio m** creando un messaggio esteso (m, H(m+s)), e lo invia a B.
3. L'host **B riceve il messaggio esteso** (m,h) e conoscendo s, **calcola il MAC H(m+s)**. Se H(m+s)=h, allora il messaggio è intatto



Come in tutti gli approcci simmetrici, dobbiamo **scambiare tale segreto**. Una possibile tecnica è **scambiarlo combinando crittografia asimmetrica e certificazioni**.

7.6 End-point Authentication

L'**end-point authentication** è il processo di **un'entità che prova la sua identità ad un'altra entità** all'interno di una rete.
Tipicamente, un **authentication protocol** (AP) dovrebbe essere eseguito **prima che le due parti che comunicano eseguano qualche altro protocollo**.

- È usato per **stabilire le identità** di entrambe le parti coinvolte **prima di cominciare a comunicare**.
- Esempi di questi protocolli possono essere un protocollo di trasferimento dati affidabile (basati su TCP), un protocollo di routing (basato su DV o LS, un protocollo e-mail (come SMTP), ecc.

Per eseguire la end-point authentication **possiamo fare affidamento sul certification authority** (CA) già introdotto in precedenza, ma non è abbastanza.

Ipotizziamo che l'**host A debba autenticarsi a B** prima che possano iniziare a lavorare insieme.
Se A e B hanno già comunicato e l'indirizzo IP di A è ancora lo stesso, **B dovrebbe autenticare A verificando l'indirizzo IP** all'interno del datagramma. **Se l'indirizzo è quello ben noto**, possiamo assumere che sia A ad inviare il messaggio.

Questa semplice tecnica di autenticazione **funziona per intrusi "naive"** che cercano di camuffarsi come A, ma non è sufficiente: un **intruso con più esperienza potrebbe forgiare l'indirizzo** nel pacchetto.
Come abbiamo visto, è possibile **forgiare un datagramma IP ed inserirvi uno specifico indirizzo IP** (spoofing).
Un modo di impedire l'IP spoofing è attraverso i router: un router potrebbe essere **configurato per inoltrare solo datagrammi legittimi**, ovvero, contenenti indirizzi IP sorgente che appartengono realmente all'host:

- Il router potrebbe essere consapevole degli indirizzi IP degli host. (altrimenti, gli host non sarebbero raggiungibili).
- Il router può rilevare i pacchetti i cui indirizzi IP non sono consistenti con quello sorgente.
- I datagrammi forgiati potrebbero essere semplicemente scartati dai router in modo tale che non possano più danneggiare nessuno.

Sfortunatamente, **questa capacità non è universalmente distribuita**, quindi dobbiamo assumere che lo spoofing sia possibile.

Un altro approccio è di usare una password segreta (scelta più comune): la password funziona come un **segreto condiviso** tra l'autenticatore e la persona che viene autenticata. Questo **segreto condiviso può essere usato anche per l'integrity check**.
Il primo problema è che **l'intruso potrebbe intercettare la password** dai messaggi, usando sia password che indirizzo IP forgiati.
Si può impedire che la password venga intercettata cifrandola, in modo tale che diventi illegibile per l'intruso. Si noti che **se usiamo la cifratura simmetrica** qui, è necessaria un'ulteriore password poiché la chiave privata stessa (che deve essere un segreto condiviso tra i 2 host) funziona come una password.

La crittografia è un buon approccio, ma non siamo ancora al sicuro. Un **intruso molto furbo potrebbe ancora essere in grado di intromettersi nella comunicazione** camuffandosi come A attraverso un **attacco playback**.
Un **attacco playback** (o attacco replay) l'intruso cerca di imitare l'host:

- L'intruso sniffa i pacchetti da A a B cercando di ottenere la versione **cifrata della password di A**.
- Se ha successo, l'**intruso potrebbe riprodurre** (play back) **la versione cifrata della password** a B in una nuova sessione, **usando la password anche senza capirla**.
- A differenza dagli attacchi MiM standard (che sono real-time), lo sniffing e il playback sono eseguiti separatamente.

Qui il problema è che l'**host B non è in grado di dire se A è live**, ovvero, se c'è **una sessione attiva** tra il vero A e B. **Questo problema è in qualche modo simile alla creazione di una connessione TCP** in cui i due host usano la three-way handshake per assicurare che entrambi siano vivi sui due capi.
Nella handshake TCP gli host usano numeri di sequenza iniziali casuali per impedire che possibili ritrasmissioni possano essere interpretati come segment SYN/ACK. Un'**idea simile può essere adottata** per l'autenticazione.

7.6.1 Nonce

In questo caso possiamo usare un **nonce**: un **numero casuale o pseudocasuale** che un protocollo usa **solo una volta nella sua vita**.
Esempio: assumiamo che A e B condividano una chiave simmetrica, il **nonce** può essere usato in questo modo:

1. L'host **A invia il messaggio "sono A" a B**.
2. L'host **B sceglie un nonce R** e lo invia ad A.
3. L'host **A cifra il nonce insieme alla password** usando la chiave simmetrica segreta e restituisce il nonce cifrato a B.
4. L'host B decifra il messaggio ricevuto. **Se il nonce decifrato corrisponde al nonce che ha inviato A**, allora A è autenticato.

Questa **password diventa una sorta di password usa e getta**:

- **Tutte le password cifrate hanno un numero nonce con esse**, quindi non possono essere riusati in altre sessioni o da qualcun altro.
- Se la **password + nonce è correttamente ricevuta da B**, possiamo ragionevolmente assumere che **sia stato A ad inviarla**.

7.7 SSL

Alcune tecniche di crittografia che abbiamo visto sono implementate attraverso il **Secure Socket Layer** (SSL), che è una **versione migliorata di TCP** che fornisce servizi di sicurezza.

- Ci sono anche versioni leggermente modificate di SSL versione 3, chiamato **Transport Layer Security** (TLS).
- C'è un equivalente per comunicazioni basate su UDP chiamato **Datagram TLS** (DTLS).

Spesso la connessione tra i browser e i siti web usano **HTTPS** (HTTP+SSL/TLS) piuttosto che HTTP.

SSL può **essere impiegato da qualsiasi applicazione che viene eseguita con TCP**. **SSL fornisce una semplice API** con le **socket**, che è simile ed analoga all'API di TCP. Quando un applicazione vuole impiegare SSL, deve includere le classi/librerie di SSL.
Nonostante **SSL tecnicamente appartenga al livello applicazione**, dalla prospettiva dello sviluppatore, **sembra un protocollo del livello di trasporto** che fornisce i servizi di TCP migliorati con i servizi di sicurezza.
SSL ha tre fasi principali: **handshake, key derivation, data transfer**.

7.7.1 SSL: Handshake

Assumiamo di avere un host client (B) che vuole usare SSL per comunicare con un server (A). Durante la **fase di handshake**, il client B deve:

- **Stabilire una connessione TCP** con A.
- **Verificare che A sia realmente A** (non un server falso).
- **Inviare ad A una chiave master segreta** (segreto condiviso).

Il processo funziona in questo modo:

- Una volta che la connessione TCP viene stabilita, **B invia ad A un messaggio di hello**.
- Il server **A risponde con un certificato**, che contiene la sua chiave pubblica (asimmetrica).
- Dato che il certificato è stato rilasciato da un CA, il client **B si fida del fatto che la chiave pubblica appartenga ad A**.
- **B quindi genera un Master Secret** (MS), che verrà **usato solamente nella sessione SSL (nonce)**, cifra l'MS con la chiave pubblica di A e la invia ad A.
- Il server **A decifra il messaggio con la chiave privata** per ottenere l'MS. Dopo questa fase, **solo B ed A conosce il master secret** per questa sessione SSL.

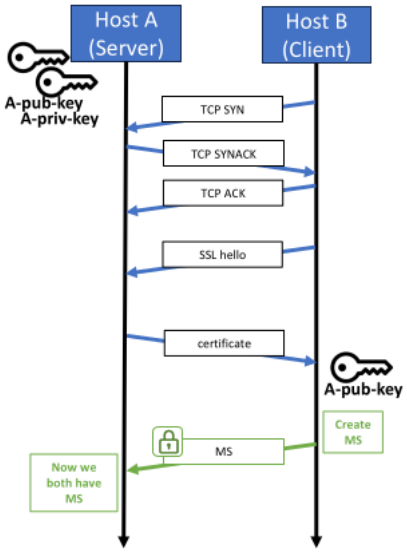


Figure 61: Handshake SSL

7.7.2 SSL: Key Derivation

Dato che **MS è un segreto condiviso tra B ed A**, potrebbe essere usato come **chiave simmetrica** per la cifratura e la data integrity checking durante la sessione.
D'altro canto, **è più sicuro per A e B di usare chiavi diverse**. **Possono essere create 4 chiavi** dall'MS:

- E_{B-to-A} = session **encryption key** per i dati inviati da B ad A.
- M_{B-to-A} = session **MAC key** per i dati inviati da B ad A.
- E_{A-to-B} = session **encryption key** per i dati inviati da A a B.
- M_{A-to-B} = session **MAC key** per i dati inviati da A a B.

7.7.3 SSL: Data Transfer

SSL **protegge TCP** suddividendo il flusso dati (dall'applicazione) in record. Per ciascun record:

1. SSL **aggiunge un MAC a ciascun record** per l'integrity checking (record$_i$+MAC)
2. Il **record modificato viene cifrato**.
3. Il **record cifrato viene passato al TCP** per la trasmissione.

Questo processo assicura l'integrità dei dati per i singoli record, **ma per l'intero stream?**
Dato che solo il payload viene cifrato, **un MiM tra A e B è ancora in grado di danneggiare il flusso invertendo o rimuovendo i segmenti**:

- Ad esempio, il MiM potrebbe catturare 2 segmenti consecutivi, poi invertire il loro ordine con i loro numeri di sequenza, infine inviare i 2 segmenti invertiti al ricevitore. Il ricevitore non può accorgersi dell'inversione finché i dati **non arrivano all'applicazione**.

Per impedire questo problema, **SSL integra anche un numero di sequenza nel messaggio** prima della cifratura:

- Il **messaggio cifrato sarà record$_i$+MAC+sequenceNumber$_i$**. Conoscendo questa procedura e il MAC, il **ricevitore dovrebbe essere in grado di certificare l'integrità del messaggio**.

7.8 Firewall

Un **firewall** è una combinazione di **hardware e software** che isola (protegge) una rete da Internet **consentendo/negando ai pacchetti** l'accesso.
Tutto il **traffico in uscita/in entrata dovrebbe passare attraverso il firewall** e solo il traffico autorizzato può passare. Il firewall funziona come un **access point singolo verso la rete pubblica**, ma organizzazioni grandi potrebbero avere livelli multipli o firewall distribuiti.
Il firewall stesso deve **essere immune a penetrazioni**. Se viene compromessi, esso può fornire un falso senso di sicurezza. Vediamo 3 approcci:

- Filtraggio dei pacchetti tradizionale.
- Filtraggio stateful.
- Application gateway.

7.8.1 Firewall: Filtraggio dei pacchetti tradizionale

In un **packet filter** è tipicamente implementato sul **gateway** (il router che connette la rete locale all'ISP). Esso **esamina ciascun datagramma**, determinando se il **datagramma debba essere accettato/scartato** in base alle specifiche **regole** adottate. Tali regole possono essere tipicamente basati su:

- Indirizzi IP sorgenti o di destinazione.
- Tipi di protocolli nel campo del datagramma IP (TCP, UDP, ICMP, ecc.).
- Porta sorgente e di destinazione
- Bit di flag TCP: SYN, ACK, ecc.
- ...

Un amministratore di rete configura il firewall in base alla **politica dell'organizzazione**. Alcuni esempi:

- **Consentire solo traffico dal Web** settando una regola che **blocca tutti i segmenti TCP SYN** che hanno porta di destinazione diversa da 80.
- **Negare servizi di streaming** settando una regola che **blocca il traffico UDP non critico**.
- **Negare ping in arrivo** settando una regola che **blocca risposte in uscita di tipo ICMP ping**.

Una politica di filtraggio può anche essere basato su una **combinazione di indirizzi e numeri di porta**. Le regole dei firewall sono implementate nei router con **liste di controllo degli accessi**.

C'è un problema con il filtraggio tradizionale: è **stateless**. Ad esempio, **una tabella che fa passare qualsiasi pacchetto** che arriva dall'esterno **con ACK=1 e la porta sorgente 80** è fallace perché tali pacchetti sono noti per essere usati negli **attacchi DoS**. La soluzione naive sarebbe quella di **bloccare i pacchetti TCP ACK**, ma tale approccio implicherebbe il fatto di **impedire ad utenti interni di navigare sul Web**.

7.8.2 Firewall: Stateful Packet Filter

I filtri stateful risolvono questo problema **tracciando tutte le connessioni TCP in corso** in una connection table, per capire se il traffico proviene da una connessione legittima:

- Il **firewall riconosce l'inizio di una nuova connessione e la sua fine**.
- Il firewall può anche (in modo conservativo) **assumere che la connessione sia finita** quando non viene vista nessuna attività sulla connessione per un certo lasso di tempo.

Una tabella stateful **tipicamente include una colonna "check connection"** che specifica se i pacchetti sono in una connessione stabilita.

7.8.3 Firewall: Application Gateway

I metodi di filtraggio visti precedentemente verificano per lo più le porte o gli indirizzi IP, ma può essere utile **consentire/negare funzionalità in base all'utente o l'applicazione**, ad esempio:

- Solo **i tecnici potrebbero avere il permesso di usare alcuni protocolli** che vengono negati agli utenti normali.
- Solo **specifici tipi di messaggi o comandi** vengono ammessi in un determinato protocollo.

Tali compiti sono aldilà delle capacità dei filtri tradizionali e stateful dato che informazioni sull'**identità dell'utente sono dati del livello applicazioni** e non sono inclusi negli header IP,TCP o UDP.

Un **application gateway** ci consente di filtrare pacchetti in base a dati del livello applicazione. Vengono tipicamente **implementati con un server separato che lavora in combinazione con il firewall**.
L'idea generale è di **negare tutte le connessioni di un protocollo specifico eccetto quelli che sono da/verso l'application gateway**.
Se un utente vuole usare l'application gateway **deve loggarsi** (con user ID e password).

- Qui il **server verifica che l'utente abbia il permesso** per quel protocollo.
- Se lo ha, **tutte le richieste risposte vengono inoltrate attraverso l'application gateway** (proxy).

Le reti interne **spesso hanno più application gateway** per più protocolli. (Telnet, HTTP, FTP, ...). Ci sono alcuni svantaggi:

- Un **diverso application gateway è necessario per ogni application**.
- Le **performance vengono penalizzate** a causa del proxying.
- I software sui **dispositivi dell'host devono sapere come usare l'application gateway**.

7.9 Sistemi Intrusion Detection

Al fine di rilevare alcuni tipi di attacchi, potremmo avere bisogno di eseguire **ispezioni approfondite sui pacchetti**, controllando quindi pattern di attacco noti o traffico sospetto. L'**Intrusion Detection System** (IDS) è un gruppo di dispositivi specializzati che monitorano la rete, alla ricerca di pacchetti sospetti.

- Il **ruolo primario dell'IDS è di riconoscere** pacchetti potenzialmenti malevoli **e allertare** l'amministratore di rete.
- In più, **è anche possibile filtrare** pacchetti potenzialmente malevoli. In questo caso, dovremmo riferirci ad esso come **Intrusion Prevention System** (IPS).

Le organizzazioni si affidano agli IDS per **rilevare un largo range di attacchi** come DoS, Worms e virus, nmap.
Il sistema è composto da:

- Uno o più **sensori IDS**.

- Un singolo **processore IDS** centrale che collezione e integra le informazioni e invia degli allarmi.

L'IDS può essere:

- **Signature-based**: gestisce un database estensivo di signature di attacchi, ovvero, **un set di regole relative ad un'attività d'intrusione**.
- **Anomaly-based**: crea un profilo del traffico e verifica gli stream che sono **statisticamente inusuali**.

7.10 Zona Demilitarizzata

In reti larghe c'è spesso il problema di **avere 2 diversi livelli di sicurezza** per diversi dispositivi. Ad esempio, **i server che devono comunicare con l'esterno** possono usare un filtraggio più morbido.

Una **zona demilitarizzata** (DMZ) è **una regione con una bassa sicurezza** in cui le restrizioni sono limitate.
Un approccio tipico è di mettere una **DMZ tra 2 firewall**: uno esterno (meno restrittivo) ed uno interno (più restrittivo).