



# Reti, Infrastruttura, Topologia e Categorie di reti

≡ Lezione	1
👤 By	👤 Simone Parente

## Introduzione

Una rete di computer è un insieme di computer connessi via cavo o wireless in modo da permettere lo scambio di informazioni.

## Infrastruttura

Una rete è composta, in linea generale, da due tipi di dispositivi:

- I dispositivi di rete ed i *link*, che permettono la connessione alla rete (Router, Switch, Hub).
- Gli *host*, sui quali sono attivi i servizi o che ospitano risorse (PC, Server, Smartphone, ecc.).

## Terminologia

I termini sono simili a quelli usati nella teoria dei grafi:

- I dispositivi sono detti *nodi*.

- I *link* sono gli archi che connettono tra loro i diversi nodi.
- Una sequenza di nodi e link è detta *path*.
- Le foglie del grafo sono i cosiddetti host.

## Comunicazione

---

Abbiamo 5 attori:

- *Messaggio*: che contiene i dati che vogliamo condividere.
- *Sorgente*: l'entità che invia il messaggio.
- *Destinazione*: l'entità che riceve il messaggio.
- *Medium*: il canale attraverso cui passa il messaggio.
- *Protocollo*: un insieme di regole condivise tra sorgente e destinazione.

## Tipi di trasmissione di dati

---

Esistono tre principali tipi di connessione:

1. *Simplex*: monodirezionale.
2. *Half-duplex*: bidirezionale, a turno.
3. *Full-duplex*: bidirezionale, simultanea.
  - La *velocità* di trasmissione è il massimo numero di informazioni che un **canale** può **trasmettere** in un'unità di tempo.
  - La *banda* è la massima quantità di informazioni che un **path** può **trasmettere** in un determinato periodo di tempo.
  - Il *throughput* è la quantità di informazioni effettiva che un path **trasmette** in un'unità di tempo.  
Le unità di misura utilizzate per misurare questi concetti sono: bits/sec o i bytes/sec.

## Tipi di connessione

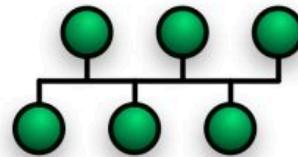
- Punto a punto: due dispositivi sono connessi tra loro da un link (via cavo o wireless).
- Multipunto o *broadcast*: più di due dispositivi condividono un singolo link.

## Topologia

Come sono organizzati i dispositivi, di seguito alcuni esempi:

### Bus

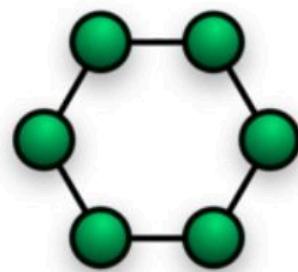
In questo tipo di connessione abbiamo un singolo link (*backbone*) a cui si connettono tutti i dispositivi. Nel caso in cui questo link principale dovesse smettere di funzionare tutti i dispositivi potrebbero essere disconnessi.



### Anello

Ogni nodo è connesso tramite due link ad altri due nodi, il segnale si propaga da un computer all'altro, fino a raggiungere la destinazione. Ha performance migliori della topologia bus, ma il malfunzionamento di un singolo nodo può compromettere le performance della rete.

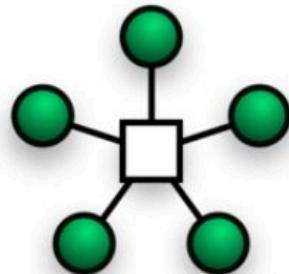
Queste reti hanno  
 $n$  link duplex.



### Stella

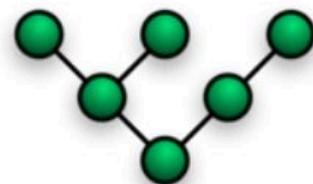
---

Tutti gli host sono connessi a un controller centrale (hub, switch o router), che reindirizza i messaggi.  
Sono dotate di un singolo controller e  $n$  link duplex.



## Albero

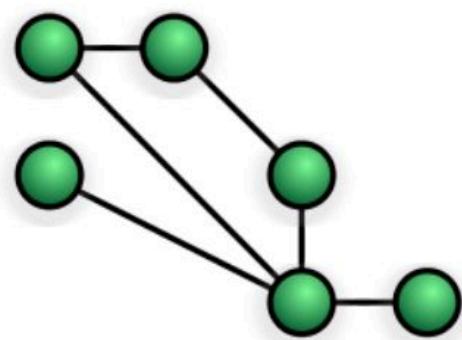
È praticamente una rete composta da una rete a stella "interna" e un bus "esterno" (che è praticamente il nodo padre).

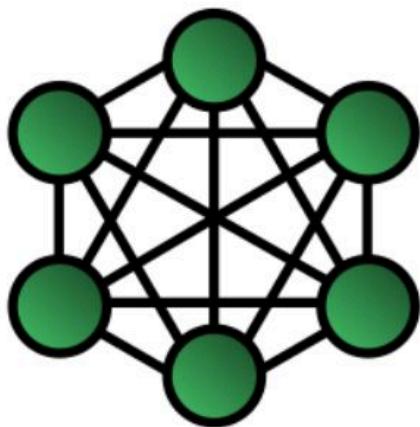


## Mesh

Si divide in:

- Full mesh: tutti i nodi sono connessi, avranno  $\frac{n(n-1)}{2}$  link duplex, l'equivalente di un grafo connesso.
- Partial mesh: i nodi sono connessi in maniera arbitraria.





## Categorie di reti

- *LAN*: Local Area Network
- *MAN*: Metropolitan Area Network
- *WAN*: Wide Area Network



# Risorse e servizi, Cloud Computing, SaaS, PaaS, IaaS, Internet of Things

≡ Lezione 2

## Risorse e servizi

Il ruolo di una rete di computer è quello di permettere lo scambio di dati tra diversi dispositivi geograficamente distanti tra loro.

Questi dati sono il *mezzo* attraverso cui passano le informazioni che permettono di utilizzare i servizi e le risorse messe a disposizione dalla rete.

- Una **risorsa** è un *object* remoto (in generale, file di qualsiasi tipo).
- Un **servizio** è un'azione che un dispositivo esegue da remoto per noi (*Gmail*, che ci permette di inviare e ricevere e-mail, ad esempio).

Le entità che offrono questi servizi su internet sono chiamate **Service Providers**.

## ISP: Internet Service Providers



Gli *ISP* sono delle organizzazioni che si occupano della vendita di servizi che, tra le altre cose, permettono l'accesso alla rete Internet.

È definito gerarchicamente:

1. **Tier-1**: Sono gli ISP di livello superiore che hanno le proprie reti globali e possono raggiungere qualsiasi punto su Internet senza intermediari.
2. **Tier-2**: Sono ISP intermedi che hanno infrastrutture regionali o nazionali e dipendono dai Tier-1 per raggiungere alcune destinazioni su Internet.
3. **Tier-3**: Sono ISP di dimensioni più piccole che forniscono servizi a livello locale o regionale e dipendono dai Tier-1 e/o Tier-2 per raggiungere molte destinazioni su Internet.

I diversi ISP si scambiano i dati attraverso i cosiddetti ***Internet Exchange Point*** (o ***Neutral Access Points***), semplificando molto, possiamo vedere questi punti di interscambio come i router di una rete domestica a cui sono collegati diversi dispositivi.

## Tipi di connessione

Le reti private si basano su dei **modem** abilitati dalla rete telefonica.

- Analogica (56kb/s)
- ISDN (128 kb/s)
- ADSL (da 1Mb/s a 20Mb/s)
- Cavo di rame (da 10Mb/s a 100Mb/s)
- Fibra ottica (da 50Mb/s a 40Gb/s)

Le reti aziendali potrebbero avere una connessione diretta/dedicata all'ISP.

## Grid computing

---

Per grid computing si intende un sistema di condivisione di risorse che combina diversi hardware geograficamente distanti tra loro, in modo tale da ottenere una maggiore potenza di calcolo per effettuare operazioni complesse o salvare grandi quantità di dati.

## Cloud computing

---

Il cloud computing è un modello che consente un accesso costante, conveniente, su richiesta e tramite rete a un insieme condiviso di risorse informatiche configurabili (reti, server, servizi, ecc.) che possono essere rapidamente fornite e rilasciate con il minimo sforzo di gestione.

Notion in questo momento sta offrendo un servizio di cloud computing.

### Caratteristiche principali

---

1. **Servizio Self-Service su richiesta:** un consumatore può richiedere capacità computazionali (ad esempio spazio di archiviazione cloud) autonomamente e senza bisogno di interazione umana.
2. **Accesso tramite rete:** Queste capacità computazionali sono fornite tramite meccanismi standardizzati che promuovono un uso da parte di utenti il più eterogenei possibili.

3. **Pooling delle risorse:** Le risorse informatiche del fornitore sono raggruppate per servire più consumatori allo stesso tempo, con diverse risorse fisiche e virtuali assegnate e riassegnate dinamicamente in base alla domanda dei consumatori.
4. **Elasticità delle risorse:** Queste risorse possono essere fornite e rilasciate in maniera elastica, ancor meglio se automaticamente.
5. **Servizio misurato:** I sistemi cloud controllano e ottimizzano automaticamente l'uso delle risorse sfruttando una misurazione a un livello di astrazione appropriato al tipo di servizio offerto.

## Modelli di servizi

---

Il National Institute of Standards and Technology ha raggruppato i servizi forniti da un'architettura cloud in tre principali categorie:

1. **Software as a Service (SaaS):** si parla di intere applicazioni fornite insieme alle componenti hardware e software per farle funzionare. (Si pensi a tutti i servizi Google).
2. **Platform as a Service (PaaS):** piattaforme fornite con specifiche configurazioni (sistemi operativi, API, ecc.).
3. **Infrastructure as a Service (IaaS):** tipo di servizio che offre risorse di calcolo, archiviazione e di rete.

## Internet of Things

---

Per **Internet of Things (IoT)** si intende la rete di oggetti fisici dotati di sensori, processori, connettività, ecc. che permette il loro controllo da remoto tramite Internet.



# Standard, Modello a strati, Modello ISO-OSI, Modello TCP-IP

≡ Lezione 3

## Standardizzazione

Si rese da subito necessaria la creazione di standard (che chiameremo anche **protocolli**) per la comunicazione tra dispositivi, in modo tale da ridurre al minimo la possibilità di errori.

Esistono centinaia di protocolli che regolano **qualunque** aspetto della comunicazione tra dispositivi, dai collegamenti fisici tramite cavo alle applicazioni *top-level*.

La definizione di questi standard è gestita dalla **Internet Engineering Task Force**, organizzata in gruppi, ogni gruppo si occupa di uno specifico aspetto della rete e crea dei documenti chiamati *Request For Comments (RFC)* in cui vengono definiti e descritti questi protocolli.

## Modello a livelli

Come nella maggior parte dei casi, l'approccio utilizzato per affrontare i diversi problemi di comunicazione è il **Dividi et Impera**, creando un modello a layer:

- Ogni layer si occuperà di uno specifico task.

- Ogni layer *si basa* sui servizi del layer sottostante.
- Ogni layer *fornisce servizi* al layer sovrastante.

## Il modello ISO-OSI

Negli anni '80 l'***International Standards Organization*** definisce un modello a livelli per le reti di computer: il ***Open System Interconnection model***.

È un modello a 7 livelli, dove ogni livello si occupa di un compito molto specifico:

1. **Fisico**: Trasmissione fisica dei bit da un nodo a un altro.
2. **Collegamento dati**: Definizione del formato dei dati.
3. **Rete**: Routing (instradamento) dei pacchetti attraverso la rete.
4. **Trasporto**: Fornire un canale logico di comunicazione *end-to-end* per pacchetti.
5. **Sessione**: Coordinare dialogo e sincronizzazione di richieste e risposte.
6. **Presentazione**: Traduzione, codifica (sia in uscita che in entrata), e compressione dei dati.
7. **Applicazione**: Il livello che fornisce direttamente servizi all'utente.

È bene ricordare che nel momento in cui due dispositivi comunicano, **ogni layer del mittente comunica con il rispettivo layer del destinatario** tramite uno specifico protocollo.

Non sempre vengono implementati tutti i diversi livelli in ogni dispositivo, ad esempio nei "nodi intermedi" (router, switch) sono implementati solo i livelli 1,2,3, questo perché, tra le

altre cose, non è necessario che questi dispositivi interagiscano direttamente con esseri umani.

## Incapsulamento e Decapsulamento

---

### Incapsulamento (Top-Down)

Ogni layer del mittente aggiunge uno specifico campo al messaggio, in testa o in coda.

### Decapsulamento (Bottom-Up)

Ogni layer del ricevente rimuove questi campi e li interpreta prima di inviarli al livello successivo.

## Modello TCP-IP

---

Il modello **TCP-IP** (*Transmission Control Protocol, Internet Protocol*) è l'insieme dei protocolli di comunicazione effettivamente utilizzati per le reti locali e per Internet, esso è *di fatto* lo standard per la comunicazione via Internet.



# Livello Applicazione

≡ Lezione 4 → 9

Network Applications

Esempio

Tipi di architetture

Client-Server

Peer-to-Peer (P2P)

Comunicazione tra processi

Sockets

Esempio della cassetta delle lettere

Protocolli

Protocolli più comuni

File Transfer Protocol (Secure)

Telnet e Secure SHell

World Wide Web

URL (Uniform Resource Locator)

Esempi di URL

HTTP

Statelessness

Connessioni persistenti e non persistenti

Connessione persistente

Connessione non persistente

Round-Trip Times

Three-Way Handshake

Pipelining

Formato dei messaggi

Richieste

Risposta

Cookies

Cache

SMTP - Simple Mail Transfer Protocol

Funzionamento

## Protocolli aggiuntivi

POP3 - Post Office Protocol 3

IMAP - Internet Mail Access Protocol

HTTP

DNS - Domain Name System

Gerarchia

Funzionamento

DNS Resolver

Come si crea una network application

Creazione di un socket

Binding del socket a un indirizzo locale

Invio e ricezione di messaggi (UDP)

Chiusura di un socket

Da UDP a TCP

Connessione (TCP)

Wait-for-connection server side

Invio e ricezione (TCP)

DNS Translation

# Network Applications

---

Una *network application* è composta da diversi programmi che vengono eseguiti su diversi host e comunicano tra loro tramite la rete.

## ▼ Esempio

Prendiamo come esempio una applicazione web, questa è composta da due programmi distinti:

1. Il programma eseguito sul browser (che a sua volta è eseguito sul dispositivo) dell'utente.
2. Il programma eseguito sul server.

In principio, creare un'applicazione significa scrivere dei programmi (esclusivamente per gli host) che:

- Funzionano su diversi sistemi, potenzialmente usando diversi linguaggi o sistemi operativi.
- Comunicano tramite rete tramite specifici protocolli.

## Tipi di architetture

---

L'architettura di un'applicazione è in genere definita dallo sviluppatore dell'applicazione e specifica *come* i diversi dispositivi sono connessi tra loro (tramite l'applicazione).

### Client-Server

---

In un'architettura client server abbiamo due tipi di host:

- **Server**: un host sempre online che si occupa di fornire servizi al client.
- **Client**: host che accede ai servizi forniti dal server, i client non comunicano tra loro.

Il server ha un *indirizzo IP* fissato, in modo da rendere possibile per i client di inviare richieste in qualsiasi momento.

In linea generale, vengono utilizzati diversi server in simultanea per gestire le diverse richieste dei client, nonostante i client siano ignari dell'esistenza di diversi server.

### Peer-to-Peer (P2P)

---

Gli host sono in generale omogenei e comunicano "a coppie".

È un tipo di architettura utilizzata per applicazione ad alta intensità di traffico, non esiste alcun server con un indirizzo IP fissato, ma ogni client ne ha uno proprio.

Applicazioni puramente P2P non sono molto comuni, spesso vengono utilizzate delle soluzioni ibride, combinando l'architettura client-server a quella P2P.

I pro di quest'architettura sono i bassi costi (non c'è bisogno di infrastrutture particolari), la scalabilità e il fatto che sia un tipo di architettura distribuita.

Questi pro sono controbilanciati da problemi di sicurezza, performance e affidabilità.

## Comunicazione tra processi

---

La maggior parte delle applicazioni consiste in coppie di processi (processi client e processi server) che comunicano tra loro inviando e ricevendo messaggi dall'uno e dell'altro tramite la rete sottostante.

## Sockets

---

Esistono delle interfacce software, dette **socket**, che permettono lo scambio di messaggi tramite la rete, ricordando l'architettura TCP/IP, un socket è l'interfaccia che permette la comunicazione tra il livello applicazione e quello di trasporto.

### ▼ Esempio della cassetta delle lettere

Possiamo considerare i socket come delle cassette delle lettere, quando un processo ha intenzione di inviare un messaggio a un altro processo, il primo inserisce questo messaggio in una cassetta postale. Il mittente assume che esista un'infrastruttura di trasporto che si occuperà di far arrivare il messaggio a destinazione.

Questi socket sono utilizzati come delle API tramite applicazioni e rete, lo sviluppatore dell'applicazione ha controllo su tutto ciò che riguarda il socket dal lato del livello applicazione, ma quasi nessun controllo sul socket sul

lato del livello di trasporto, le uniche scelte che può effettuare sono:

- La scelta del protocollo di trasporto (TCP/UDP)
  - Settare qualche parametro quali: buffer massimo e dimensione massima dei pacchetti.
- 

Ampliando l'esempio della cassetta postale, i processi hanno bisogno di un indirizzo a cui inviare i dati, visto che su uno stesso host possono essere in esecuzione diverse applicazioni, è necessario specificare:

1. L'**indirizzo IP** dell'host (livello rete).
2. La **porta** a cui il processo è connesso (livello trasporto).

## Protocolli

---

Un protocollo del livello applicazione si occupa di definire come i processi si inviano messaggi a vicenda, in particolare definisce:

1. Il **tipo** di messaggi scambiati.
2. La **sintassi** dei messaggi (ad esempio i campi del messaggio e come questi sono **delineati?**).
3. La **semantica** dei messaggi (il significato delle informazioni nei campi)
4. Le **regole** per determinare come e quando un processo invia messaggi e risponde a questi ultimi.

## Protocolli più comuni

---

- **DHCP**: si occupa di assegnare gli indirizzi IP (Dynamic Host Configuration Protocol)
- **DNS**: traduce i nomi dei siti web in indirizzi IP (Domain Name System)

- **HTTP/HTTPS**: si occupa del trasferimento di dati tra pagine web (HyperText Transfer Protocol)
- **SMTP/SMTPS**: permette di inviare email (Simple Mail Transfer Protocol)
- **SNMP**: gestione dei dispositivi della rete (Simple Network Management Protocol)
- **Telnet/SSH**: permette di comunicare con altri host via linea di comando (Teletype Network/Secure SHell)
- **FTP/FTPS**: rende possibile lo scambio di file (File Transfer Protocol)

Le **S** indicano i protocolli in versione *sicura*, necessari per le applicazioni che scambiano informazioni sensibili

## File Transfer Protocol (Secure)

---

È uno dei primi protocolli creati, è utilizzato per lo scambio di file tramite rete. Nella versione *non-secure*, i dati sono scambiati in chiaro, quindi è bene utilizzare questa versione solo su reti locali o applicazioni private. Entrambe le versioni sono dotate di due componenti:

1. Il protocollo che specifica i comandi (come show, get, delete, ecc.).
2. Un software che implementa il protocollo (uno lato client, uno lato server).

## Telnet e Secure SHell

---

Telnet è un protocollo che permette all'utente di iniziare sessioni di login remoto. SSH è la sua versione *secure*.

Entrambi i protocolli hanno due componenti:

1. Il protocollo che specifica come i messaggi sono strutturati e come gli host comunicano in una sessione.

2. Un software che implementa il protocollo (uno lato client, uno lato server).

## World Wide Web

---

Il World Wide Web è un insieme di informazioni che a cui è possibile accedere tramite la rete Internet tramite un protocollo chiamato *HyperText Transfer Protocol (HTTP)*.

La comunicazione *HTTP* è basata generalmente su architetture client-server:

- un **client** che **traduce** le richieste dell'utente in *messaggi HTTP*
- un **server** che esegue *richieste HTTP* e ritorna *risposte HTTP*.

## URL (Uniform Resource Locator)

---

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

- **protocol** è il protocollo da usare per accedere alla risorsa (HTTP, HTTPS, FTP, ...).
- **usrinfo** (opzionale e quasi sempre deprecato a causa di problemi di sicurezza) sono informazioni sull'utente (username e password, per esempio).
- **host** è il nome o l'indirizzo IP del server.
- **port** (opzionale) è la porta da utilizzare (di solito dedotto dal protocollo).
- **path** è il path della risorsa sul server.
- **query** è preceduto da "?" e specifica possibili richieste.

- *fragment* preceduto da "#" serve per identificare uno specifico elemento nella risorsa.

La maggior parte delle pagine è formata da una base HTML e diversi riferimenti a altri object (immagini, video, ecc.).

## Esempi di URL

---

| <https://qualcosa.dominio/pagina/immagine.jpg>

- *https* è il **protocollo**.
- *qualcosa.dominio* è il nome dell'**host**.
- */pagina/immagine.jpg* è il **path** dell'oggetto a cui vogliamo accedere.

[https://en.wikipedia.org/w/index.php?title=SSC\\_Napoli](https://en.wikipedia.org/w/index.php?title=SSC_Napoli) dove

- tutto ciò dopo ? è la **query**  
è equivalente a

[www.en.wikipedia.org/wiki/SSC\\_Napoli](http://www.en.wikipedia.org/wiki/SSC_Napoli)

## HTTP

---

Per il trasporto dei dati, *HTTP* utilizza *TCP* come protocollo "sottostante", uno dei motivi è l'**affidabilità** di *TCP* per il trasferimento dei dati, in questo modo le applicazioni possono essere più semplici, delegando il più del lavoro al livello sottostante.

## Statelessness

---

| Una applicazione è **stateless** se **non conserva informazioni sulle interazioni passate**.

### ▼ Esempio

Se un client richiede lo stesso object due volte, il server non considera questa richiesta ridondante, bensì re-invia l'object come se avesse "dimenticato" l'interazione.

Un'applicazione che non è stateless, che quindi ricorda le interazioni passate, viene detta **stateful**.

## Connessioni persistenti e non persistenti

---

In base al tipo di applicazione, può essere necessario che la connessione tra client e server venga mantenuta per un periodo di tempo più esteso, abbiamo quindi due tipi di connessioni: **persistenti e non persistenti**.

### Connessione persistente

---

Tutte le richieste e le rispettive risposte vengono scambiate **sulla stessa connessione TCP**.

In linea generale, le applicazioni HTTP utilizzano connessioni persistenti.

Il server dopo aver inviato una risposta al client, non chiude la connessione.

### Connessione non persistente

---

Per ogni coppia richiesta-risposta HTTP, viene stabilita una nuova connessione TCP.

### ▼ Esempio di funzionamento di una connessione non persistente

I primi 4 step sono ripetuti per ogni object.

1. Il client HTTP inizializza una connessione TCP col server, avremo 2 socket, uno per il client e uno per il server.

2. Il client HTTP invia una richiesta HTTP al server, includendo il *path name*.
3. Il server HTTP riceve la richiesta e recupera l'object relativo al path ricevuto.
4. I processi del server inviano una richiesta TCP di chiudere la connessione. TCP terminerà la connessione non appena avrà la certezza che il client abbia ricevuto correttamente la risorsa richiesta.
5. Il client riceve il messaggio di risposta, la connessione termina.

## Round-Trip Times

---

È possibile stimare il tempo di una connessione in termini di *round-trip times*.

L'

*RTT* è il tempo necessario per un pacchetto per andare dal client al server e ritornare al client: le connessioni TCP garantiscono questa cosa, ogni volta viene effettuato un *three-way handshake*, sia quando viene aperta la connessione, sia quando viene chiusa (non funzionano esattamente allo stesso modo).

## Three-Way Handshake

---

Il funzionamento è il seguente:

1. Il client invia un piccolo *segment TCP* al server, per segnalare la richiesta di connessione.
2. Il server riceve questo *segment* e risponde con un ulteriore *segment TCP*.
3. Il client riceve questo *segment*.

Le prime due parti del three-way handshake impiegano un RTT.

Quando la richiesta arriva al server, esso invia il file HTML, questa sequenza richiesta-risposta necessita di un ulteriore

RTT.

Quindi il tempo totale necessario al trasferimento di un file è di due RTT + il tempo di trasmissione del file.

## Pipelining

---

È possibile per il client effettuare richieste anche senza aver ricevuto risposte per le precedenti richieste.

Questo tipo di connessioni è la modalità di default di trasmissione per HTTP.

## Formato dei messaggi

---

Il protocollo HTTP prevede due formati diversi per le richieste e le risposte HTTP.

### Richieste

---

1. `Method` : specifica il metodo da eseguire: `GET`, `POST`, `HEAD`, `PUT` o `DELETE`.
2. `URL` : specifica l'URL dell'oggetto su cui stiamo effettuando la richiesta
3. `version` : versione del protocollo HTTP
4. `header lines` : contiene i parametri della richiesta
5. `body` : contiene dati associati al metodo scelto

### Risposta

---

1. `version` : versione HTTP
2. `code` : status code del comando inviato nella richiesta:
  - `100-199` `informational`
  - `200-299` `success`
  - `300-399` `redirect`
  - `400-499` `client error`

500-599 server error

3. phrase : risultato della richiesta

## Cookies



Un cookie è un token digitale usato da un server per identificare specifici client

Quattro componenti principali.

1. Set-cookie header nella prima risposta HTTP del server.
2. "Cookie" header nelle richieste HTTP.
3. Un file presente nella memoria del client.
4. Un database del server che tiene traccia dei cookie generati.

## Cache



Una web cache è un'entità di rete che ha il compito di conservare le copie degli object richiesti di recente e fornirli in caso vengano richiesti nuovamente.

- Utile per ridurre il carico di lavoro sul server e i tempi di attesa del client.
- Potrebbe causare errori o inconvenienti se la copia salvata in cache è particolarmente datata

## SMTP - Simple Mail Transfer Protocol



Il Simple Mail Transfer Protocol è il protocollo su cui si basa l'invio e la ricezione delle e-mail.

Due elementi coinvolti:

1. **User-agent** (applicazione che permette la gestione delle mail).
2. **Mail server** (che contiene le mailbox degli utenti).

## Funzionamento

Gli utenti non si scambiano direttamente mail ma si appoggiano sui mail server.

In SMTP ci sono un lato client e un lato server in esecuzione sui mail server.

### ▼ Esempio di comunicazione tra due utenti

Supponiamo di avere due utenti A e B e che A voglia inviare una mail a B.

1. A, utilizzando il proprio user-agent, scrive la mail e la invia
2. L'user-agent di A si occupa di inviare la mail al mail server, inserendolo in una coda.
3. Il lato client di SMTP, che è in esecuzione sul mail server di A, apre una connessione con il mail server (lato server) di B.
4. Dopo l'handshake, la mail viene inviata tramite una connessione TCP.
5. Il mail server (lato server) di B riceve il messaggio.
6. B può usare il proprio user-agent per leggere il messaggio.

## Protocolli aggiuntivi

---

Questi protocolli sono necessari per accedere alle mail

### POP3 - Post Office Protocol 3

---

L'user-agent apre una connessione TCP col mail server, dopodiché abbiamo 3 fasi:

1. **Authorization**: l'user-agent autentica l'utente inviando username e password al server.
2. **Transaction**: è possibile effettuare azioni basilari sui messaggi.
3. **Update**: la sessione termina e le azioni basilari effettuate al punto 2. vengono eseguite e sovrascritte.

### IMAP - Internet Mail Access Protocol

---

Questo protocollo fornisce ulteriori funzionalità come creare cartelle, effettuare ricerche e permette a più client di connettersi allo stesso server

### HTTP

---

In questo caso l'user-agent è un browser, mentre i mail server continuano a utilizzare lo standard SMTP.

### DNS - Domain Name System

---



Il Domain Name System è un protocollo client-server che si occupa di tradurre gli hostname in indirizzi IP.

È un sistema distribuito e gerarchico, così da:

- Evitare single points of failure

- Regolare il traffico
- Facilitare la manutenzione

## Gerarchia

---

1. Root DNS Servers
2. Top-Level Domain servers
3. Authoritative DNS

I root DNS servers forniscono gli indirizzi IP dei server Top-Level Domain, a loro volta i TLD forniscono gli IP per gli authoritative DNS.

## Funzionamento

---

1. Il client invia un hostname a uno dei root server, al client viene ritornato l'IP di un Top-Level Domain (in base al dominio dell'hostname richiesto)
2. A questo punto il client invia l'hostname al TLD, al client ritorna l'IP di un authoritative server.
3. A questo punto il client invia l'hostname all'autoritative server, che ritorna l'indirizzo IP richiesto.

## DNS Resolver

---



Server DNS locali generalmente gestiti dagli ISP, si comportano come dei proxy inviando la query alla gerarchia dei server DNS vista prima. Quindi l'utente invia la richiesta, il DNS resolver si occupa di scambiarsi le informazioni con la gerarchia dei server DNS e ritorna l'IP richiesto.

# Come si crea una network application



La maggior parte delle applicazioni sono basate su un architettura client-server.

Quando i due programmi vengono eseguiti, vengono creati due processi, uno client e uno server che comunicano tra loro tramite un **socket**.

I protocolli usati per il trasporto possono essere:

- **TCP**, protocollo orientato alla connessione e garantisce l'affidabilità dei dati: i dati inviati vengono confermati e ricevuti nell'ordine corretto.
- **UDP**, protocollo senza connessione e non garantisce l'affidabilità. Non effettua ritrasmissioni in caso di perdita di pacchetti e non garantisce che i pacchetti arrivino nell'ordine corretto

## Creazione di un socket

Iniziamo introducendo le librerie e una definendo una struct che ci permetterà la gestione del socket

```
#include <netinet/in.h>
#include <arpa/inet.h> // contiene inet_addr()
#include <sys/types.h> // contiene alcuni tipi custom
struct sockaddr_in {
    short sin_family;    // tipicamente settato a AF_INET (IPv4)
    unsigned short sin_port; // port number, e.g. htons(3490)
    struct in_addr sin_addr; // vedi struct in_addr sotto
    char sin_zero[8]; // tipicamente zeros, ora questa struttura
```

```
//e può essere castato in un in_addr  
};  
  
struct in_addr {  
    unsigned long s_addr; // IP address, can be loaded with inet_aton  
};
```

Da qui in poi socket vengono creati tramite la funzione

```
socket(int domain, int type, int protocol)
```

che si trova in `sys/socket.h`, il parametro `type` specifica il tipo di socket creato:

- `SOCK_STREAM` per socket TCP
- `SOCK_DGRAM` per socket UDP
- ▼ per `domain` si utilizza `AF_INET`

**Address Family: Internet** e viene utilizzata per specificare il tipo di indirizzi con cui il socket può comunicare, in questo caso indirizzi IPv4.

Quando `protocol` è `0` non viene specificato nessun protocollo in particolare.

## Binding del socket a un indirizzo locale

Tramite la funzione

```
bind(int socket, const struct sockaddr *address, socketlen_t address_len)
```

è possibile associare il socket a un indirizzo locale e una porta.

- `socket` è la variabile in cui il socket è stato precedentemente salvato

- `address` sarà spesso INADDR\_ANY così che vengano accettate connessioni da qualsiasi IP

Ritorna 0 se il binding è effettuato con successo, -1 altrimenti.

Quando la funzione viene utilizzata per un socket lato server è necessario specificare la porta, lato client è possibile ometterla perché viene assegnata automaticamente dal sistema operativo.

## Invio e ricezione di messaggi (UDP)

---

L'invio e la ricezione di messaggi su socket UDP è effettuata rispettivamente dalle funzioni

```
sendto(int osock, const void *obuf, size_t olen, int oflags,
       const struct sockaddr *oaddr, socklen_t oaddr_len)
recvfrom(int isock, void *ibuf, size_t ilen, int iflags, struct sockaddr *iaddr,
         socklen_t *iaddr_len)
```

dove:

- `osock / isock`: sono i file a cui inviare o da cui ricevere i messaggi
- `obuf / ibuf`: sono i puntatori ai buffer contenenti i messaggi da inviare/ricevere
- `olen / ilen`: lunghezze dei messaggi in byte
- `oflags / iflags`: tipicamente vengono passati 0 oppure `MSG_WAITALL` per aspettare finché tutti i messaggi sono stati inviati/ricevuti
- `oaddr_len / iaddr_len`: lunghezza in byte della struttura `sockaddr`

La funzione ritorna il numero di byte effettivi inviati/ricevuti

## Chiusura di un socket

---

È possibile chiudere un socket usando la funzione

```
close(int socket)
```

che prende in input un socket e ritorna 0 se la chiusura avviene con successo, -1 altrimenti.

## Da UDP a TCP

Prima di stabilire una connessione TCP, client e server devono effettuare un handshake, utilizziamo quindi due socket lato server:

1. Un primo socket, **sempre attivo**, che accetta richieste di connessione con i client ed effettua l'handshake preliminare
2. Un socket **client-specific**, creato dopo l'handshake, per permettere a client e server di comunicare tra loro.

NB: il three-way-handshake che viene effettuato al livello trasporto è totalmente invisibile per client e server

Non appena l'handshake preliminare ha successo, la connessione si sposta sul secondo socket.

## Connessione (TCP)

La connessione lato client viene effettuare tramite la funzione

```
connect(in socket, const struct sockaddr *address, socklen_t address_len);
```

dove:

- `socket`: variabile in cui è stato salvato il socket
- `address`: puntatore alla struttura contenente l'IP del server
- `address_len`: specifica la lunghezza del socket (possiamo utilizzare la funzione `sizeof()`)

La funzione ritorna 0 se la connessione è stabilita con successo, -1 altrimenti

## Wait-for-connection server side

---

Abbiamo due funzioni `listen()` e `accept()`, utilizzate in combinazione con `connect()`.

- `listen(int socket, int backlog)`
- `accept(int socket, struct sockaddr *address, socklen_t *address_len)`

dove:

- `socket`: è la variabile contenente il socket da cui attendere le connessioni
- `backlog`: indica la massima lunghezza della coda delle connessioni in sospeso
- `address`: può essere null oppure un pointer ad una struttura `sockaddr` dove l'indirizzo del socket deve essere ritornato
- `address_len`: lunghezza dell'indirizzo

`listen()` ritorna 0 in caso di successo, -1 altrimenti  
`accept()` ritorna il nuovo socket su cui client e server comuniceranno.

## Invio e ricezione (TCP)

---

L'invio e la ricezione di messaggi avviene tramite le funzioni

```
send(int osock, const void *obuf, size_t olen, int flags)
read(int isock, void *ibuf, size_t ilen
)
```

dove:

- `osock / isock`: file descrittori del socket su cui inviare/ricevere un messaggio
- `obuf / ibuf`: puntatori ai buffer contenenti i messaggi da inviare/ricevere
- `flags`: dipende dal protocollo, in genere 0

Le funzioni ritornano il numero di byte effettivamente inviati/ricevuti.

inserire immagine fine slide

## DNS Translation



Abbiamo visto che per creare dei socket per permettere la comunicazione tra due host dobbiamo conoscere IP e porta del server. Ma su internet, generalmente, gli host sono definiti da nomi più che da indirizzi, mentre i socket Berkeley (visti poco fa) funzionano unicamente tramite indirizzi IP.

Di conseguenza, per connetterci a un host tramite un hostname, dobbiamo tradurre l'hostname in indirizzo IP

In C e C++ esistono delle funzioni che permettono di contattare i server DNS e effettuare così una traduzione da hostname a IP, in particolare

```
gesthostbyname(const char *name)
```

- `name` : stringa contenente il nome da tradurre

La funzione ritorna una struct `hostent` contenente alcune info sull'host:

- Nome canonico
- Possibili alias
- Uno o più IP

La struct è definita come segue:

```
struct hostent{
    char *h_name;          //nome canonico
    char **h_aliases;      //alias
```

```
int h_addrtype;      //famiglia di indirizzi (tipicamente AF_INET)
int h_length;        //lunghezza degli indirizzi (tipicamente 4 o 16)
char **h_addr_list; //lista degli indirizzi (terminata con NULL)
};
```

Esiste una funzione

```
gethostbyaddr(const void *address, size_t length, int type)
```

dove:

- `address`: indica l'IP dell'host
- `length`: indica la lunghezza dell'indirizzo
- `type`: specifica il dominio dell'indirizzo, può essere `AF_INET` (IPV4) o `AF_INET6` (IPV6).



# Livello trasporto

≡ Lezione 10→13

Livello trasporto

Servizi

Protocolli

Responsabilità

Porte

Multiplexing e Demultiplexing

Multiplexing

Demultiplexing

In UDP

In TCP

Perché usare UDP e non TCP?

UDP Segment

Checksum

Affidabilità del trasferimento

Corruzione dei pacchetti

Approccio stop-and-wait

Corruzione o perdita dei pacchetti

Approccio tramite timeout

Performance

Approccio stop-and-wait

Approccio tramite pipeline

Go-Back N

Selective Repeat

Buffer TCP

TCP Segment

Three-way handshake

Stabilire una connessione

Chiudere una connessione

Flusso di controllo

Funzionamento

# Livello trasporto

## Servizi

Il livello di trasporto offre dei protocolli che garantiscono:

- **Affidabilità** nel trasporto dei dati.
- **Throughput**: il volume a cui il processo mittente invia i dati al processo destinatario.
- **Timing**: ogni bit inviato arriva al socket ricevente entro un certo tempo.
- **Sicurezza**: **encryption** e **decryption** dei dati.

## Protocolli



Un protocollo del livello di trasporto si occupa principalmente di fornire una comunicazione logica tra processi in esecuzione su host diversi.

In particolare, i messaggi ricevuti dal livello sovrastante (applicazione) vengono convertiti in pacchetti, chiamati **segments** o **datagrams**, vengono trasmessi uno per volta e possono essere decompatti in **chunk** più piccoli, ogni chunk sarà fornito di un **trasport-layer header**.

## Responsabilità

- ▼ **Consegna da processo a processo**: i messaggi sono consegnati a prescindere da dove questi processi si trovano.

Per garantire ciò vengono utilizzati i socket: data la presenza di diversi socket nell'host ricevente abbiamo bisogno di:

- Un identificatore per ogni socket (porta) allegato ai segmenti ricevuti
- Un processo multiplexer/demultiplexer su entrambi i processi (mittente e destinatario).

#### ▼ Multiplexing

È il processo di creare diversi segmenti da diversi processi, assegnandovi il giusto identificatore

#### ▼ Demultiplexing

Il processo di reindirizzare un segmento al giusto socket in base all'identificatore allegato.

2. **Controllo integrità**: includendo campi per rilevare errori.

3. **Affidabilità del trasferimento**: assicurarsi che i dati vengano consegnati dal mittente al destinatario, correttamente e in ordine.

4. **Congestion/Flow Control**

**UDP** fornisce solo i primi due, gli altri sono forniti solo da **TCP**

## Porte

Gli identificatori dei socket sono detti **porte sorgenti** e **porte di destinazione**.

Una porta è un numero<sub>16</sub> tra 0 e 65535, le porte che vanno da 0 a 1023 sono riservate.

#### ▼ Esempi

Porta	Utilizzo
20	FTP Data Transfer

Porta	Utilizzo
21	FTP Command Control
22	SSH
23	Telnet
25	SMTP
53	DNS
80	HTTP
110	POP3
119	Network News Transfer Protocol
123	Network Time Protocol
143	IMAP
161	SNMP
443	HTTPS

## Multiplexing e Demultiplexing

---

### Multiplexing

Il transport layer del mittente crea un segment che include i dati dell'applicazione, la porta da cui è stato inviato e la porta di destinazione. A questo punto il transport layer passa il segment ottenuto al livello sovrastante (applicazione), che lo incapsula in un IP Datagram.

### Demultiplexing

Se il datagram arriva al destinatario, il livello trasporto lo riceve e lo decapsula, a questo punto il segment viene passato al giusto socket, tramite la porta inclusa nel segment stesso dal mittente.

### In UDP

---

Un socket **UDP** può essere identificato da 2 elementi:

1. IP di destinazione
2. Porta di destinazione

Infatti, si utilizza lo stesso socket per ritornare messaggi a diverse sorgenti.

## In TCP

---

Un socket **TCP** può essere identificato da 4 elementi:

1. IP d'origine
2. Porta di origine
3. IP di destinazione
4. Porta di destinazione

Una delle principali differenze con UDP è che se uno tra client e server interrompe la comunicazione, vengono chiusi i socket di entrambi.

## Perché usare UDP e non TCP?

---

- La connessione avviene più velocemente (non avviene handshake)
- Connectionless
- L'header segment UDP pesa solo 8 byte in più rispetto al segment originale, mentre quello TCP ne pesa 20

## UDP Segment

---

Composto da 5 elementi:

- UDP Header, composto da 4 elementi da 16 bit ciascuno
  1. Porta d'origine
  2. Porta di destinazione
  3. Lunghezza del datagram

4. Checksum: usato dal ricevente per verificare che il messaggio sia intatto
5. Messaggio ricevuto tramite il livello applicazione.

## Checksum

---



Sequenza di bit che, associata al pacchetto trasmesso, viene utilizzata per verificare l'integrità di un dato o di un messaggio che può subire alterazioni durante la trasmissione.

- Il mittente calcola il checksum e lo invia insieme ai dati.
- Il destinatario, una volta ricevuti i dati, ricalcola il checksum sui dati ricevuti e lo confronta con quello inviato.
- Se i checksum coincidono, i dati sono integri; se non coincidono, c'è stato un errore nella trasmissione, e i dati potrebbero essere richiesti nuovamente.

## Affidabilità del trasferimento

---

Tramite checksum abbiamo la possibilità di controllare se un messaggio è corrotto, ma per definire un canale di trasmissione affidabile bisogna garantire:

- Nessuno dei bit trasmessi deve essere corrotto.
- Nessuno dei bit trasmessi deve essere perso o ripetuto.
- Tutti i bit devono essere ricevuti nell'ordine in cui sono stati inviati.

In linea generale dobbiamo assumere che il network layer sia inaffidabile.

## Corruzione dei pacchetti

Supponiamo di avere un canale in cui i bit possono essere solo corrotti (e non persi).

### Approccio stop-and-wait



Si basa sull'invio di un messaggio che attesta il successo o meno della ricezione del pacchetto (ACK oppure NCK), quando il messaggio non viene ricevuto, quest'ultimo deve essere reinviato.

I protocolli basati sulla ritrasmissione dei messaggi sono anche detti **ARQ** (Automatic Repeat reQuest).

Per attuare questo approccio c'è bisogno di aggiungere un nuovo campo all'header dei pacchetti, questo campo specificherà l'ordine in cui i pacchetti dovrebbero essere ricevuti.

## Corruzione o perdita dei pacchetti

Supponiamo ora di trovarci in un canale in cui è possibile anche la perdita di pacchetti.

### Approccio tramite timeout



Un tempo ragionevole per il timeout sarebbe poco più lungo di un del tempo di un Round-Trip.

Aggiungendo un timeout al lato del mittente, abbiamo la possibilità di risolvere sia il problema della perdita del pacchetto stesso, sia il problema della perdita del pacchetto di avvenuta ricezione (ACK).

## Performance

Prendiamo come esempio il caso in cui abbiamo due host che si trovano sui lati opposti degli USA.

Il tempo di un Round-Trip tra questi due host alla velocità della luce è di circa 0.03 secondi, nel caso in cui avessimo:

- un canale con velocità di trasmissione di 1 Gbps ( $10^9$  bitps)
- un pacchetto di 1000 bytes (8000 bit)

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

### Approccio stop-and-wait

Il mittente inizia ad inviare il pacchetto al tempo  $t_0$ , l'ultimo bit entra nel canale a  $t_0 + 0.000008$  secondi, il pacchetto impiega 0.015 secondi per raggiungere il destinatario, supponendo che il pacchetto ACK sia molto piccolo (così da poter trascurare il suo tempo di trasmissione), il pacchetto di ACK ritorna al mittente a

$t = RTT + \frac{L}{R} = 0.030008$  secondi. In questi 0.030008 secondi, il mittente ha aspettato per il 99.973% del tempo.

### Approccio tramite pipeline



Pipelining: al mittente è permesso di inviare più pacchetti senza aspettare conferma di ricezione.

Con questo approccio abbiamo bisogno di:

- Aumentare il range dei sequence numbers dato che ogni pacchetto in transito deve avere un numero unico e potrebbero esserci diversi pacchetti in transito la cui conferma di ricezione (ACK) non è ancora stata ricevuta.

- Abbiamo bisogno di buffers per salvare i pacchetti in entrata, perché non vogliamo avere "buchi" nella trasmissione

Esistono due protocolli base che utilizzano questo approccio:

**Go-Back N, Selective Repeat.**

## Go-Back N

---

Questo protocollo permette al mittente di trasmettere diversi pacchetti senza aspettare di ricevere una conferma di avvenuta ricezione, ma possono esserci al più  $N$  di questi pacchetti.

**Base:** il sequence number del più vecchio pacchetto per cui non è stata ricevuta una conferma di ricezione

**Nextsequm:** il sequence number più piccolo non ancora utilizzato (cioè il prossimo pacchetto che verrà inviato), avremo che:

- I pacchetti da 0 a base – 1 sono quelli trasmessi e la cui ricezione è avvenuta correttamente.
- I pacchetti da base a seqnum – 1 sono quelli inviati ma la cui ricezione non è (ancora) avvenuta.
- I pacchetti da nextseqnum a base +  $N$  – 1 possono essere inviati.
- I pacchetti da base +  $N$  a  $+\infty$  non possono essere usati fino a quando non viene ricevuta una ricezione di consegna dai precedenti.

Lo svantaggio principale è che alcuni pacchetti correttamente ricevuti potrebbero essere buttati via, perché non consegnati nell'ordine previsto, di conseguenza sarà necessario inviare nuovamente il pacchetto.

## Selective Repeat

---

Con questo protocollo il mittente ritrasmette unicamente i pacchetti che sono stati probabilmente persi o sono stati corrotti.

## Buffer TCP

---

Nelle connessioni TCP, le operazioni di invio e recezione fanno affidamento sui **buffer**, questi ci permettono tra le altre cose di compensare differenti velocità di trasmissione tra mittente e destinatario.

## TCP Segment

---

Composto da un **header** e un **data field**, il data field contiene qualche informazione dell'applicazione, l'header invece contiene:

1. **Sequence number**
2. **ACK Number**
3. **Receive window**
4. Lunghezza dell'header
5. **Flags**
6. **Checksum**
  - Il numero di sequenza indica la posizione del primo byte di dati nel segmento all'interno del flusso di dati TCP.
  - Viene utilizzato per il controllo dell'ordinamento dei dati ricevuti e per garantire che i dati vengano consegnati al destinatario nell'ordine corretto
  - Viene utilizzato dal destinatario per confermare l'avvenuta ricezione dei dati e per informare il mittente su quale dato specifico è pronto a ricevere successivamente.
  - Questo numero viene incluso nei segmenti di dati che il destinatario invia al mittente per indicare quali dati sono stati ricevuti correttamente.

## Three-way handshake

---

## **Stabilire una connessione**

---

Più in dettaglio, la procedura è:

1. Il client invia un SYN segment dove:
  - a. SYN=1
  - b. Il sequence number è un numero casuale (che chiameremo `client_isn`)
2. Quando il server riceve il SYN segment, il server alloca i buffer e le variabili TCP, inviando al client un SYNACK segment avente
  - a. SYN=1
  - b. ACK number = `client_isn +1`
  - c. Il sequence number è un numero casuale (`server_isn`)
3. Quando il client riceve il SYN segment, il client alloca i buffer e le variabili, inviando un ACK segment finale, avente:
  - a. (Opzionale) application data
  - b. SYN=0
  - c. ACK number = `server_isn+1`

## **Chiudere una connessione**

---

Partiamo dal presupposto che entrambi (client e server) possono richiedere la chiusura della connessione, in questo esempio supponiamo sia il client a effettuare la richiesta di chiusura.

1. Il client invia un segment al server contenente il bit FIN=1
2. Il server riceve il segment e invia, come risposta (nello stesso segment o in due segment separati):
  - a. ACK=1
  - b. FIN=1

3. Il client riceve una conferma

## Flusso di controllo

Nel momento in cui una connessione TCP riceve pacchetti corretti e nel giusto ordine, mette questi dati all'interno di un buffer, che verranno letti dall'applicazione nel momento del bisogno.

Nel caso in cui l'applicazione sia particolarmente lenta a leggere questi dati, è possibile che il buffer vada in overflow.



Il **TCP flow-control** è un servizio di adattamento della velocità, si occupa di matchare le velocità di trasmissione del mittente con la velocità di lettura dell'applicazione.

## Funzionamento

In questo esempio assumiamo che il destinatario scarti tutti i pacchetti non ordinati.

Dal lato destinatario abbiamo:

- $S_{buff}$ : dimensione del buffer
- $i_{read}$ : l'ultimo byte letto dall'applicazione
- $i_{rec}$ : ultimo byte ricevuto
- $rwnd$ : **receive window**, la cui dimensione sarà pari a



Dal lato del mittente abbiamo:

- $i_{sent}$ : l'ultimo byte inviato
- $i_{ackd}$ : l'ultimo byte per cui è arrivato un pacchetto di acknowledgement dal destinatario
- Nota la dimensione di  $rwnd$ , il mittente garantisce in ogni momento che

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

## Congestione



La congestione si verifica quando il traffico dati che attraversa una rete supera la capacità della rete stessa di gestirlo, causando ritardi e/o perdita di pacchetti.

Supponiamo di avere due host che comunicano tramite un unico router su un singolo link con capacità  $R$ , che il router abbia dei buffer in cui immagazzinare dei pacchetti e che entrambe le applicazioni (lato mittente e lato destinatario) inviano dati alla velocità di  $\lambda_{in}$  bytes/sec, abbiamo due casi:

1. Se  $\lambda_{in} \leq R/2$ , ogni byte sarà ricevuto con un delay finito
2. Se  $\lambda_{in} \geq R/2$ , arriverà un momento in cui il link sarà pieno e i pacchetti extra saranno scartati

Esistono due approcci per gestire questo problema: **end-to-end** e **assistito dalla rete**.

1. **End-to-end**: è lo standard TCP dove la congestione è dedotta dagli host basandosi su packet loss e delay. (TCP si basa principalmente su questo)
2. **Assistito dalla rete**: con questo approccio, livello rete e trasporto lavorano insieme. I router forniscono un feedback esplicito a mittente o destinatario riguardo lo stato di congestione della rete.

## Algoritmo di Jacobson

Si occupa di gestire il tempo di round-trip ed si articola in 3 fasi:

1. **Slow start**: si inizia con una velocità di trasmissione molto bassa, aumentandola esponenzialmente e settando una variabile *thresh* (soglia) ad un valore piuttosto alto.

2. **Additive increase**: la velocità di trasmissione aumenta linearmente
3. **Fast recovery**: (opzionale), nel momento in cui c'è la perdita di un pacchetto, invece di ricominciare a trasmettere a una velocità molto bassa, si va a dimezzare la velocità di trasmissione e la si fa aumentare linearmente di nuovo.

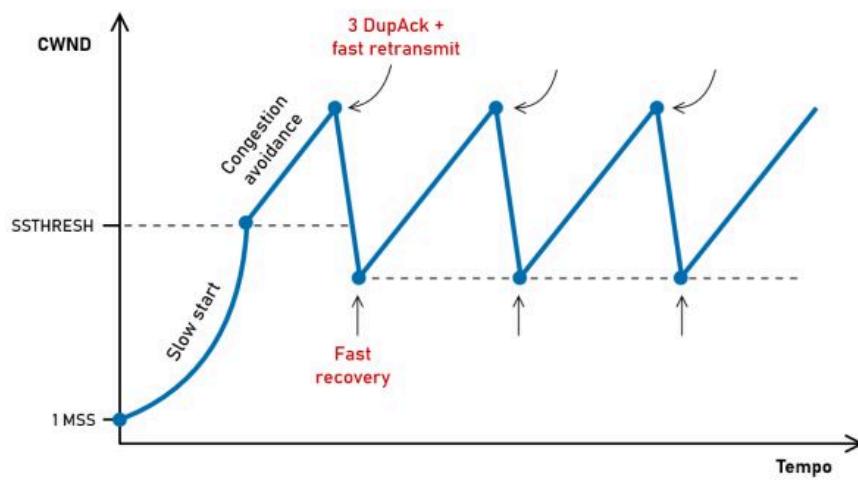


Grafico che mostra l'algoritmo in azione.



# Livello Rete

≡ Lezione 14 → 17

## Livello Rete (Network)



I compiti di questo livello sono:

- Lato mittente:

**incapsulare** ogni segment ricevuto dal livello trasporto in un datagram, immettendoli poi nella rete.

- Lato destinatario: ricevere i datagram dalla rete, **decapsulare** i segments dai datagram e inviarli al livello trasporto sovrastante.

All'interno della rete ci sono dei nodi che inoltrano i datagram ai nodi adiacenti (router), fino a raggiungere la destinazione.

### Best-effort service

Questo livello si occupa principalmente di fornire una **host-to-host delivery**, ma questa **non è garantita**: la rete fa del suo meglio per far arrivare un pacchetto dal mittente al destinatario, però

- non vi è garanzia che i pacchetti arrivino nel giusto ordine, né che effettivamente arrivino
- così come non vi è garanzia che non ci siano delay

# Routers

Il processo di **host-to-host delivery** è eseguito dai router, nodi che hanno diversi link sia in entrata che in uscita, questi forniscono due funzionalità:

- **Forwarding**: quando un pacchetto arriva in input al router, quest'ultimo deve instradarlo sull'output link appropriato, è inoltre possibile **bloccare** o **duplicare determinati pacchetti**, inviandoli su diversi output links.
- **Routing**: decidere il path da seguire per la consegna di un determinato pacchetto, gli algoritmi che si occupano di calcolare questi path sono detti **algoritmi di routing**.

Esistono diversi tipi di router, utilizzati per compiti diversi:

- Router casalinghi.
- **Edge routers**: distribuisce dati tra diverse reti.
- **Core routers**: usati per la distribuzione di dati sulla stessa rete.

**Forwarding**: ci riferiamo all'azione (interna a un router) di **trasferire un pacchetto da un link di input a uno di output**, è una funzionalità implementata in genere lato hardware.

Per **Routing** intendiamo il processo che determina il path da seguire per far sì che un pacchetto raggiunga il nodo di destinazione, ed è generalmente implementato via software.

## Tabelle di routing (1)

Al livello inferiore (Rete) abbiamo delle **tabelle di instradamento (forwarding table)** che specificano a quale link in

uscita deve essere inviato il pacchetto per far sì che raggiunga la propria destinazione, questa decisione viene presa dal router analizzando alcuni valori presenti nell'header dei pacchetti.

Il contenuto di una *forwarding table* deve essere determinato in base alle informazioni ottenute dagli altri router, esistono 2 approcci:

1. **Decentralizzato**: i router comunicano tra loro tramite una componente apposita.
2. **Centralizzato**: abbiamo un controller remoto che si occupa unicamente di effettuare computazioni e fornire le tabelle ad ogni router.

## Componenti di un router (1)

---

- **Porte di input**: si occupano di effettuare **lookup** della tabella di routing, preparare la **switching factory** alla scelta della porta di output e inoltrare i pacchetti contenenti **informazioni di routing** al **routing processor**
- **Switching fabric**: connette le porte di input a quelle di output
- **Porte di output**: ricevono i pacchetti dalla switching fabric e li trasmettono sui link in uscita (e sono generalmente accoppiate alle porte di input)
- **Routing processor**: esegue i protocolli di routing e calcola le tabelle di routing (nell'approccio decentralizzato)

## Tabelle di routing (2)

---

Il loro scopo è di associare indirizzi IP (che a questo livello vengono visti come sequenze composte da 4 byte ( $4 * 8 = 32$  bits)) a porte di output, nel momento in cui un pacchetto viene ricevuto da un link, viene effettuata una *lookup operation* (ogni porta ha una copia della tabella di routing).

È possibile che durante l'operazione di lookup, un IP faccia match con diverse porte di uscita, in questo caso verrà scelta quella che fa match con il maggior numero di bit.

#### ▼ Esempio

Prefisso dell'indirizzo IP	Porta
11001000 00010111 00010 *** * * * * * *	0
11001000 00010111 00011000 * * * * * * *	1
11001000 00010111 00011 *** * * * * * * *	2
altro	3

Prendiamo come esempio l'IP 1100100 00010111 00011000 10011010, questo ip fa match sia con la porta 1 che con la 2, in questo caso sarà scelta la numero 1 perché fa match con più bit rispetto alla due.

Dopo che la porta di output è stata determinata, il pacchetto può essere **invia**to nella switching fabric (o in un buffer dedicato).

Questa procedura di **lookup** e **inoltro** è detta **match-plus-action**, viene inoltre effettuata da: *switches* (nodi), *firewalls*, *NATs* (Network Address Translators).

## Porte

Dato che il processo di *switch* richiede del tempo, ogni porta possiede una coda in cui conserva momentaneamente i pacchetti, è inoltre possibile che diversi pacchetti (provenienti da diverse porte di input), debbano essere re-instradati tutti nella stessa porta di output, c'è quindi bisogno di utilizzare dei meccanismi di scheduling, principalmente 3:

1. **First Come First Served (FCFS)**: l'usuale funzionamento di una coda.

Supponiamo però di trovarci nella situazione in cui il buffer sia pieno, i pacchetti che verranno scartati saranno quelli

arrivati più recentemente, inoltre, i pacchetti verranno rimossi dalla coda solo quando saranno stati trasmessi (in uscita).

## 2. Priority queuing

Ogni classe di priorità avrà una sua coda, la coda non vuota con priorità più alta sarà quella che verrà liberata per prima, generalmente tramite algoritmo **FCFS**.

## 3. Round-Robin: i pacchetti vengono divisi in classi di priorità e ogni classe viene servita a turno.

# Internet Protocol (IP)

---

Attualmente ci sono due versioni in uso:

1. **IPv4**, più usato e comune
2. **IPv6**, più recente, che prima o poi sostituirà Ipv4

La funzionalità principale di questo protocollo è identificare univocamente gli host.

## IPv4 Datagram

---

I campi principali del datagram **IPv4** sono:

1. Versione (4 bits): specifica se v4 o v6
2. Lunghezza dell'header (4 bits)
3. Tipo di servizio (8 bits)
4. Lunghezza del datagram (16 bits): in bytes
5. ID (16 bits)
6. Flags e offset (3+13 bits)
7. "Tempo di vita" (8 bits): per assicurare una circolazione fissata all'interno della rete

8. Protocollo del transport-layer (8 bits)
9. Checksum (16 bits)
- .0. IP di origine e destinazione (32+32 bits)
- .1. Opzioni (non fissato)
- .2. Dati (non fissato)

## Fragmentation

---

Uno dei problemi con i datagram IP è che questi possono venire frammentati al *link-layer*, la quantità massima di dati che può trasmettere un link è detta **Maximum Transport Unit** (MTU), un router che connette 2 link aventi diversi MTU potrebbero ricevere un datagram dal link di input più grande della dimensione massima che può trasmettere il link di output.

Di conseguenza il router dovrà dividere in uno o più pacchetti il datagram, incapsularli e inviarli al link in uscita.

I pacchetti vengono poi riassemblati negli host finali, così da non gravare eccessivamente sulle prestazioni dei router.

## Addressing

---

Ogni indirizzo **IP(v4)** ha 32 bit (4 byte), quindi ci sono un totale di  $2^{32}$  (4 miliardi) possibili indirizzi e c'è bisogno che ogni host ne abbia uno unico.

Gli **IP** sono quindi divisi in due parti:

- la prima parte identifica la sottorete (*subnet*) a cui è connesso il nodo
- la seconda parte identifica la singola interfaccia (il cavo o il collegamento wireless del dispositivo)

Per distinguere la parte della sottorete da quella dell'host, un indirizzo IP è associato a una **subnet mask** che specifica quali bit dell'indirizzo appartengono alla **mask**.

## ▼ Esempio

Supponiamo di avere l'indirizzo **192.168.1.10** e una subnet mask **255.255.255.0**, in binario sono rispettivamente

**Indirizzo IP:** 11000000.10101000.00000001.00001010  
(192.168.1.10)

**Subnet Mask:** 11111111.11111111.11111111.00000000  
(255.255.255.0)

Effettuando l'operazione AND bit-a-bit otteniamo  
11000000.10101000.00000001.00000000  
(Indirizzo di rete: 192.168.1.0)  
L'ultimo byte (8 bit) identifica l'host.

---

È molto importante dividere la rete internet in diverse sottoreti per evitare alcuni problemi (il più banale è il potenziale esaurimento degli IP disponibili). Esistono due approcci per affrontare questo problema:

1. Classful addressing (non più utilizzato)
2. Classless addressing

## **Classful addressing**

---

Gli indirizzi erano divisi in classi, ogni classe aveva un numero fisso di IP disponibili per ogni rete, questo creava un problema di fondo: se a una determinata organizzazione servivano un numero di IP compreso tra una classe e un'altra, tutti i restanti IP rimanevano inutilizzati.

## **Classless addressing (Classless InterDomain Routing [CIDR])**

---

Il classless addressing (CIDR) è un sistema moderno e flessibile per assegnare indirizzi IP, superando le limitazioni del vecchio sistema classful. Utilizzando subnet mask di lunghezza

variabile, CIDR ottimizza l'uso degli indirizzi IP e migliora la gestione delle reti.

## Assegnazione degli IP

Per ogni blocco di indirizzi è necessario assegnare gli IP alle singole interfacce (host) manualmente o automaticamente.

Per l'approccio automatico si utilizza in genere il **Dynamic Host Configuration Protocol (DHCP)**.

### DHCP

Nel momento in cui un host prova a connettersi al server DHCP (in genere gestito dal router stesso o da un host), il server gli assegna un IP, eseguendo questi passaggi:

1. Il nuovo host invia un messaggio sulla rete per trovare il server
2. Il server risponde con un messaggio contenente una possibile configurazione
3. Il client accetta l'offerta, facendo echo del messaggio ricevuto al punto 2.
4. Il server invia un messaggio contenente un ACK.

### Visibilità

Non è necessario che tutti i dispositivi siano visibili globalmente, questo approccio creerebbe ulteriori problemi:

- gli indirizzi sono **finiti**
- rendere visibili globalmente tutti gli host è indesiderabile (e insicuro)

### Network Address Translation (NAT)



È un servizio che permette di rimappare gli indirizzi IP dei pacchetti in indirizzi differenti all'interno di una rete (**network masquerading**).

## IPv6

IPv6 è stato progettato per aumentare il numero di IP disponibili e aggiornare altri aspetti di IPv4:

- IPv4 permetteva la creazione di ~4 miliardi di indirizzi IP ( $2^{32}$ ), con IPv6 hanno 128 bits, di conseguenza non saremo mai più a corto di indirizzi IP ( $2^{128} = 340282366920938463463374607431768211456$ ).
- Alcuni campi dei messaggi IPv4 sono stati eliminati
- È stato deciso di implementare un header di lunghezza fissa (40 bytes)
- I pacchetti che arrivano da determinate applicazioni possono essere raggruppati nello stesso **flow**.

## IPv6 Datagram

Composto dai seguenti campi:

- Versione
- Classe di priorità
- Nome flow
- Lunghezza payload
- Prossimo header
- Hop limit (time-to-live)
- Indirizzi sorgente e destinazione
- Dati

## Routing formalizzato

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

- $N$  è un insieme di nodi (routers)
- $E \subseteq (N \times N)$  rappresenta i lati del grafo

Un lato ha anche un valore, che rappresenta il costo del percorso, possiamo formalizzare questo costo come una funzione  $c : N \times N \rightarrow \mathbb{R}$  tale che, data una coppia di nodi  $(x, y) \in N \times N$ :

- $\forall x \in N \ c(x, x) = 0$
- $(x, y) \notin E \implies c(x, y) = \infty$
- $(x, y) \in E \iff (y, x) \in E$ , e inoltre  $c(x, y) = c(y, x)$

Un path  $\pi \in G$  è una sequenza di nodi  $\pi = (x_1, x_2, \dots, x_n)$  tali che  $\forall x_i, x_{i+1}, (x_i, x_{i+1} \in E)$ , il costo di un path è dato dalla somma dei costi di ogni link  $c(\pi) = \sum_{i=1}^{n-1} c(x_i, x_{i+1})$ .

Chiamiamo  $P(x, y)$  l'insieme di tutti i possibili path tra  $x$  e  $y$ , possiamo definire il percorso (o path) migliore come  $\pi^* \in P(x, y)$  con il minimo costo, cioè  $\forall \pi \in P(x, y), c(\pi^*) \leq c(\pi)$

## Algoritmi per il percorso minimo

Nel caso delle reti di computer non siamo interessati a trovare il miglior percorso, ma quello ottimale per tutte le possibili coppie di nodi. Si può dire che un algoritmo di routing **converge** quando viene trovato il percorso minimo per ogni coppia di nodi.

Esistono due famiglie di algoritmi: **Distance-Vector** e **link-state**.

### Distance-Vector

È un approccio decentralizzato che sfrutta la conoscenza locale della rete combinata con la comunicazione tra i nodi e si basa

sull'algoritmo **Bellman-Ford**. Questo algoritmo è asincrono (nel senso che non richiede che i nodi siano "coordinati").

Chiamiamo  $d^*(x, y)$  la distanza del miglior percorso da  $x$  a  $y$ , possiamo applicare l'**equazione di Bellman**:  $d^*(x, y) = \min\{c(x, v) + d^*(v, y)\}$ , dove  $v$  è un nodo adiacente a  $x$ .

Intuitivamente, se

$v$  è il primo nodo del miglior percorso, allora il resto dei nodi saranno nel miglior percorso tra  $v$  e  $y$ .

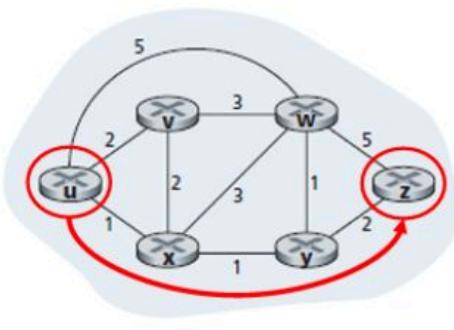
Se riusciamo a trovare questo  $v$  possiamo semplicemente aggiungerlo alla tabella di routing e inoltrarvi tutti i pacchetti diretti a  $y$ .

Utilizziamo la struttura dati  $D_w(v)$  per salvare le distanze di un nodo  $w$  agli altri nodi  $v$ , durante la fase di inizializzazione vengono settate solo le distanze dei nodi adiacenti, nella fase **online** vengono ricevute delle *news* da parte dei nodi adiacenti ed i **distance-vectors**  $D_w$  vengono aggiornati.

#### ▼ Esempio

Consideriamo di voler calcolare il percorso minimo tra un nodo  $u$  ed uno  $z$ :

1.  $u$  inizialmente conosce solo i suoi nodi adiacenti, il costo di  $z$  è  $\infty$
2. Il nodo  $y$  scopre un percorso migliore per  $z$ , che è l'arco  $(y, z) \in E$  con costo 2,  $D_y(z) = c(y, z) = 2$ , a questo punto  $y$  comunica la notizia a  $x$ .
3.  $x$  scopre che il miglior percorso per  $z$  passa per  $y$  con costo 3, cioè  $D_x(z) = c(x, y) + D_y(z) = 1 + 2$ ) e comunica la notizia ad  $u$ .



4.  $u$  riceve questa notizia, adesso esiste un percorso da  $u$  a  $z$  che ha costo minore di  $\infty$ , cioè  $D_u(z) = c(u, x) + D_x(z) = 1 + 3$

Questo processo avviene per ogni percorso/nodo.

#### ▼ Code

```

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) = ? // Initialize Dw(y) appropriately based on
                    // information from neighbors

    send initial Dx(·) to all neighbors

repeat:
    if cost of a neighbor is updated then
        for all nodes y:
            Dx(y) = min_v { c(x, v) + Dv(y) }

    if Dx(y) changed for any destination y then
        send updated Dx(·) to all neighbors

until false // Loop forever

```

#### Pro:

- L'algoritmo è asincrono, quindi la computazione è più semplice

#### Contro:

- Convergenza lenta
- Count-to-infinity: se un nodo o un link fallisce, la

propagazione di questa *news*  
è lenta.

## Algoritmi Link-State

---

Approccio centralizzato che sfrutta la conoscenza completa del grafo per trovare il miglior percorso e si basano sull'algoritmo di Dijkstra.

Utilizziamo una struttura dati  $D(v)$  per salvare la distanza dal nodo corrente a tutte le possibili destinazioni, abbiamo due fasi:

### 1. Fase di inizializzazione

Supponiamo che tutti i costi siano noti ma vengono settate soltanto le distanze dai nodi adiacenti, l'insieme  $N'$  è l'insieme di nodi per cui è già stato scoperto il percorso minimo dalla sorgente su cui viene eseguito l'algoritmo.

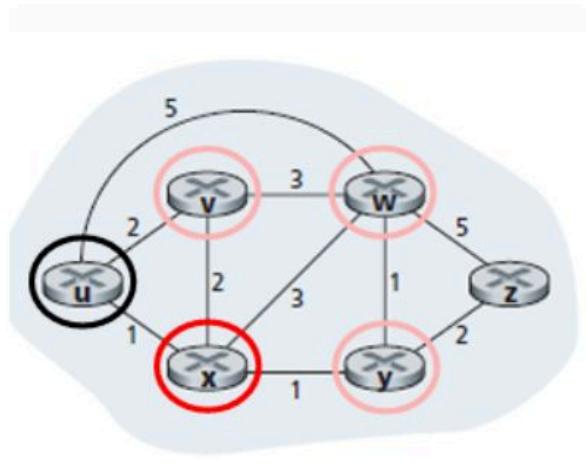
### 2. Fase di costruzione

Si seleziona il nodo  $w$  più vicino, si esplorano tutti i nodi adiacenti a  $w$ , controllando se il path che passa per  $w$  è migliore di quello corrente, l'algoritmo termina quando tutti i nodi sono stati esplorati.

#### ▼ Esempio non completo

Consideriamo ora di voler calcolare i percorsi minimi in un grafo di 6 nodi, facciamo vedere come funziona l'algoritmo quando chiamato sul nodo  $u$ .

Utilizziamo una funzione  $p(x)$  per salvare il nodo predecessore, così che quando vogliamo ottenere un percorso, ci basta scorrere i predecessori fino a quando non arriviamo al nodo corrente (a ritroso).



### Step 1:

$$N' = \{u\}$$

$$D(v) = 2 \quad // \text{ Nodo } v \text{ è un vicino di } u \text{ con costo 2}$$

$$D(w) = 5 \quad // \text{ Nodo } w \text{ è un vicino di } u \text{ con costo 5}$$

$$D(x) = 1 \quad // \text{ Nodo } x \text{ è un vicino di } u \text{ con costo 1}$$

$$D(y) = \infty \quad // \text{ Nodo } y \text{ non è un vicino diretto di } u$$

$$D(z) = \infty \quad // \text{ Nodo } z \text{ non è un vicino diretto di } u$$

### Step 2:

Aggiungiamo  $x$  a  $N'$ , che diventa  $N' = \{u, x\}$ , troviamo ora il nodo  $w$  con  $D(w)$  minimo, cioè  $D(w) = D(x) + c(x, w) = 1 + 3 = 4$ , aggiorniamo ora le distanze minime:

$$D(v) = \min(D(v), D(x) + c(x, v)) = \min(2, 1 + 1) = 2$$

$$D(y) = \min(D(y), D(x) + c(x, y)) = \min(\infty, 1 + 1) = 2$$

$$D(w) = \min(D(w), D(x) + c(x, w)) = \min(5, 1 + 3) = 4$$

$D(z)$  rimane  $\infty$

**Step 3:**

Troviamo il nodo  $w$  non in  $N'$  con  $D(w)$  minimo passante per  $v$ , con  $D(v) = 2$

▼ Code

```
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'

        update D(v) for each neighbor v of w and not in N':
            D(v) = min(D(v), D(w) + c(w, v))

    until N' = N
```

Ogni iterazione controlla ogni nodo della rete tranne la radice, a ogni iterazione viene selezionata una nuova radice, ciò significa che ad ogni iterazione viene rimosso un nodo, quindi il numero di iterazioni è  $|N|(|N| + 1)/2$ , che è un  $O(|N|^2)$  nel caso peggiore

**Pro:**

- Convergenza più rapida

**Contro:**

- Count-to-infinity non possibile: lo stato dei link viene propagato.
- Sincrono, quindi è necessario avere informazioni sull'intera rete prima di iniziare l'invio di pacchetti.



# Livello Link e Fisico

≡ Lezione 18 - 19

## Servizi

Tecnologie utilizzate (Link Layer)

Error Detection and Correction

Parity Check

Multiple bit parity check

Cyclic Redundancy Check (CRC)

## Tipi di link

Collisioni

Multiple Access Protocol

## Protocolli per il partizionamento dei canali

Time Division Multiple Access (TDMA)

Code Division Multiple Access (CDMA)

Random Access Protocols

Slotted ALOHA

Carrier Sense Multiple Access [Collision Detecting] (CSMA) (CSMA/CD)

A turno

Polling Protocol

Token-passing protocol

Media Access Control

Comunicazione

## Switches

Switched LANs

Address Resolution Protocol

## Ethernet Frame

I livelli link e fisico consentono la comunicazione tra due host connessi tra loro e sono i layer presenti in ogni attore di una rete:

- Il livello link si occupa della trasmissione di informazioni da nodo a nodo su un apposito canale di comunicazione.

- Il livello fisico si occupa della struttura dei link e come sono rappresentati i bit.

Per trasmettere un datagram da un host all'altro abbiamo bisogno di farlo passare attraverso diversi link/dispositivi, questi datagram vengono incapsulati all'interno di link-layer frames, che variano in base al tipo di link utilizzato, questi frames vengono convertiti in

**segnali**, che possono essere:

- impulsi elettrici
- luce
- onde radio

## Servizi



Il livello link offre fino a 4 diversi servizi: framing, accesso ai link, affidabilità (per la consegna), rilevazione e correzione di errori.

1. **Framing**: si occupa di incapsulare i datagram in dei frame del layer corrente, contenenti, nel datagram di livello superiore i dati necessari all'utente e degli header/tailer specifici.
2. **Link Access**: definisce le norme che regolano l'accesso al link tramite un protocollo detto **Medium Access Protocol (MAC)**, queste regole dipendono dall'architettura del link:
  - a. Per i **link punto a punto**, questo protocollo è basilare: il mittente può inviare un frame ogni qualvolta che il link non sta venendo utilizzato.
  - b. Per i **link broadcast** potrebbe verificarsi il problema dell'accesso concorrenziale, in questo caso il **MAC** deve

occuparsi di coordinare la trasmissione dei frame da parte di diversi nodi.

3. ***Reliable Delivery***: garantisce che i frames trasmessi attraverso i link vengano ricevuti (in genere questa funzionalità non è implementata perché può produrre un grande sovraccarico e in genere i protocolli di livello superiore sono abbastanza affidabili)
4. ***Error detection and correction***: l'hardware a questo livello potrebbe essere affetto dal problema del ribaltamento dei bit.

## Tecnologie utilizzate (Link Layer)

---

I protocolli a questo livello dipendono dalle tecnologie utilizzate dal tipo di link, le più comuni sono:

- ***Ethernet 802.3***: connessione via cavo basata su impulsi elettrici inviati tramite cavi in rame a 4 coppie intrecciate oppure fotoni attraverso cavi in fibra ottica.
- ***WiFi 802.11***: connessione wireless basata su radiofrequenze (2.4GHz, 5GHz, 6GHz)
- ***Bluetooth***: connessione wireless basata su radiofrequenze a 2.4GHz

## Error Detection and Correction

---

A questo livello è possibile effettuare rilevamento e correzione degli errori a livello dei bit.

Supponiamo di avere dei dati  $D$  di dimensione  $d$ , nel momento in cui inviamo il messaggio possiamo includere dei bit ***EDC*** (***Error Detection Control***). L'obiettivo è di controllare se i dati ricevuti  $D'$  e  $EDC'$  sono o meno uguali a quelli iniziali

## Parity Check

---

Probabilmente la forma di **error detection** più semplice, inseriamo nel messaggio un bit aggiuntivo (che chiameremo parity bit), in modo che il numero totale di bit settati a 1 tra i  $d+1$  bit del messaggio sia pari o dispari, in base allo schema scelto. Il destinatario dovrà solo controllare che il numero di bit settati a 1 rispetti lo schema, è possibile però che qualche errore non venga rilevato.

## Multiple bit parity check

---

Una possibile implementazione prevede l'uso di parity bits multipli, utilizzando ad esempio una matrice di  $n$  righe ed  $m$  colonne, ognuna delle quali avente uno specifico parity bit, in questo modo il destinatario può rilevare gli errori e potenzialmente tentare una correzione, questo metodo può inoltre rilevare qualsiasi combinazione di due errori per ogni pacchetto.

## Cyclic Redundancy Check (CRC)

---

Supponiamo di avere dei dati  $D$  di dimensione  $d$ , per far sì che questi dati vengano inviati, il mittente e il destinatario devono concordare un pattern di  $r+1$  bit chiamato **Generator** ( $G$ ), avente il bit più significativo settato a 1.

Per ogni pezzo di

$D$ , il mittente sceglierà  $r$  bit extra (detti **CRC** bits) da aggiungere al messaggio, tali che  $D + CRC \equiv_2 G$ .

In caso questa cosa sia vera, il messaggio è corretto, altrimenti no.

Quest'operazione è implementata facendo uno shift di  $G$  fino al bit più significativo del messaggio e eseguendo un'operazione XOR bit-a-bit

## Tipi di link

---

Esistono due tipi diversi di link:

- **Punto a punto**: prevedono un unico mittente da un capo del link ed un unico destinatario dall'altro capo del cavo (ad esempio due computer connessi tramite un cavo ethernet)
- **Broadcast**: ci sono diversi nodi che inviano e ricevono dati, tutti connessi allo stesso canale.

## Collisioni

---

Uno dei principali problemi dei link di tipo **broadcast** sono le collisioni, se ci sono diversi nodi che vogliono trasmettere dati sullo stesso canale, questi dati potrebbero sovrapporsi, diventando illeggibili. Questi frame corrotti sono poi ricevuti da tutti i nodi e scartati come errori.

## Multiple Access Protocol

---

Questi protocolli vengono utilizzati per gestire le trasmissioni via canali **broadcast** in modo che:

- le collisioni siano gestite
- ogni nodo abbia la possibilità di trasmettere
- le connessioni già stabilite non vengono interrotte

Il ruolo primario di questi protocolli è proprio quello di evitare le collisioni, gli approcci principali sono:

- Partizionamento del canale (più in particolare della banda)
- Accesso casuale
- Accesso a turno

---

Sarebbe ottimale avere un protocollo per un canale broadcast di  $R$  bps che possa:

- **Massimizzare l'utilizzo del canale di comunicazione**, se ci fossero  $M$  nodi che necessitano di inviare dati, ognuno di

questi dovrebbe avere, in media, un *throughput* di  $R/M$  bps

- Essere **decentralizzato**, un nodo **master** potrebbe essere un *single point of failure*
- Essere **semplice e leggero**, in modo da **limitare le possibilità di sovraccarico**

## Protocolli per il partizionamento dei canali

---

### Time Division Multiple Access (TDMA)

---

Questo protocollo divide il tempo in *time frames*, divide poi ogni time frame in  $N$  *time slots*, dove  $N$  è il numero di nodi.

La dimensione di questi slot è decisa in modo tale che ogni time slot sia assegnato ad uno dei nodi.

Tipicamente la dimensione degli slot viene presa in modo da **permettere la trasmissione di un intero pacchetto durante un time slot**.

### Code Division Multiple Access (CDMA)

---

Viene assegnato un codice ad ogni nodo, questo codice viene usato per codificare e decodificare i bit di dati inviati dal nodo.

Se i codici sono assegnati correttamente, questo protocollo permette la trasmissione

### Random Access Protocols

---

I canali che adottano questo protocollo permettono a un nodo di trasmettere alla velocità massima del canale ( $R$ ).

Se avviene una **collisione**, tutti i nodi aspettano un tempo casuale prima di ritentare una connessione, dato che questa selezione è effettuata in maniera indipendente, è probabile che due nodi selezionino un tempo abbastanza diverso per permettere ad uno dei due riesca a trasmettere l'intero messaggio, in caso contrario il processo viene reiterato fino a quando non si verifica la condizione.

## Slotted ALOHA

---

Il funzionamento è il seguente:

- il tempo viene diviso in **slots**, ogni slot è grande abbastanza da contenere un frame
- i nodi sono sincronizzati, ogni nodo trasmette frames solo nel momento iniziale dello slot
- se non viene rilevata alcuna collisione, la comunicazione continua, altrimenti ogni nodo ha una probabilità  $p \in [0,1]$  di inviare nuovamente il frame, fino a quando il frame non viene inviato correttamente

### Pro:

- Se c'è un solo nodo, questo userà l'intera banda del canale
- È decentralizzato, i nodi sono totalmente indipendenti tra poco, sincronizzazione a parte
- Semplice da implementare ed eseguire
- Avere diverse collisioni consecutive è poco probabile

### Contro:

- È possibile che venga tentata una trasmissione quando il canale è già in uso, generando una collisione

Per risolvere il contro è possibile monitorare il canale e provare a trasmettere solo quando il canale è inattivo.

## **Carrier Sense Multiple Access [Collision Detecting] (CSMA) (CSMA/CD)**

---

Questi protocolli, **Carrier Sense Multiple Access** e **Carrier Sense Multiple Access Collision Detecting** si basano su due ulteriori principi:

- **Carrier sensing**: ogni nodo deve ascoltare il canale prima di iniziare la trasmissione
- **Collision detecting**: ogni nodo deve ascoltare il canale anche durante la trasmissione, e se viene rilevata una collisione, termina la trasmissione e aspetta un tempo randomico prima di ricominciare.

Nonostante questi accorgimenti, è comunque possibile che si verifichino delle collisioni, questo a causa del ritardo di trasmissione:

- nonostante la velocità di propagazione dei segnali sia prossima alla velocità della luce, ci vuole un po' di tempo prima che l'informazione venga ricevuta da tutti i nodi
- durante questo tempo i nodi potrebbero considerare libero un canale attualmente in uso

**CSMA**: molto semplice

- controlla che il canale sia libero
- se il canale è libero, invia un frame

Le collisioni sono ovviamente possibili, ma non vengono rilevate, se non nel momento in cui non

**CSMA/CD**: più evoluto

- controlla che il canale sia libero
- se il canale è libero, invia un frame
- durante la trasmissione, controlla che non avvengano collisioni

viene ricevuto un messaggio ACK.

- se avviene una collisione, metti in pausa la trasmissione e aspetta un tempo  $k \in \{0, \dots, 2^{n-1}\}$ , dove  $n$  è il numero di collisioni rilevate

## A turno

---

### Polling Protocol

---

C'è un nodo **master** che seleziona tramite Round-Robin un nodo alla volta a cui è permesso trasmettere

#### Pro:

- non ci sono collisioni

#### Contro:

- Delay causato dal polling
- Approccio centralizzato, single point-of-failure

### Token-passing protocol

---

Nessun nodo **master**, i nodi si scambiano un frame chiamato **token**, quando un nodo possiede il token, a quel nodo è permessa la trasmissione.

#### Pro:

- Nessuna collisione
- Approccio decentralizzato

#### Contro:

- monopolizzazione del canale (nel caso in cui un nodo dimentichi di rilasciare il token)

## Media Access Control

---

A questo livello, i diversi dispositivi sono identificati da un indirizzo **MAC**, ogni interfaccia ne possiede uno, ed ogni

produttore ne ha uno proprio.

È un indirizzo composto da 6 byte, in genere rappresentato in esadecimale.

## Comunicazione

---

In una rete **LAN**, la comunicazione avviene in questo modo:

- Un interfaccia *A* include l'indirizzo MAC di un'interfaccia *B* all'interno del frame da inviare e lo immette nel canale di comunicazione
- *B* riceve il frame e confronta il proprio indirizzo MAC con quello di destinazione all'interno del frame
- Se i due indirizzi sono uguali, allora il frame viene accettato, altrimenti viene rifiutato.

## Switches

---

Gli switch sono l'equivalente del link layer dei router, però:

- non implementano alcun algoritmo di routing
- vengono utilizzati solo indirizzi MAC, gli IP non vengono considerati

Il ruolo di uno switch è di ricevere frames e inoltrarli ai giusti link in uscita, uno switch è trasparente per gli host e i router nella sottorete, presenta inoltre dei buffer.

Presentano inoltre delle “tabelle di routing” (forwarding tables) che associano indirizzi MAC alle interfacce.

## Switched LANs

---

È tipico per le reti LAN l'utilizzo di uno o più switch che connettono diversi dispositivi in locale, a differenza dei router, gli switch sono **più veloci e plug-and-play**.

Sono però **limitate in dimensioni** e devono avere una **struttura gerarchica**.

## Address Resolution Protocol

---

Dato che i protocolli dei livelli sovrastanti sono lavorano tramite indirizzi IP, abbiamo bisogno di tradurre questi IP in indirizzi MAC.

L'**Address Resolution Protocol** gestisce queste conversioni:

- ogni interfaccia è dotata di un modulo avente una tabella ARP che associa ad ogni IP nella LAN con un indirizzo MAC con uno specifico **time-to-live**.
- Dato che gli indirizzi MAC sono locali, anche questo protocollo funziona solo per connessioni locali.

Supponiamo che un host *C* (222.222.222.220) voglia inviare un messaggio ad un host *A* (222.222.222.222), per fare ciò abbiamo bisogno di conoscere il MAC associato ad *A*.

Prima di inviare il messaggio, se

*A* non è presente nella tabella ARP di *C*, viene inviato un pacchetto ARP in broadcast a tutti i dispositivi, tutti i nodi ricevono questo pacchetto, ma soltanto il dispositivo con IP 222.222.222.222 risponderà a questo messaggio.

Se un IP si trova esternamente alla rete (cioè non è locale) invece, il router che connette le due reti deve avere almeno due interfacce (2 IP, 2 MAC e 2 tabelle ARP), i frame diretti esternamente alla rete vengono inviati alla prima interfaccia del router, inoltrati alla seconda e diretti verso il giusto host utilizzando la seconda tabella ARP.

## Ethernet Frame

---

Composto dai seguenti campi:

- Data field: contenente l'IP datagram

- Indirizzo di destinazione (MAC)
- Indirizzo del mittente (MAC)
- Type field: specifica il protocollo livello rete usato per il frame
- CRC number
- Preamble: “wake up block” usato per sincronizzare domande e risposte



# Sicurezza di rete

≡ Lezione 21

Sicurezza informatica

Malwares

Denial of Service (DoS)

Packet Sniffing

IP Spoofing

Basi di sicurezza informatica

Crittografia

Chiavi

Crittografia simmetrica

Key exchange

Crittografia asimmetrica

Certification Authority

Message Integrity

## Sicurezza informatica

È la branca che si occupa di attacchi informatici e possibili soluzioni per prevenirli.

### Malwares



Un malware è un software maligno che può essere trasferito da un computer a un altro attraverso una rete (download di file, allegato ad una mail, ecc.)

Possono essere divisi in ulteriori sottocategorie:

- **Adware**: forzano il computer a mostrare annunci pubblicitari indesiderati
- **Scareware**: mostrano falsi messaggi d'allarme per indurre gli utenti a scaricare (ulteriori) malware.
- **Wiper**: cancellano i file senza il consenso dell'utente.
- **Ransomware**: bloccano i file chiedendo spesso un riscatto per sbloccarli.
- **Spyware**: rubano i dati sensibili presenti sulla macchina.
  - **Keylogger**: software che registra e salva ogni tasto premuto sulla tastiera.
- **Rootkit**: ottiene i privilegi di root del sistema operativo.
- **Zombie o botnet**: rende il computer infettato uno *slave* o un punto d'appoggio per infettare altri computer.

I malware attuali sono spesso **auto-replicanti**, una volta che infettano un host, provano ad infiltrarsi su altri host in giro per internet, diffondendosi a velocità esponenziale. Possiamo dividerli in **virus** e **worm**:

- i **virus** sono malware che hanno bisogno di qualche tipo di interazione da parte dell'utente per infettare un host, possono diffondersi, ad esempio, tramite allegati a mail, inviando allegati simili a tutti i contatti del primo host infettato.
- i **worms** sono più pericolosi perché possono infettare dispositivi senza una diretta interazione dell'utente, ad esempio diffondendosi in una rete che accetta il worm senza intervenire direttamente.

## Denial of Service (DoS)

---



Sono attacchi abbastanza comuni, mirano a rendere inutilizzabile una specifica infrastruttura (un host, una rete, dei server, DNS, ecc.).

Ne esistono di 3 tipi:

1. **Vulnerability attack**: inviano specifici messaggi a applicazioni o sistemi operativi vulnerabili costringendoli a "stopparsi".
2. **Bandwidth flooding**: consistono nell'inviare ingenti quantità di pacchetti a un determinato host, in modo tale da non rendere possibile ai pacchetti "previsti" di raggiungere il server.
3. **Connection flooding**: stabilire un gran numero di connessioni TCP all'host bersaglio, così da non rendergli possibile di stabilirne di nuove.

## Packet Sniffing



Prevede che un *ricevitore passivo (sniffer)* crei delle copie di pacchetti inviati da un host in modo tale da rubare informazioni sensibili (eavesdropping)

Questi sniffer possono essere distribuiti in tutti i tipi di reti broadcast, semplicemente copiando i pacchetti destinati altrove invece di scartarli.

Sono molto difficili da individuare perché non creano traffico aggiuntivo sulla rete (sono per questo detti passivi).

## IP Spoofing



È una tecnica che consente a host malintenzionati di inviare pacchetti con l'IP dei mittenti "falsato".

Possono sfruttare le vulnerabilità delle applicazioni per attaccare specifici host mascherando la propria identità dietro quella di un diverso utente.

Può essere utilizzato anche per attacchi *man-in-the-middle* in cui l'utente malintenzionato si trova tra due host che stanno comunicando, mascherandosi prima da uno e poi dall'altro.

## Basi di sicurezza informatica

Visti i diversi tipi di attacco, è ora possibile definire delle proprietà che una **connessione sicura** dovrebbe garantire:

1. **Confidenzialità**: solo il mittente e il destinatario dovrebbero essere in grado di leggere il contenuto dei messaggi trasmessi.
2. **Integrità dei messaggi**: il contenuto della comunicazione **non** deve essere alterato. Né da utenti malintenzionati, né per errori.
3. **Autenticazione end-point**: sia il mittente che il destinatario devono essere in grado di confermare l'identità dell'altra parte coinvolta nella comunicazione.
4. **Operational security**: fare affidamento su un'infrastruttura di rete che previene l'intrusione di utenti malintenzionati sulla rete.

Le prime 3 proprietà sono *software-based*, l'ultima si basa su hardware specifico (ad esempio un firewall).

## Crittografia

Una **tecnica crittografica** permette a un mittente di rendere incomprensibili i dati per un intruso, allo stesso tempo, il destinatario deve essere in grado di recuperare i dati originali da quelli resi incomprensibili.

Il messaggio nella forma iniziale è detto **plaintext** ed è **leggibile per chiunque**, prima di essere inviato l'host utilizza un **algoritmo crittografico** così da renderlo illeggibile (questo messaggio ora sarà **ciphertext**) e dovrà essere **decriptato** all'arrivo al destinatario.

## Chiavi

---

Nella maggior parte dei sistemi di crittografia moderni, la tecnica crittografica è ben nota, la parte sconosciuta è la parte di *encrypt* e *decrypt* (le chiavi).

Queste chiavi sono delle stringhe alfanumeriche che devono essere fornite agli algoritmi di *encrypt/decrypt* per criptare o decriptare i messaggi. Possono essere identiche (crittografia simmetrica) o diverse (crittografia asimmetrica).

## Crittografia simmetrica

---

Esiste un'unica chiave utilizzata sia per criptare che decriptare i messaggi.

- **Encryption:** il *plaintext* e la chiave vengono passati **all'algoritmo di criptazione**, il *ciphertext* viene quindi inviato
- **Decryption:** il *ciphertext* e la chiave vengono passati **all'algoritmo di decrypt** per ricreare il *plaintext*

## Key exchange

---

Il problema della **crittografia simmetrica** è che la chiave deve necessariamente essere comunicata al destinatario, di

conseguenza è **necessario** utilizzare **un canale sicuro** per la comunicazione o utilizzare dei protocolli che permettono loro di convergere su una chiave condivisa.

## Crittografia asimmetrica

---

Vi sono due diverse chiavi, una per criptare, una per decriptare, in genere **quella per criptare è pubblica**, mentre **quella per decriptare è privata**

## Certification Authority

---

Nel caso di una **chiave crittografica pubblica** sarebbe utile **verificare** se una chiave pubblica **appartenga davvero all'entità con cui si desidera comunicare**, altrimenti potremmo trovarci in possesso della chiave di un utente malintenzionato, in grado di decriptare i nostri dati sensibili.

L'associazione di una chiave pubblica a una particolare entità viene tipicamente effettuata da un'Autorità di Certificazione (Certification Authority)

Una **CA** verifica che un'entità sia chi dice di essere. Non esiste un protocollo per questo, **ci si deve fidare** del fatto che la CA abbia effettuato una verifica dell'identità sufficientemente rigorosa.

Una volta che la CA verifica l'identità dell'entità, la CA crea un certificato che associa la chiave pubblica dell'entità all'identità. Il certificato contiene la chiave pubblica e un identificatore univoco globale del proprietario (ad esempio, un nome o un indirizzo IP).

## Message Integrity

---

Si tratta di controllare che il messaggio non sia stato manomesso e che sia stato inviato dall'host previsto, possiamo

creare un *check-item* simile al **checksum** o il **CRC**, in genere viene utilizzata una **funzione hash**.

Una funzione hash crittografica è una funzione  $h$  che converte un messaggio  $x$  in una stringa di lunghezza fissa  $h(x)$  per cui trovare un messaggio  $y$  tale che  $h(y) = h(x)$  sia computazionalmente impraticabile.

Come abbiamo fatto per checksum o CRC, possiamo allegare questo *hash* al messaggio:

1. L'Host  $A$  crea il messaggio  $m$  e calcola l'hash  $H = h(m)$ .
2.  $A$  inserisce  $H$  in coda al messaggio  $m$ , creando un messaggio esteso  $(m, h)$  e inviandolo a  $B$
3. L'Host  $B$  riceve il messaggio  $(m, h)$  e calcola  $h(m)$ , se  $h(m) = H$ , allora il messaggio è intatto.

Questo approccio è chiaramente difettoso, per migliorare questo esempio possiamo far sì che  $A$  e  $B$  abbiano una **chiave o password segreta**, conosciuta solo da loro due, sia questa chiave  $s$ .

Supponendo che esista tale  $s$ , allora:

1. L'Host  $A$  crea un messaggio  $m + s$  (come concatenazione del messaggio e del segreto) e calcola l'hash  $H = h(m + s)$ , noto anche come **Message Authentication Code (MAC)**.
2. L'Host  $A$  aggiunge in coda il **MAC** al messaggio  $m$ , creando un messaggio esteso  $(m, h(m + s))$ , e invia il tutto a  $B$ .
3. L'Host  $B$  riceve il messaggio esteso  $(m, H)$  e, conoscendo  $s$ , calcola il MAC  $h(m + s)$ . Se  $h(m + s) = H$ , il messaggio è intatto.



# End-point Authentication

≡ Lezione 22

## End-point Authentication



È il processo che permette ad un'entità di dimostrare la propria identità ad un'altra entità attraverso una rete.

Un **protocollo di autenticazione** viene in genere eseguito **prima dell'inizio della comunicazione** tra due attori per stabilire che entrambi siano chi dicono di essere.

Supponiamo che un host *A* debba autenticarsi a *B*, se vi è già stata una comunicazione in precedenza, *B* potrebbe accertarsi dell'identità di *A* controllando l'IP all'interno del datagram, ma questa modalità di autenticazione può funzionare soltanto se supponiamo che un utente malintenzionato non possa creare un datagram con un IP fasullo (tramite spoofing).

Possiamo utilizzare i **router** per evitare questa eventualità, **configurandoli in modo che inoltrino soltanto dei datagram legittimi**, contenenti IP che arrivano effettivamente dagli host che li hanno inviati.

Un altro approccio prevede l'utilizzo di una **password**, che può essere utilizzata, oltre che per l'autenticazione, anche per controlli di integrità. Qui sorge un nuovo problema, l'intruso potrebbe ottenere la password in diversi modi:

- ottenendo la password criptata (playback attack)

- ottenendo la password decriptata
- in entrambi i casi, l'autenticazione sarebbe compromessa.

## Nonce



Il protocollo TCP, per effettuare il three-way-handshake, utilizza delle sequenze di numeri casuali per evitare che possibili ri-trasmissioni vengano interpretate come segnali SYN/ACK, possiamo utilizzare un'idea simile per l'autenticazione.

In questo caso utilizzeremo un **nonce**, un numero casuale o pseudo-casuale che il protocollo utilizzerà una sola volta nella sua vita.

### ▼ Esempio di funzionamento

1. Un host *A* invia un messaggio "io sono A" all'host *B*
2. L'host *B* sceglie un **nonce** e lo invia ad *A*
3. *A* crittografa il nonce e la password, utilizzando e invia il tutto a *B*
4. *B* decripta il messaggio, se il nonce decriptato è lo stesso che è stato inviato in precedenza da *B* ad *A*, *A* sarà autenticato.

## Secure Socket Layer (SSL)

Abbiamo visto che vengono utilizzate diverse tecniche crittografiche a per garantire l'autenticazione point-to-point per connessioni TCP.

Queste tecniche vengono implementate tramite il **Secure Socket Layer**, una versione migliorata di **TCP** che presenta diverse feature di sicurezza.

SSL prevede tre fasi principali: **handshake**, **key derivation**, **data transfer**.

## Handshake



Se un client  $B$  vuole utilizzare SSL per comunicare con un server  $A$ , durante questa fase il client deve:

- stabilire una connessione TCP con  $A$
- verificare che  $A$  sia effettivamente  $A$  (e non un server falso)
- Inviare ad  $A$  una chiave segreta

### ▼ Esempio di funzionamento

1. Non appena la connessione è stata stabilita,  $B$  invia un *hello message*.
2. Il server risponde inviando un certificato, che contiene la propria chiave pubblica
3. Il client può fidarsi di  $A$  perché il certificato è stato emesso da una **Certification Authority**
4.  $B$  a questo punto genera un **nonce** che verrà utilizzato solamente per questa sessione, criptandolo con la chiave pubblica ricevuta da  $A$
5. Il server  $A$  decripta il messaggio utilizzando la propria chiave pubblica.
6. La connessione tra  $A$  e  $B$  è ora sicura.

## Key derivation



La chiave ottenuta da  $A$  tramite  $B$  può essere utilizzata per lo scambio di dati privati e per controlli di integrità.

In ogni caso, sarebbe più sicuro l'utilizzo di diverse chiavi, possono esserne create quattro:

- $E_{B-to-A}$ : chiave utilizzata per i dati scambiati da  $B$  ad  $A$  per questa sessione
- $M_{B-to-A}$ : chiave **MAC** (Message Authentication Code) per i dati scambiati da  $B$  ad  $A$
- $E_{A-to-B}$ : chiave utilizzata per i dati scambiati da  $A$  a  $B$  per questa sessione
- $M_{A-to-B}$ : chiave **MAC** (Message Authentication Code) per i dati scambiati da  $A$  a  $B$

## Data transfer

**SSL** mette in sicurezza **TCP** andando a dividere i dati in record, per ogni record:

- SSL aggiunge in coda un **Message Authentication Code** per effettuare controlli di integrità
- Il record modificato viene criptato
- Il record criptato viene passato a **TCP** per la trasmissione

## Firewall



Un firewall è una combinazione hardware-software utilizzata per isolare una rete da pacchetti malevoli, rifiutandoli.

Tutto il traffico dovrebbe passare attraverso questo firewall.

Ci sono tre approcci principali: **Traditional packet filtering**, **Stateful filtering**, **Application gateway**

### **Traditional packet filtering**

Un packet filter è in genere implementato sul gateway (cioè il router che connette la rete locale all'ISP).

Si occupa di esaminare ogni datagram, determinando se un datagram debba o meno essere accettato, in base alle regole scelte dall'amministratore.

L'amministratore configura il firewall in base alle policy dell'organizzazione, una possibile policy può essere basata su una combinazione di IP e porte.

Queste regole sono implementate tramite liste di controllo di accesso, il problema principale di questo approccio è il fatto che sia stateless.

### **Stateful filtering**

Questi filtri risolvono il problema dell'approccio tradizionale andando a tracciare tutte le connessioni TCP in una tabella, così da capire se il traffico proviene da connessioni legittime.

Il firewall riconosce e tiene traccia sia del momento in cui viene stabilita una connessione, sia quando questa viene terminata, in più, possiamo assumere che una connessione sia terminata se non viene effettuata alcuna azione tramite essa per un determinato periodo di tempo.

## **Application gateway**

---

Potrebbe essere utile fornire o rifiutare determinate funzionalità in base all'utente o l'applicazione che effettua la richiesta, in questo caso possiamo servirci di un gateway implementato a livello applicazione, questa funzionalità viene in genere implementato come un server a parte che lavora in combinazione con il firewall.

Se un utente vuole utilizzare questo application gateway, avrà bisogno di autenticarsi, il server controllerà se questo utente ha effettivamente i permessi per l'utilizzo di questo protocollo, quindi tutte le richieste e le risposte passeranno attraverso questo gateway.

Sorgono però diversi svantaggi:

- serve un gateway diverso per ogni applicazione
- problemi di performance
- i software sugli host devono essere a conoscenza dell'esistenza di questo gateway

## **Intrusion Detection and Prevention Systems**

---



Sono un gruppo di dispositivi specializzati che monitorano la rete, cercando pacchetti sospetti, riconoscendoli e bloccandoli.

Il sistema è in genere composto da:

- Uno o più sensori IDS
- Un singolo processore che colleziona e integra le informazioni, lanciando l'allarme quando qualcosa non va.

Gli IDS possono essere:

- Signature-based: hanno un database di firme di attacchi.

- Anomaly-based: crea un profilo di traffico e controlla per streams che sono statisticamente non usuali.

Nelle reti particolarmente estese c'è il problema di avere diversi livelli di sicurezza per diversi dispositivi: è possibile istituire una "zona demilitarizzata" in cui la sicurezza è bassa e dove le restrizioni sono limitate. In genere queste zone sono messe tra due firewall, uno esterno, meno restrittivo ed uno interno, più restrittivo.



# XML, YAML, JSON

≡ Lezione 23

Network programming

Come funziona un socket TCP (Java - C++)

Scambio di dati

XML

YAML (Yaml Ain't Markup Language)

Liste

Dizionari

JSON

Sintassi

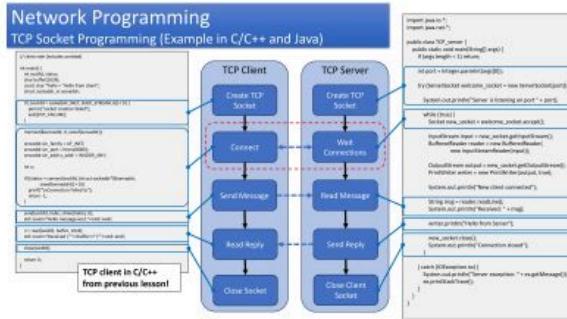
Esempi

## Network programming

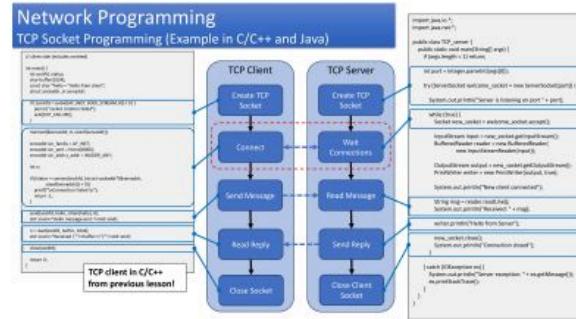


Per *network programming* si intende la creazione di applicazioni che funzionano su dispositivi diversi, potenzialmente in simultanea, tramite una rete. L'infrastruttura su cui si basano queste applicazioni è standardizzata e la maggior parte dei processi non devono essere gestiti, se ne occupano i socket (black box).

## Come funziona un socket TCP (Java - C++)



Creazione di un socket TCP con client in C e server Java



Creazione di un socket TCP con client in C e server Java

In questo esempio vediamo due programmi scambiarsi delle stringhe, ma in generale le applicazioni si scambiano delle strutture dati, in generale, i formati più comuni per lo scambio di informazioni sono XML, YAML e JSON.

## Scambio di dati

### XML



**e**xensible **M**arkup **L**anguage è un linguaggio meta-markup per documenti e dati testuali, leggibile da computer e umani.

```

<xml>
    <article>
        <author>Gerhard Weikum</author>
        <title>The Web in 10 Years</title>
    </article>
</xml>

```



### Possibili vantaggi dell'utilizzo di XML

- Facile da leggere per umani
- Molto espressivo
- Flessibile, personalizzabile
- Molto utilizzato e supportato

```
<person born="1912-06-23" died="1954-06-07"> Alan Turing</person>
```

## YAML (Yaml Ain't Markup Language)



YAML è un linguaggio creato appositamente per essere minimale e facilmente leggibile dagli esseri umani.

A differenza di XML, i contenuti YAML sono definiti tramite indentazione (NB: non si usa un'indentazione TAB-based, ma solo tramite *whitespaces*).

In YAML i nomi e i valori sono separati da i due punti (colon) (`:`).

### Liste

Una lista può essere definita in due modi:

- Una struttura di elementi preceduti da trattini

```
people:  
  - person:  
    personID: 77  
    firstName: John  
    lastName: Doe
```

```
- person:  
    personID: 78  
    firstName: Alice  
    lastName: Doe
```

- Una sequenza di elementi separati da virgole all'interno di parentesi quadre

```
names: [John, Alice, Bob]
```

## Dizionari

Un dizionario è rappresentato da coppie di **elementi - valori** separati da virgole all'interno di parentesi graffe:

```
person: {personid: 77, firstName: John, lastName: Doe}
```

## JSON

Formato utilizzato per lo scambio di dati, facilmente leggibile dagli umani, facilmente computabile e basato sulle convenzioni del linguaggio C. È considerato il linguaggio per lo scambio di dati ideale (ed è uno dei più usati).

### Sintassi

I nomi dei campi sono definiti tra virgolette (perché sono delle stringhe), seguiti da due punti (:) e il valore.

NB: l'indentazione non è necessaria, ma utile per visualizzare meglio i dati.

È possibile creare dei *JSON objects* come segue:

```
{"Name1": Value1, "Name2": Value2, ...}
```

Oppure degli array:

```
[{"Name1": Value1, "Name2": Value2, ...]
```

I valori sono definiti con una sintassi simile a quella del C:

- Stringhe: `"name": "Bob"` (all'interno di virgolette)
- Numeri (int, float, double): `"age: 27"`, `"weight":60.5"`
- Array: `"pets": ["cat", "dog"]`, `"siblings": []`
- Valori booleani: `"isAlive": true`
- Valori nulli o non disponibili: `"phoneNumber": null`

## Esempi

- Definizione di una persona come visto precedentemente per [YAML](#).

```
{  
    "person": {  
        "personID": 77,  
        "firstName": "John",  
        "lastName": "Doe"  
    }  
}
```

- Definizione di una lista di persone utilizzando un array

```
{  
    "people": [  
        "person":{  
            "personID": 77,  
            "firstName": "John",  
            "lastName": "Doe"  
        },  
        "person":{  
            "personID": 78,  
            "firstName": "Jane",  
            "lastName": "Doe"  
        }  
    ]  
}
```

```
        "personID": 78,  
        "firstName": "Alice",  
        "lastName": "Doe"  
    }  
]  
}
```

È possibile semplificare la sintassi rimuovendo il field `person`:

```
{  
  "people": [  
    {  
      "personID": 77,  
      "firstName": "John",  
      "lastName": "Doe"  
    },  
    {  
      "personID": 78,  
      "firstName": "Alice",  
      "lastName": "Doe"  
    }  
  ]  
}
```



# REST

≡ Lezione 24

## Network programming

È possibile scegliere tra due diverse alternative per la progettazione di applicazioni che usano protocolli UTP o TCP:

1. **Standard protocol**
2. **Proprietary protocol (custom)**

L'approccio più comune consiste nel progettare applicazioni **on top** del protocollo standard HTTP, questo perché esso è **versatile** ed **esistono già delle API utilizzabili**.

## REST



**REST** (REpresentational State Transfer) è un **insieme di principi architetturali** che guidano i programmatore nella **definizione di applicazioni web flessibili, scalabili e modulari**, queste applicazioni vengono dette **RESTful APIs**. Una

**REST API** è una sorta di **interfaccia tra diversi client e delle risorse condivise** (e.g. un database).

## Elementi

---

- **Risorse:** le **informazioni che gli host manipoleranno**, identificate da **URI**(Uniform Resource Identifiers), possiamo vedere questi URI come una generalizzazione di URL che contengono anche il nome della risorsa.
- **Rappresentazioni:** Come le **risorse sono strutturate e rappresentate** in uno specifico formato (e.g. testo, XML, YAML, JSON, etc.)

## Principi

---

1. **Client-Server:** nelle applicazioni **REST** i due host sono indipendenti:
  - I **client conoscono** unicamente gli **URI** delle risorse richieste.
  - I **server** non fanno altro che **inviare le risorse** tramite HTTP.
2. **Statelessness:** ogni richiesta deve contenere tutte le informazioni necessarie per essere compresa dal server, anziché dipendere dal fatto che il server ricordi richieste precedenti.
3. **Cacheability:** le risorse dovrebbero essere memorizzabili nella cache lato client o server.  
Le risposte dovrebbero anche contenere informazioni su se la memorizzazione nella cache è consentita o meno per una risorsa specifica.
4. **Uniform interface:** tutte le richieste API per le stessa risorsa dovrebbero essere simili, a prescindere da dove provenga la richiesta.
5. **Layered system architecture:** il messaggio può essere inoltrato a diversi intermediari, ma questo non dovrebbe essere "percepito" né dal client né dal server.

6. **Code on demand** (opzionale): le risposte possono anche contenere del codice eseguibile, in questo caso questo codice dovrebbe esser eseguibile solo su richiesta.

La maggior parte di questi principi somigliano a delle feature del protocollo *HTTP*, in un certo senso, REST definisce come usare HTTP in una applicazione, infatti:

- **il client accede alle risorse tramite richieste HTTP**
- **il server fornisce queste risorse tramite risposte HTTP.**

## HTTP

---

### Richieste

Un messaggio di richiesta include i seguenti campi:

- **method**: specifica il comando che il server deve eseguire.
- **URL**: serve per identificare l'object su cui vogliamo operare.
- **version**: specifica la versione di HTTP (e.g. HTTP/1.1))
- **header lines**: contiene i parametri della richiesta, il numero e il tipo di queste linee non sono fissi, ogni linea include il nome e il valore del parametro
- **body**: contiene dati potenzialmente associati al comando

### Risposte

---

Simili a quelli di richiesta, contengono:

- **version**: cioé la versione HTTP della risposta del server.
- **status code**: un codice che specifica il risultato della richiesta
- **phrase**: il risultato della richiesta

### In pratica

---

## Richieste

---

Le risorse sono identificate dal field **URL**, le operazioni fatte sulle risorse sono specificate nel campo **method**, esse sono:

- **GET**: richiedi una risorsa.
- **POST**: modifica una risorsa.
- **PUT**: crea una nuova risorsa.
- **DELETE**: elimina una risorsa.

## Risposte

---

Le informazioni sulle richieste sono specificate dallo **status code** e da **phrase**.

- **200**: risorsa gestita con successo.
- **201**: risorsa creata con successo.
- **404**: risorsa non trovata.
- **405**: metodo non supportato.
- etc.

## Rappresentazione delle risorse

---

Un **URL HTTP** viene usato per identificare le risorse, formati comuni sono:

- [http(s)]://[Domain name of REST API][:Port]/[API version]/[Path to resource]
- [http(s)]://[Domain name][:Port]/[REST API]/[API version]/[Path to resource]

Linee guida per la denominazione e l'uso delle risorse:

- Usa sostantivi plurali, non verbi:
  - Good: /users

- Bad: /doSomethingOnUsers
- Usa metodi HTTP invece di termini CRUD (o simili) nei nomi:
  - Good: /users (with GET method)
  - Bad: /getAllUsers
- Identifica risorse uniche con ID:
  - Good: /users/1
  - Bad: /users?id=1
- Puoi usare query per selezionare o ordinare le risorse:
  - Good: /users?nickname=Bob&sort=age
  - Bad: /usersNamedBobSortedByAge



# Domande + risposte orale

HTTP

TCP (bit Syn ack)

HTTP STATELESS

PROXY

SERVIZI DI TRASPORTO

Applicazioni CLIENT SERVER

Tabelle NAT

INDIRIZZI IP (Internet Protocol)

SERVER DNS

(gethostbyname) + algoritmo a fine slide per velocizzare il processo?

UDP

ALGORITMO DI JACOBSON

TCP differenze con UDP (cosa serve in più: socket)

SICUREZZA DELLA RETE

TCP HANDSHAKE

Come si realizza un server in c (firme principali)

MAIL PROTOCOL

Funzionamento

Protocolli aggiuntivi

POP3 - Post Office Protocol 3

IMAP - Internet Mail Access Protocol

HTTP

HTTP E TCP (perche si utilizza invece di UDP)

COSA RESTITUISCE LA CONNESSIONE CON SOCKET

APP CON RICHIESTE UDP DIFFERENZA CON TCP (con tcp possiamo avere più socket che comunicano sulla stessa porta, con udp credo di no)

PORTE PER LA COMUNICAZIONE: QUANTE E QUALI SONO

APPLICAZIONI REST: FORMATO DELLE RICHIESTE E RISPOSTE

COMPOSIZIONE DELL'URL

USO DEL DNS PER CONVERTIRE L'HOSTNAME IN IP

HTTP CON TCP E DNS CON UDP

DIFFERENZA TRE UDP E TCP: affidabilità, flusso dei dati e controllo della congestione

DHCP CON UTILIZZO DI UDP

ROUTER: COS'E E A COSA SERVE

INDIRIZZI IP MAPPATI NELLE TABELLE LOCALI DI ROUTING

ROUTER FORWARDING TABLE (tabelle di indirizzamento)

ALGORITMI DI ROUTING

Distance Vector

Link State

SWITCH E MAC ADDRESS

PROBLEMA DELLE COLLISIONI E CONGESTIONE DI RETE + ALGORITMI DI RISOLUZIONE

Address Resolution Protocol - ARP

PROTOCOLLO IMAP

PORTE DI COMUNICAZIONE

PROBLEMA DEGLI IP CON IPv4

FRAMMENTAZIONE CON IP ATTRAVERSO I ROUTER

CONNESSIONE PERSISTENTE/ NON PERSISTENTE (+USO DI RICHIESTE SINCRONE IN PIPELINE)

SEQUENCE NUMBER

ATTACCO PLAYBACK

## HTTP

HyperText Transfer Protocol è un protocollo di trasmissione che si trova al livello applicazione che permette lo scambio di file ipertestuali, utilizza TCP come protocollo sottostante per garantire l'affidabilità delle informazioni trasmesse.

Possiamo parlare di richieste e risposte HTTP:

Una richiesta HTTP prevede: un metodo, l'URL a cui vogliamo effettuare la richiesta, la versione di http, eventuali parametri da includere come header, e un body.

I metodi per le richieste HTTP sono: GET, PUT, HEAD, POST, DELETE.

- GET: richiedere una risorsa alla pagina

- HEAD: richiedere l'header di una risorsa alla pagina (ma senza ricevere la risorsa)
  - PUT: crea una risorsa alla pagina indicata dall'URL
  - POST: modifica una risorsa
  - DELETE: cancella una risorsa
- 

Le risposte HTTP invece prevedono: uno status code, la versione di HTTP, e una stringa contenente il risultato della richiesta, i codici sono:

- 100-199 INFO
  - 200-299 OK
  - 300-399 REDIRECT
  - 400-499 CLIENT ERROR
  - 500-599 SERVER ERROR
- 

HTTPS è invece un protocollo che permette una comunicazione sicura, sta per HyperText Transfer Protocol over Secure socket layer, fornisce come requisiti chiave:

1. un'autenticazione del sito web visitato;
2. protezione della privacy
3. integrità dei dati scambiati tra le parti comunicanti

## TCP (**bit Syn ack**)

---

TCP è un protocollo del livello trasporto, insieme al protocollo UDP sono lo standard per la comunicazione tramite rete.

Un socket **TCP** può essere identificato da 4 elementi:

1. IP d'origine
2. Porta di origine
3. IP di destinazione

#### 4. Porta di destinazione

---

Una delle principali differenze con UDP è che se uno tra client e server interrompe la comunicazione, vengono chiusi i socket di entrambi.

---

Una connessione TCP tra due host viene stabilita tramite un three-way handshake:

- L'host A invia all'host B un segment contenente il bit SYN=1 e un sequence number generato casualmente
- L'host B riceve il messaggio e ne invia un altro all'host A, contenente i bit SYN=1 e ACK=1, avente come sequence number il numero precedente+1
- L'host A riceve il messaggio e invia un ulteriore messaggio con il bit SYN=0 e ACK=1 e può già iniziare a trasmettere dati

## HTTP STATELESS

---

HTTP è un protocollo stateless, questo significa che non conserva informazioni sulle interazioni passate, ad esempio, se venisse effettuata due volte di fila la stessa richiesta, la seconda non sarebbe considerata ridondante e verrebbe comunque inviata una risposta. Il contrario di stateless è stateful.

## PROXY

---

Per proxy si intende un server che fa da intermediario tra un client e altri server, in pratica riceve richieste da parte del client, le inoltra ad un server (o un altro client) e poi ritorna una risposta, nascondendo, tra le altre cose, l'identità del client che ha fatto la richiesta.

# SERVIZI DI TRASPORTO

---

Il livello di trasporto offre dei protocolli che garantiscono:

- **Affidabilità** nel trasporto dei dati.
- **Throughput**: il volume a cui il processo mittente invia i dati al processo destinatario.
- **Timing**: ogni bit inviato arriva al socket ricevente entro un certo tempo.
- **Sicurezza**: **encryption** e **decryption** dei dati.

# Applicazioni CLIENT SERVER

---

Un applicazione client server è un'applicazione che si basa, appunto, sull'architettura client-server.

In un'architettura client server abbiamo due tipi di host:

- **Server**: un host sempre online che si occupa di fornire servizi al client.
- **Client**: host che accede ai servizi forniti dal server, i client non comunicano tra loro.

Il server ha un *indirizzo IP* fissato, in modo da rendere possibile per i client di inviare richieste in qualsiasi momento.

In linea generale, vengono utilizzati diversi server in simultanea per gestire le diverse richieste dei client, nonostante i client siano ignari dell'esistenza di diversi server.

La maggior parte dei servizi online sono basati su un'architettura client server: instagram, youtube, facebook, segrepass, etc.

## Tabelle NAT



Network Address Translation è un servizio (livello rete) che permette di rimappare gli indirizzi IP dei pacchetti in indirizzi differenti all'interno di una rete (**network masquerading**).

In breve, questo servizio si occupa di ricevere richieste da parte dei client e di instradarle sulla rete internet, modificando l'IP sorgente del client; in particolare, nel momento in cui inviamo una richiesta tramite rete internet, il nostro dispositivo invia una richiesta al router a cui siamo collegati, il router cambia l'IP all'interno del segment, inserendo l'IP pubblico del router.

## INDIRIZZI IP (Internet Protocol)

Per parlare di indirizzi IP iniziamo introducendo le due versioni attualmente in utilizzo, IPv4 e IPv6:

- IPv4 utilizza indirizzi a 32 bit, che sono quindi relativamente pochi (4 miliardi,  $2^{32}$ ), considerando la diffusione esponenziale che ha avuto internet
- IPv6 utilizza indirizzi a 128 bit,  $2^{128}$  possibili combinazioni
- La prima parte di ogni IP identifica la subnet a cui appartiene, la seconda parte indica l'host.

Il **classful addressing** è il metodo utilizzato agli albori per l'assegnazione degli IP alle organizzazioni, vi erano delle classi predefinite, ogni classe aveva un numero fisso di IP. Il problema nasceva nel momento in cui un'organizzazione aveva bisogno di un numero di IP leggermente maggiore rispetto a

quelli forniti da una classe, perché ricadeva nella classe successiva e rimaneva inutilizzato un numero molto alto di IP.

---

Il **classless addressing** (CIDR) è un sistema moderno e flessibile per assegnare indirizzi IP, superando le limitazioni del vecchio sistema classful. Utilizzando subnet mask di lunghezza variabile, CIDR ottimizza l'uso degli indirizzi IP e migliora la gestione delle reti.

---

Per l'approccio automatico di assegnazione degli IP si utilizza in genere il protocollo **Dynamic Host Configuration Protocol (DHCP)**.

- Nel momento in cui un host prova a connettersi al server DHCP (in genere gestito dal router stesso o da un host), il server gli assegna un IP:
  1. Il nuovo host invia un messaggio broadcast sulla rete per trovare il server
  2. Il server risponde con un messaggio contenente una possibile configurazione
  3. Il client accetta l'offerta, facendo echo del messaggio ricevuto al punto 2.
  4. Il server invia un messaggio contenente un ACK.

## SERVER DNS

**(gethostbyname) + algoritmo a fine slide per velocizzare il processo?**

---

In generale, per gli esseri umani è più semplice ricordare delle sequenze di parole piuttosto che delle sequenze di numeri, il Domain Name System è un sistema distribuito e gerarchico che si

utilizza per tradurre gli URL dei siti web in indirizzi IP, il funzionamento è il seguente:

- L'host A effettua una richiesta per un determinato sito al DNS resolver
- Il DNS resolver si occupa di interrogare la gerarchia di DNS (root, top level, authoritative DNS)
- Appena il DNS resolver ha ottenuto l'indirizzo IP del sito richiesto, lo ritorna all'host, che può raggiungerlo.

`gethostbyname` è una funzione in C deprecata, al suo posto viene utilizzata la funzione `getaddrinfo`.

La funzione `getaddrinfo` fornisce una conversione indipendente dal protocollo da un nome host ANSI a un indirizzo.

## UDP

---

UDP è un protocollo **connectionless** del livello trasporto che si occupa di:

- consegna dei messaggi tra processi
- controllo dell'integrità dei messaggi

Viene utilizzato principalmente per applicazioni che non hanno bisogno di affidabilità nella trasmissione dei dati (streaming, videogiochi, servizi real-time), dove la latenza è più importante dell'occasionale perdita di dati.

Non presenta un sistema di controllo di congestione o del flusso dei dati, né garantisce la consegna dei pacchetti.

## ALGORITMO DI JACOBSON

---

Si occupa di gestire il tempo di round-trip ed si articola in 3 fasi:

- Slow start:** si inizia con una velocità di trasmissione molto bassa, aumentandola esponenzialmente e settando una variabile `thresh` (soglia) ad un valore piuttosto alto.
- Additive increase:** la velocità di trasmissione aumenta linearmente
- Fast recovery:** (opzionale), nel momento in cui c'è la perdita di un pacchetto, invece di ricominciare a trasmettere a una velocità molto bassa, si va a dimezzare la velocità di trasmissione e la si fa aumentare linearmente di nuovo.

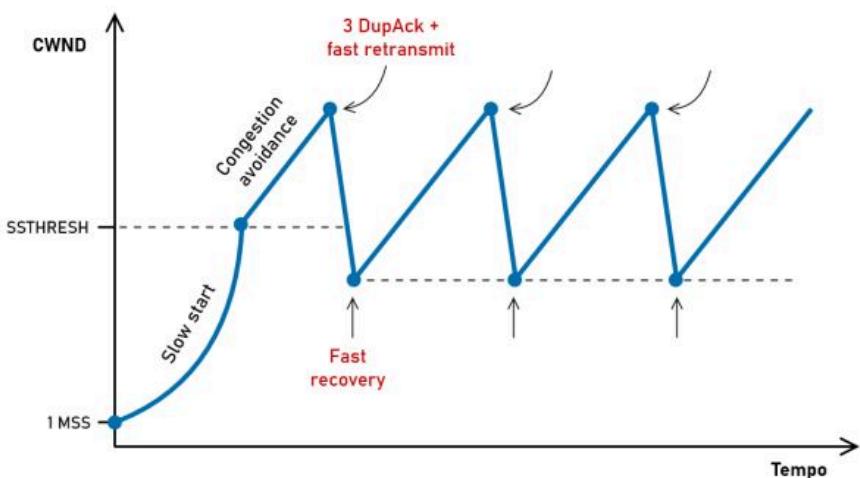


Grafico che mostra l'algoritmo in azione.

## TCP differenze con UDP (cosa serve in più: socket)

Le differenze principali tra TCP e UDP sono:

- TCP è più connection oriented, effettua il three-way handshake prima di iniziare la trasmissione di dati.
- UDP è connectionless, non è previsto il three-way-handshake

- TCP si assicura che i pacchetti vengano consegnati tutti e nel giusto ordine, ritrasmettendo quelli che non arrivano al destinatario
  - Effettua controlli sul flusso di dati e sulla congestione della rete
  - Risulta essere più lento
- 
- UDP invia i pacchetti, ma non assicura che questi vengano effettivamente recapitati, ancor meno che arrivino nel giusto ordine
- 
- Non effettua alcun tipo di controllo, né di congestione, né di flusso di dati
- 
- Compensa la scarsa affidabilità con una maggiore velocità

## SICUREZZA DELLA RETE

Un connessione sicura dovrebbe fornire le seguenti proprietà:

- **Confidenzialità:** il contenuto dei messaggi dovrebbe essere leggibile solo da mittente e destinatario
- **Integrità dei messaggi:** i messaggi non devono essere alterati mentre vanno da un host all'altro, né da altri utenti, né per errori
- **Autenticazione end-point:** mittente e destinatario devono essere in grado di confermare l'identità dell'altro durante la comunicazione
- **Sicurezza operativa:** l'infrastruttura su cui si basa la comunicazione deve prevenire l'intrusione di utenti malintenzionati sulla rete

## TCP HANDSHAKE

1. Il client A invia un bit SYN=1 accompagnato da un `client_number`
2. Il client B riceve il bit SYN e invia al client A un segment contenente un bit **SYNACK** con ack number = `client_number+1`
3. Quando il client A riceve questo bit **SYNACK**, invia un ultimo segment ACK contenente dati, SYN=1

## Come si realizza un server in c (firme principali)

1. Creazione del Socket: `socket()`
2. Binding: `bind()`
3. Listen: `listen()`
4. Accettazione delle Connessioni: `accept()`
5. Invio e Ricezione dei Dati: `send()` e `recv()`
6. Chiusura del Socket: `close()`

## MAIL PROTOCOL



Il Simple Mail Transfer Protocol è il protocollo su cui si basa l'invio e la ricezione delle e-mail.

Due elementi coinvolti:

1. **User-agent** (applicazione che permette la gestione delle mail).
2. **Mail server** (che contiene le mailbox degli utenti).

## Funzionamento

Gli utenti non si scambiano direttamente mail ma si appoggiano sui mail server.

In SMTP ci sono un lato client e un lato server in esecuzione sui mail server.

▼ Esempio di comunicazione tra due utenti

Supponiamo di avere due utenti A e B e che A voglia inviare una mail a B.

1. A, utilizzando il proprio user-agent, scrive la mail e la invia
2. L'user-agent di A si occupa di inviare la mail al mail server, inserendolo in una coda.
3. Il lato client di SMTP, che è in esecuzione sul mail server di A, apre una connessione con il mail server (lato server) di B.
4. Dopo l'handshake, la mail viene inviata tramite una connessione TCP.
5. Il mail server (lato server) di B riceve il messaggio.
6. B può usare il proprio user-agent per leggere il messaggio.

## Protocolli aggiuntivi

---

Questi protocolli sono necessari per accedere alle mail

### POP3 - Post Office Protocol 3

---

L'user-agent apre una connessione TCP col mail server, dopodiché abbiamo 3 fasi:

1. **Authorization:** l'user-agent autentica l'utente inviando username e password al server.
2. **Transaction:** è possibile effettuare azioni basilari sui messaggi.

3. **Update:** la sessione termina e le azioni basilari effettuate al punto 2. vengono eseguite e sovrascritte.

## **IMAP - Internet Mail Access Protocol**

---

Questo protocollo fornisce ulteriori funzionalità come creare cartelle, effettuare ricerche e permette a più client di connettersi allo stesso server

## **HTTP**

---

In questo caso l'user-agent è un browser, mentre i mail server continuano a utilizzare lo standard SMTP.

## **HTTP E TCP (perche si utilizza invece di UDP)**

---

HTTP è il protocollo standard che si utilizza per lo scambio di informazioni sul web, possiamo dividere HTTP in richieste e risposte.

Una richiesta HTTP è composta da: un *metodo*, la versione di http in utilizzo, dei parametri (header) e un body.

Una risposta http è composta da: **codice**, versione, phrase

I *metodi* possono essere:

- GET: richiedi una risorsa
- HEAD: richiedi solo l'header di una risorsa
- PUT: crea una risorsa
- POST: modifica una risorsa
- DELETE: elimina una risorsa

I **codici** possibili sono:

- 100-199 info
- 200-299 OK
- 300-399 redirect
- 400-499 client error
- 500-599 server error

Il protocollo sottostante utilizzato è TCP perché garantisce affidabilità e corretto ordine dei dati inviati, controllo di congestione e di flusso, è inoltre connection oriented.

## COSA RESTITUISCE LA CONNESSIONE CON SOCKET

---

**APP CON RICHIESTE UDP DIFFERENZA CON TCP (con tcp possiamo avere più socket che comunicano sulla stessa porta, con udp credo di no)**

---

## PORTE PER LA COMUNICAZIONE: QUANTE E QUALI SONO

---

Le porte, sono numeri che identificano punti specifici di connessione e comunicazione su un dispositivo di rete. Ogni porta consente di distinguere diversi servizi e applicazioni che funzionano contemporaneamente su un singolo indirizzo IP

Le porte possibili sono 65.535, le prime 1023 sono riservate.

Porta	Utilizzo
20	FTP Data Transfer
22	Secure SHell
23	TERminal NETwork
25	Standard Mail Transfer Protocol

Porta	Utilizzo
53	DNS
80	HTTP
110	POP3
143	Internet Message Access Protocol
443	HTTPS

## APPLICAZIONI REST: FORMATO DELLE RICHIESTE E RISPOSTE

Le applicazioni REST utilizzano in genere come protocollo quello HTTP, di conseguenza le richieste e le risposte HTTP per comunicare.

Una REST API funziona come un'interfaccia tra i client e le risorse presenti su un server. Gli elementi architetturali principali prevedono:

- **risorse**: i pezzi di informazioni che gli host manipolano, identificati da URIs (Uniform Resource Identifiers)
- **Rappresentazioni**: come le risorse sono strutturate e in che formato (plaintext, XML, YAML, JSON)

I principi di una REST API sono 6:

1. **Architettura Client-Server**
2. **Statelessness**: non devono essere salvate informazioni sulle richieste effettuate in precedenza e due richieste uguali effettuate di fila non devono essere considerate ridondanti
3. **Cacheability**: le risorse devono essere “cacheable”, quindi deve essere possibile salvarle all'interno di una cache, lato client o lato server
4. **Uniform interface**: le risorse, a prescindere dalla richiesta effettuata, devono avere un aspetto uniforme

5. Architettura a layer: queste API devono considerare il fatto che il messaggio debba passare attraverso diversi intermediari, ma questo non deve essere percepito né dal client né dal server
6. Codice su richiesta: deve essere possibile eseguire codice su richiesta, perché alcune richieste potrebbero prevedere di effettuare delle computazioni prima di dare risposta.

Per le richieste abbiamo bisogno di: metodo, uri, header, e body contenente i dati della richiesta.

I *metodi* possono essere:

- GET: richiedi una risorsa
- HEAD: richiedi solo l'header di una risorsa
- PUT: crea una risorsa
- POST: modifica una risorsa
- DELETE: elimina una risorsa

L'uri è l'identificatore della risorsa, ad esempio /api/users/123

Gli header contengono dei metadati, tipo il formato della richiesta, i formati di risposta accettati, auth, etc.

Le risposte invece contengono: status code, header, body.

I **codici** possibili sono:

- 100-199 info
- 200-299 OK
- 300-399 redirect
- 400-499 client error
- 500-599 server error

Gli headers possono contenere il tipo di formato della risposta, il body contiene dei potenziali dati richiesti.

## COMPOSIZIONE DELL'URL

Un URL è formato nel seguente modo:

[protocollo]://[userinfo][hostname]:[port]/[path]?[query]#[fragment]

## USO DEL DNS PER CONVERTIRE L'HOSTNAME IN IP

---

Il Domain Name System viene utilizzato per convertire un URL (più facile da ricordare per gli esseri umani) in un indirizzo IP, in particolare, quando digitiamo un URL all'interno della casella di ricerca di un browser e clicchiamo invio:

- il DNS resolver prende la nostra richiesta e inizia a risalire la gerarchia dei server DNS per trovare l'IP dell'URL richiesto (top level DNS, authoritative DNS, root DNS)
- Nel momento in cui il DNS resolver ottiene l'IP richiesto, lo ritorna al client

## HTTP CON TCP E DNS CON UDP

---

HTTP utilizza il protocollo TCP perché più affidabile, per quanto lento, DNS utilizza UDP perché necessita di maggiore velocità, perché la traduzione di un URL in IP può causare dei ritardi non banali.

## DIFFERENZA TRE UDP E TCP: affidabilità, flusso dei dati e controllo della congestione

---

TCP è connection oriented, la connessione viene stabilita prima che inizi la comunicazione tra gli host, in particolare avviene un three-way-handshake, non previsto per UDP, questo three way handshake avviene in tre fasi:

- L'host A invia all'host B un segmento contenente il bit SYN=1 e un sequence number generato casualmente
- L'host B riceve il messaggio e ne invia un altro all'host A, contenente il bit SYN=1 e ACK=1, avendo come sequence number il numero precedente+1
- L'host A riceve il messaggio e invia un ulteriore messaggio con il bit SYN=0 e ACK=1 e può già iniziare a trasmettere dati

## DHCP CON UTILIZZO DI UDP

---

Il Dynamic Host Configuration Protocol è un protocollo del livello rete che permette a un host connesso a un router di ricevere automaticamente un indirizzo IP effettuando i seguenti passaggi:

1. L'host A si connette al router e invia un messaggio DHCPDISCOVER in broadcast, quindi all'indirizzo 255.255.255.255
2. Il (o i) server DHCP all'interno della rete possono rispondere ("privatamente") a questo messaggio, inviando un messaggio DHCPOFFER, contenente l'IP assegnato all'host
3. L'host A può accettare la configurazione inviata dal DHCP server
4. Il server DHCP invia un messaggio di ACK

## ROUTER: COS'E E A COSA SERVE

---

Il router è l'interfaccia che si occupa di tradurre le richieste inviate dalla rete locale alla rete internet globale, in particolare è composto da:

- porte di input e di output (in genere accoppiate)

- switching factory
- routing processor

Le porte di input e output contengono un buffer, nel caso in cui il link di collegamento in uscita sia saturo; nel caso in cui anche il buffer sia pieno, i pacchetti già consegnati verranno scartati, seguiti da quelli che non riescono a "entrare" in coda.

Il routing processor è l'elemento che si occupa di decidere la giusta porta di output per ogni pacchetto (nell'approccio decentralizzato). Nell'approccio centralizzato si occupa di questo un host specifico, che calcola le tabelle di routing e le invia ai singoli router.

Gli algoritmi di routing si dividono in due categorie:

- Algoritmi distance-vector
  - decentralizzati (non c'è bisogno di conoscere l'intera struttura della rete per iniziare la consegna dei pacchetti)
  - la propagazione delle bad news è lenta
- Algoritmi link state
  - centralizzati (ogni router conosce completamente il resto della rete)
  - i router si scambiano tutte le informazioni che hanno sul resto della rete
  - il router calcola i percorsi minimi verso i diversi router, creando così la propria tabella di routing

## INDIRIZZI IP MAPPATI NELLE TABELLE LOCALI DI ROUTING

---

# **ROUTER FORWARDING TABLE (tabelle di indirizzamento)**

---

Le tabelle di indirizzamento sono delle tabelle interne al router che decidono, in base alla destinazione di ogni pacchetto, a quale porta inviare quest'ultimo.

Esistono 2 approcci per la determinazione di queste tabelle:

- Centralizzato: esiste un host specifico che si occupa unicamente di calcolare le tabelle di routing di ogni router in modo da diminuire il lavoro al carico dei router stessi
- Decentralizzato: ogni router possiede un componente chiamato routing processor che si occupa di determinare il miglior percorso in base alle informazioni ricevute dagli altri router

## **ALGORITMI DI ROUTING**

---

Esistono due famiglie di algoritmi di routing:

1. Distance Vector
2. Link state

### **Distance Vector**

Gli algoritmi distance vector utilizzano la struttura dati  $D_w(v)$  per salvare le distanze di un nodo  $w$  agli altri nodi  $v$ .

### **Link State**

Gli algoritmi link-state sono degli algoritmi centralizzati (che quindi hanno bisogno di una conoscenza completa della rete per funzionare) e si basano sull'algoritmo di Dijkstra.

Questi algoritmi sono asincroni e decentralizzati, quindi non hanno bisogno di una conoscenza completa della rete per iniziare a funzionare.

Il funzionamento prevede che inizialmente ogni nodo salvi i dati sui nodi adiacenti ad esso, settando gli altri a  $\infty$ , aggiornando poi le distanze mano a mano che vengono ricevute news da parte dei nodi adiacenti

Utilizziamo una struttura dati  $D(v)$  per salvare la distanza dal nodo corrente a tutte le possibili destinazioni.

Il funzionamento prevede che inizialmente siano note tutte le distanze dai nodi, settando però solo quelle dei nodi adiacenti, nella fase di costruzione si seleziona il nodo più vicino a quello iniziale e si esplorano i nodi adiacenti a quello attuale, aggiornando le distanze se se ne trovano di minori rispetto a quelle già settate.

## SWITCH E MAC ADDRESS

Gli switch sono l'equivalente del link layer dei router, però:

- non implementano alcun algoritmo di routing
- vengono utilizzati solo indirizzi MAC, gli IP non vengono considerati

Il ruolo di uno switch è di ricevere frames e inoltrarli ai giusti link in uscita, uno switch è trasparente per gli host e i router nella sottorete, presenta inoltre dei buffer.

Un MAC Address è un indirizzo composto da 6 byte che identifica un host all'interno di una sottorete, ogni byte è rappresentato da un numero in esadecimale.

# **PROBLEMA DELLE COLLISIONI E CONGESTIONE DI RETE + ALGORITMI DI RISOLUZIONE**



La congestione si verifica quando il traffico dati che attraversa una rete supera la capacità della rete stessa di gestirlo, causando ritardi e/o perdita di pacchetti.

Per risolvere i problemi di congestione è possibile utilizzare algoritmi come quello di Jacobson, il funzionamento è diviso in 3 parti:

1. Slow start: si inizia la trasmissione a una velocità bassa, aumentandola esponenzialmente
2. Additive increase: la velocità viene aumentata in maniera lineare
3. Fast recovery: nel momento in cui avviene una perdita di dati, invece di ricominciare la trasmissione da zero, si dimezza la velocità attuale e si ricomincia aumentandola linearmente

## **Address Resolution Protocol - ARP**

È un protocollo del livello link che si occupa della traduzione degli IP dei livelli sovrastanti in indirizzi MAC. Ogni interfaccia è dotata di un modulo avente una tabella di conversione ARP da IP a MAC

## **PROTOCOLLO IMAP**

È un protocollo “aggiuntivo” per il protocollo del livello applicazione SMTP, che permette la creazione di cartelle per le

diverse email, effettuare ricerche e permette a più client di connettersi allo stesso server mail.

## PORTE DI COMUNICAZIONE

---

Le porte di comunicazione sono delle "interfacce" a cui un client può inviare o ricevere delle richieste.

Alcuni esempi sono la porta 80 per HTTP, 20 per FTP, 23 telnet, 53 DNS, 110 POP3

## PROBLEMA DEGLI IP CON IPv4

---

Gli IPv4 sono indirizzi a 32 bit, di conseguenza esistono soltanto  $2^{32}$  possibili indirizzi assegnabili, questo, sommato alla diffusione esponenziale che ha avuto internet negli ultimi tempi, ha portato all'adozione di un nuovo metodo per fare in modo che gli IP non finissero. IPv6 a differenza di IPv4 presente ip a 128 bit, quindi gli IP possibili sono  $2^{128}$ , inoltre la migrazione completa a IPv6 richiede tempo e risorse. Molti dispositivi e applicazioni potrebbero non supportare IPv6, richiedendo aggiornamenti o sostituzioni.

## FRAMMENTAZIONE CON IP ATTRAVERSO I ROUTER

---

Alcuni pacchetti IP hanno bisogno di essere divisi in pacchetti più piccoli per essere trasmessi, in modo che possano passare attraverso reti aventi diversi MTU (Maximum Transmission Unit) senza essere scartati.

Questa divisione viene effettuata dal router, quest'ultimo, oltre a dividere i diversi pacchetti si occupa anche di cambiare gli header, in particolare:

- Vengono cambiati i sequence number dei diversi pacchetti, mettendoli nell'ordine in cui viene effettuata la divisione
- Viene settato un offset per ogni pacchetto, questo per far sì che all'arrivo al destinatario, quest'ultimo sappia in che ordine "riassemblare i pacchetti"
- Si utilizza il flag MF (More Fragments) a 0 per identificare l'ultimo pacchetto (nel senso che non ci sono ulteriori pacchetti dopo questo)

## **CONNESSIONE PERSISTENTE/ NON PERSISTENTE (+USO DI RICHIESTE SINCRONE IN PIPELINE)**

Una connessione è detta non persistente se ad ogni richiesta (e relativa risposta ricevuta) è necessario effettuare una nuova connessione con l'host. Una connessione è invece detta persistente se è possibile effettuare diverse richieste in sequenza sulla stessa connessione.



Pipelining: al mittente è permesso di inviare più pacchetti senza aspettare conferma di ricezione.

## **SEQUENCE NUMBER**

Il sequence number è uno dei campi dell'header dei segmenti utilizzati al livello trasporto, serve principalmente a identificare i diversi segmenti di dati, per rilevare dei pacchetti mancanti e per il controllo del flusso dei dati, oltre che per riassemblare, nell'host finale, i pacchetti che sono stati divisi dal router perché di dimensione troppo grande

## **ATTACCO PLAYBACK**

---

Un attacco playback è un particolare tipo di attacco informatico atto all'individuazione della password di un utente, a differenza di un normale attacco per scoprire la password, questo non punta a scoprire la password in chiaro, ma a trovare quella criptata.