RSO - Reti e Sistemi Operativi

Sara Angeretti

Ottobre 2023

Indice

Capitolo 1

Organizzazione del corso

1.1 Informazioni sul corso

1.1.1 Lezioni

Crediti: 4 crediti lezione (32 ore) + 4 crediti esercitazione (40 ore).
Due turni

- Turno 1: cognomi dalla A alla L
- Turno 2: cognomi dalla M alla Z

Corso in blended e-learning:

- Lezioni frontali in presenza in aula e da remoto (a causa di ristrutturazione delle aule, tenere sempre controllato il calendario)
- Esercitazioni in e-learning asincrono
- Materiale didattico attraverso il sito del corso (su elearning.unimib.it)
  - Slides e video registrazioni delle lezioni in presenza
  - Materiale video e testuale erogato in e-learning
  - Quiz di autovalutazione
  - Materiale di approfondimento (non oggetto di esame)
  - Indispensabile l'interazione attraverso i forum con docenti, esercitatori e compagni

1.1.2 Personale del corso

Docenti:

- Pietro Braione: docente per la parte di sistemi operativi e responsabile del corso
- Marco Savi: docente per la parte di reti

Esercitatori:

- Jacopo Maltagliati: esercitatore per la parte di sistemi operativi
- Samuele Redaelli: esercitatore per la parte di reti

Tutor:

- Samuele Redaelli: tutor e-learning

1.1.3 Modalità di svolgimento del corso

Le parti di reti e di sistemi operativi si svolgono in contemporanea.
Il calendario del corso, disponibile sul sito, riporta le date delle lezioni in presenza, in remoto a causa di assenza aula, e le date in cui saranno pubblicati i materiali per l'e-learning asincrono.
Il programma del corso (anch'esso disponibile sul sito) riporta le esatte sezioni dei libri di testo da studiare per ciascun argomento del corso, e la distribuzione degli argomenti sulle due prove in itinere.

1.1.4 Il sito del corso

Strumento indispensabile, dal momento che il corso è in blended e-learning!
Aperte le iscrizioni spontanee (iscrivetevi il prima possibile se non siete già iscritti).
Le notizie sul corso verranno comunicate attraverso il forum avvisi.
I materiali didattici vengono distribuiti attraverso il sito.
Sono a disposizione dei forum anonimi per interagire con docenti, esercitatori e tra di voi, allo scopo di discutere gli argomenti del corso e di chiarirvi i dubbi: usateli!

1.1.5 Appelli d'esame

Cinque (o sei?) appelli:

- Due prove in itinere, la prima a novembre 2023 e la seconda a gennaio 2024.
- Due appelli nella sessione di gennaio/febbraio 2024.
- Due appelli nella sessione di giugno/luglio 2024.
- Un appello (o due?) nella sessione di settembre 2024.
- Si può recuperare una (una sola) prova in itinere nel secondo appello della sessione di gennaio/febbraio 2024.

Modalità d'esame:

- Questionario online svolto su computer in laboratorio.
- Le domande possono essere sia teoriche che esercizi che richiedono calcoli, a scelta multipla oppure domande aperte, in qualunque combinazione.
- Ogni prova d'esame comprende sia domande di reti che domande di sistemi operativi: non è possibile sostenere separatamente le parti di reti e di sistemi operativi!
- Regolamento dettagliato di esame disponibile sulla pagina del corso.

1.1.6 Le tempistiche

Parte Sistemi Operativi

Durata Corso (4 CFU)

- 16 ore di didattica frontale
- 10 ore verranno erogate in presenza (aula), le restanti 6 ore online (sincrono)
- Più due ulteriori lezioni in remoto asincrono
- 20 ore di esercitazioni in blended e-learning
- Video e quiz di autovalutazione caricati sulla pagina Moodle secondo calendario didattico
- Possibile qualche incontro in remoto sincrono e un incontro in presenza fuori orario (per chi è interessato)
- Previste inoltre due sessioni di Q&A prima delle prove in itinere

Orario lezioni: vedi calendario sul sito (verranno comunicate di volta in volta così come se in presenza o da remoto).
Controllare sempre il calendario sul sito per sapere se c'è lezione e quando verranno pubblicati i video/quiz per le esercitazioni!

Parte Reti

Durata Corso (4 CFU)

- 16 ore di didattica frontale (in presenza o da remoto a seconda dei giorni)
- Previste inoltre due sessioni di Q&A prima delle prove in itinere
- 20 ore di esercitazioni in blended e-learning
- Video e quiz di autovalutazione caricati sulla pagina Moodle secondo calendario didattico

Orario lezioni: vedi calendario sul sito (verranno comunicate di volta in volta così come se in presenza o da remoto).
Controllare sempre il calendario sul sito per sapere se c'è lezione e quando verranno pubblicati i video/quiz per le esercitazioni!

1.2 Contatti

Parte Sistemi Operativi

Prof. Pietro Braione.
Ufficio: Edificio U14 (DISCo), secondo piano, stanza 2051.
Email: pietro.braione@unimib.it

- Inserire "[RSO]" prima dell'oggetto dell'email!

Telefono: 0264487915
Orario di ricevimento: appuntamento via email.
Team:

- Jacopo Maltagliati - Esercitatore (email: j.maltagliati@campus.unimib.it)
- Samuele Redaelli - Tutor (email: samuele.redaelli@unimib.it)

Parte Reti

Prof. Marco Savi
Ufficio: Edificio U14 (DISCo), Secondo Piano, Stanza 2035
Email: marco.savi@unimib.it

- Inserire "[RSO]" prima dell'oggetto dell'email!

Telefono: 0264487884
Orario di ricevimento: appuntamento via email
Team:

- Samuele Redaelli - Esercitatore (email: samuele.redaelli@unimib.it)
- Samuele Redaelli - Tutor (email: samuele.redaelli@unimib.it)

1.3 Obiettivi del corso

Parte Sistemi Operativi

Acquisire le conoscenze fondamentali relativi ai sistemi operativi:

- A cosa servono i servizi operativi? Perché sono necessari?
- Che servizi offrono ai programmi e agli utenti di un sistema?
- Come sono implementati i sistemi operativi e i servizi che offrono?

Particolarmente importanti sono le esercitazioni pratiche, nelle quali imparerete ad usare i servizi di un sistema operativo moderno (Linux).

- Necessario un computer laptop
- Sono argomento di esame!

Parte Reti

Acquisire le conoscenze fondamentali per comprendere l'architettura e i protocolli principali delle reti di telecomunicazioni basate sullo stack TCP/IP.

- Lo stack TCP/IP è alla base della quasi totalità dei servizi di comunicazione.

Al termine del corso avrete appreso i principi fondamentali del funzionamento di Internet.

1.4 Materiale e strumenti didattici

Parte Sistemi Operativi

Libro di riferimento:

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Sistemi Operativi - Concetti e Esempi, Decima edizione, Pearson, 2019.
- Versione in inglese: Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating Systems Concepts, 10th Edition, John Wiley and Sons, 2018.

Materiale online su Moodle.

- Slides e registrazioni video delle lezioni
- Quiz di autovalutazione
- Video delle esercitazioni

Forum tematico anonimo di discussione su Moodle (indispensabile per le esercitazioni!)

Parte Reti

Materiale online su Moodle

- Slides ufficiali del libro di riferimento (rivisitate, in inglese)
- Video sulla parte di esercitazioni

Libro di riferimento

- Jim Kurose, Keith Ross, Reti di calcolatori ed Internet - Un approccio top-down, Ottava edizione, Pearson, 2021
- Versione in inglese: Jim Kurose, Keith Ross, Computer Networking - A Top-Down Approach, 8th Edition, Pearson, 2021

Forum su Moodle per la parte di reti (specialmente utile per la parte di esercitazioni...)

1.5 Programma del corso

Parte Sistemi Operativi

Sistemi operativi: struttura e servizi
Servizi:

- Processi e thread: i servizi
- Gestione della memoria: i servizi
- File system: i servizi

Struttura:

- Interfaccia e struttura del kernel
- Processi e thread: la struttura
- Gestione della memoria: la struttura
- File system: la struttura

Parte Reti

Parti specifiche del libro (da Capitolo 1 a Capitolo 6)

- Capitolo 1: Introduzione alle reti di calcolatori e Internet
- Capitolo 2: Livello di applicazione [cenni]
- Capitolo 3: Livello di trasporto
- Capitolo 4: Livello di rete - Piano dei dati
- Capitolo 5: Livello di rete - Piano di trasporto
- Capitolo 6: Livello di collegamento e reti locali

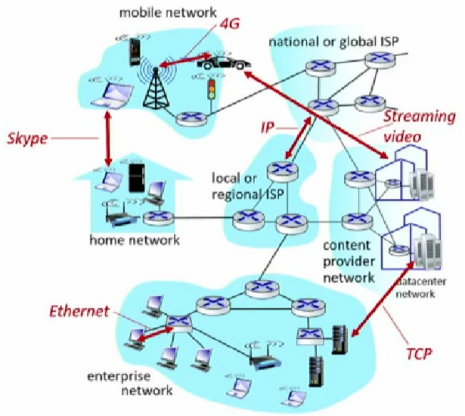
Capitolo 2

Introduzione alle reti

2.1 Cos'è Internet?

Due prospettive:

- Visione degli ingranaggi (dadi e bulloni), delle componenti della rete.
- Visione della rete come un'infrastruttura che fornisce servizi.



Nell'immagine che schematizza, sui bordi della rete troviamo un grandissimo numero di devices (cell, laptops, ...) che eseguono *applicazioni di rete* che richiedono uno scambio di dati e che la rete si occupa di collegare. Alcuni nodi di questa applicazione eseguono parte di un'applicazione e altri nodi interconnessi eseguono altre parti di applicazioni (sistemi distribuiti).

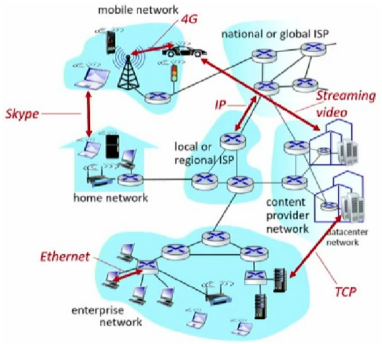
2.1.1 Visione nuts and bolts

hosts: dispositivi *end system* ovvero terminali.

packet switches o *commutatori di pacchetto*: sono *routers* e *switches*, dispositivi che si occupano di trasferire pacchetti (unità) di informazione.
I pacchetti sono unità in cui l'informazione viene scomposta ed etichettata per essere trasferita da un dispositivo all'altro.

Communication links o *collegamenti di comunicazione*: sono i canali che collegano i nodi della rete. Li disegnamo come righe che uniscono i vari nodi. Possiamo crearli usando diversi mezzi trasmissivi, filo di rame, onde radio, fibra ottica... Ognuno di questi link è definito da una **banda**, la quantità di informazione che il link può trasferire in un secondo.

Networks: Internet è una rete di reti, ovvero una rete di dispositivi che sono collegati tra loro e che a loro volta sono reti di dispositivi collegati tra loro. Quindi tutta la rete è una connessione globale di reti locali (es. la rete dell'ufficio, la rete domestica...)



Sempre più dispositivi oggi giorno si sono evoluti fino ad adattarsi all'uso di Internet, come ad esempio le auto, i frigoriferi, le lavatrici...

Quindi anche Internet deve evolversi e adattarsi per poter gestire un numero sempre maggiore di dispositivi con esigenze sempre più diverse.

Internet come rete di reti

Abbiamo detto che Internet è viibile appunto come una rete di reti, un *insieme interconnesso* di **ISPs** (Internet Service Providers), le organizzazioni che gestiscono la rete. A volte ISPs viene usato anche come sinonimo della rete vera e propria.
Gli ISPs comunicano fra loro tramite l'uso di **protocolli**. I protocolli nella rete sono ovunque, controllano il modo in cui i messaggi vengono inviati e ricevuti tra i dispositivi. Alcuni esempi sono HTTP (Web), streaming video, Skype, TCP, IP, WiFi, 4G, Ethernet...
Questi protocolli si basano su **standards**.
Un lavoro fondamentale a riguardo è svolto da **enti di standardizzazione** (il più famoso è IETF, *Internet Engineering Task Force*) che stilano documenti (RFC, *Request for Comments*) che spiegano come i protocolli devono essere implementati e come i dispositivi che implementano questi protocolli devono

comportarsi.
Gli standard sono fondamentali per la comunicazione univoca tra i dispositivi, senza di essi non ci sarebbe interoperabilità tra i dispositivi.

2.1.2 Visione come infrastruttura di servizi

Internet può essere visto come un'**infrastruttura** che fornisce **servizi** alle applicazioni distribuite.
È una vista importante perché la rete sottostante è fondamentale per quando dobbiamo per esempio programmare delle applicazioni o dei servizi: da questo pov è molto importante il concetto di socket, un'interfaccia che ci permette di interfacciarci alla rete senza necessariamente sapere come la rete sotto funziona.

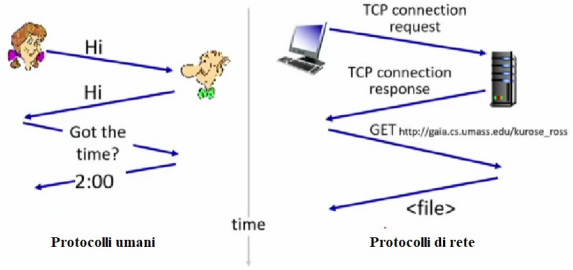
2.2 Protocolli

2.2.1 Cos'è un protocollo?

Protocolli umani	**Protocolli di rete**
I protocolli umani sono le regole che seguono gli esseri umani quando comunicano tra loro. Es.: incontro una persona, scambio di convenevoli, poi chiedo: "Che ore sono?" e in base alla risposta ho delle reazioni diverse. Altro es.: "Ho una domanda" durante una lezione, così che per non interrompere il professore, lui possa finire il suo discorso e poi rispondere alla domanda. Altro es.: un giro di tavolo per fare le presentazioni, seguo delle convenzioni di discorso.	Emulano il funzionamento dei protocolli umani. Tutte le comunicazioni di rete sono gestite da protocolli.

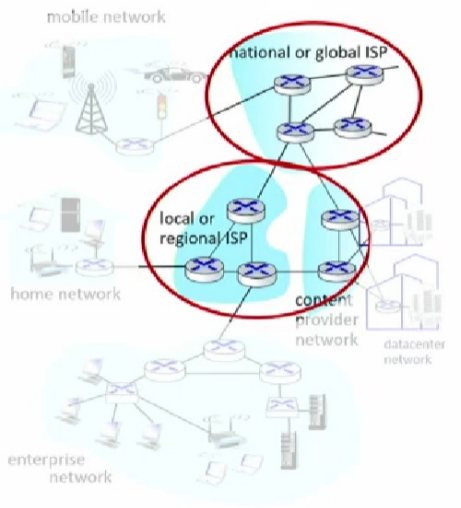
DEFINIZIONE

I protocolli di rete definiscono le regole per i messaggi inviati in rete. In particolare definiscono il *formato dei messaggi*, *l'ordine* in cui vengono *inviati* e *ricevuti* tra le entità di rete (host, commutatori, ...) e le *azioni da intraprendere* una volta che questi messaggi vengono inviati e ricevuti.



2.3 Da cosa è composta la rete?

Andiamo più nel dettaglio.



Network edge

Primo segmento della struttura generale della rete. Ci troviamo sui bordi della rete (quindi c'è un po' una diatriba sul fatto che appartengano o meno alla rete, in ogni caso sono molto importanti), dove si trovano gli **hosts** (client, server). Sono intesi in senso un po' lato, ovvero:

- I *client* sono intesi come dispositivi con una *bassa* capacità computazionale.
- I *server* sono intesi come dispositivi con una *alta* capacità computazionale. Infatti di solito si trovano nei data center.

Access networks

Addentrandoci ancora più nella rete, abbiamo le reti di accesso. Tipicamente hanno dei collegamenti di comunicazione che possono essere *wired* (cavi, stoppini, fibre ...) e *wireless* (4G, onde radio...). Sono molto importanti perché accolgono il traffico dagli utenti e lo portano verso la rete. O viceversa accolgono il traffico di dati da passare al client. Per accedere alla rete bisogna acquistare un accesso alla rete per mezzo di un provider, un ISP.
Sono importanti perché evolvono molto rapidamente nel tempo, sostengono sempre più il cambiamento e l'evoluzione del traffico generato dagli utenti a loro volta in costante evoluzione.
Non parleremo delle reti di accesso.

Network core

La rete di core è una rete con una maglia di dispositivi solitamente molto performanti (quindi in grado di gestire una gran quantità di informazione) e che si

trovano al centro della rete. Permettono di implementare quella rete di reti di cui abbiamo parlato prima.
Per mezzo delle reti di core garantisco che ogni utente che si trova in rete possa comunicare con ogni altro utente che si trova nella stessa rete. Ovviamente non è sempre concretamente possibile.
Internet è una rete connessa: da un nodo posso raggiungere un qualsiasi altro nodo. Questo è quello di cui si occupano le reti core: permettono di mettere in comunicazione reti più al limitare della rete di altre.

2.3.1 Network edge

L'host, abbiamo detto, ha il compito di inviare pacchetti di dati, prende un messaggio applicativo che deve inviare e lo scompone in pacchetti di lunghezza pari ad L (per semplicità ora li consideriamo tutti della stessa lunghezza, ma di solito hanno dimensioni di lunghezza variabile). Questi pacchetti vengono inviati dall'host in rete dalle reti di accesso che forniscono l'accesso ad un rate trasmissivo (banda, capacità...) pari a R (è in bit/s). Questo vuol dire che l'host può inviare pacchetti di lunghezza L a velocità R.

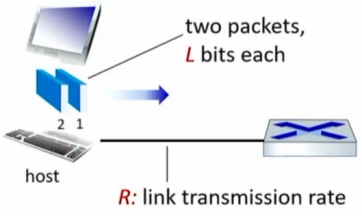
Ovviamente più è alta la banda, più è alta la velocità di trasmissione.

Ma come si calcola il **ritardo di trasmissione**?

DEFINIZIONE

Il *ritardo di trasmissione* è il ritardo che intercorre tra l'invio del primo bit di un pacchetto di L bit alla ricezione dall'altro lato del link dell'ultimo bit dello stesso pacchetto.

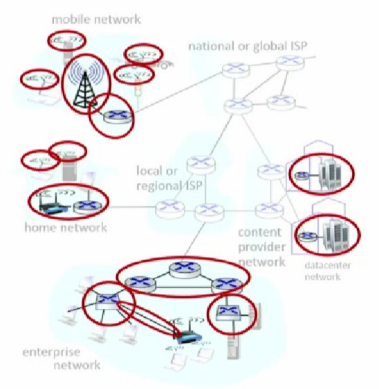
Essendo L la lunghezza del pacchetto e R la velocità di trasmissione, il *ritardo di trasmissione* sarà pari a L/R.



2.3.2 Access networks

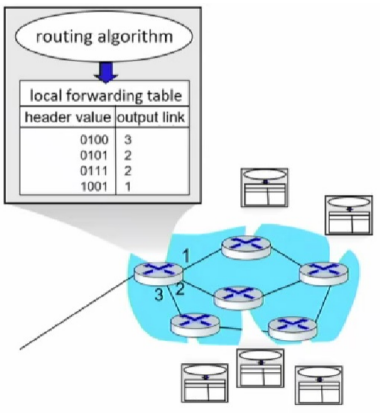
Abbiamo detto che Internet è un *packet switching network*, una rete a commutazione di pacchetto. L'unità informativa che viene scambiata in rete è il **pacchetto** che appunto viaggiano in rete, ricevuti e trasmessi da più dispositivi.
Il compito della rete è di inoltrare i pacchetti verso i router o i commutatori di pacchetto sul link corretto che va da sorgente a destinazione. Questo è il compito principale dei dispositivi della rete di core: creare un percorso (possibilmente il migliore) fra sorgente e destinazione.

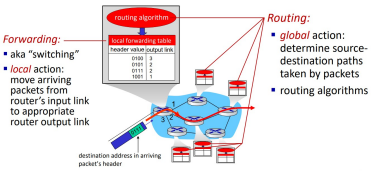
Sembra, almeno concettualmente, abbastanza banale come concetto. Ma la gerarchia della rete è studiata in modo che nodi sufficientemente vicini possano essere messi in contatto senza passare da router o commutatori.



2.3.3 Network core

Come sono fatti i nodi della rete di core:





Forwarding table: è una tabella che contiene le intestazioni dei vari pacchetti (dove c'è specificato l'indirizzo della destinazione e quale interfaccia ovvero l'uscita del link da cui il pacchetto deve uscire) da inviare verso una specifica interfaccia.
L'**inoltro** (forwarding) ha ovviamente valenza locale. Ogni singolo nodo consultando la propria tabella prende decisioni autonome riguardo l'inoltro.

Switching: commutazione, sinonimo di inoltro (forwarding). Le due terminologie hanno una valenza simile ma si riferiscono a due cose leggermente

diverse: l'*inoltro* è l'operazione di ricevere un messaggio e inoltrarlo verso un altro nodo, mentre la *commutazione* è l'operazione per trasferire il pacchetto da un'interfaccia di ingresso ad una di uscita di un router.
Il risultato è lo stesso, ma lo *switching* è più a livello interno del nodo ignorando ciò che sta all'esterno, il *forwarding* invece tiene in considerazione che io prendo e inoltro da un nodo un pacchetto che sposto poi attraverso 1+ link ad un altro nodo.

Routering: ogni singolo nodo prende decisioni localmente, ma io devo garantire che una volta che diversi nodi effettuano l'operazione di inoltro in serie il pacchetto arrivi da sorgente a destinazione senza intoppi. Parliamo di **routing**, instradamento, azione **globale** che ha l'obiettivo di trovare un percorso tra sorgente che invia il pacchetto a destinazione che lo riceve. In ognuno dei nodi avremo un **algoritmo di routing** che agisce in maniera distribuita; un algoritmo di routing comunica con un altro algoritmo di routing tramite protocolli di routing che permettono di popolare la **tabella di forwarding** in maniera corretta.

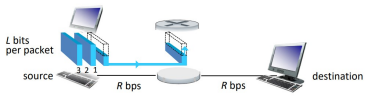
2.4 Modalità packet-switching

2.4.1 Store-and-forward

La rete internet abbiamo detto essere a commutazione di pacchetto e opera secondo la modalità store-and-forward.

Cos'è lo store-and-forward?

È quella modalità per cui un pacchetto prima di poter essere inviato attraverso un link (ad esempio verso un router), deve essere prima inviato nella sua interezza dal nodo da cui arriva. Questo è dovuto al fatto che ogni pacchetto è dotato di un'intestazione che se persa mi fa perdere anche informazione su cosa sia quel pacchetto.
Nell'immagine abbiamo una sorgente, un commutatore e una destinazione.



Se per dire inviassi un bit al commutatore e lui lo spedisse verso la destinazione senza aspettare il resto del pacchetto dalla sorgente, se il bit si perdesse non saprei a chi quel bit appartiene.

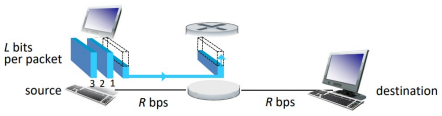
Perché lo store-and-forward è importante?

La modalità store-and-forward è importante quindi perché semplifica notevolmente come le reti debbano essere costruite, prima di inviare un pacchetto devo riceverlo tutto, ricostruirlo, e analizzare l'intestazione per sapere cosa contiene e che è legata a quello specifico pacchetto.

Problemi dello store-and-forward

Introduce dei ritardi, ovviamente, dovendo aspettare che un pacchetto sia ricevuto per intero prima di poterlo inoltrare. Questo ritardo vale $2\frac{L}{R}$ secondi a trasmettere attraverso un link un pacchetto di L bits ad una velocità R di trasmissione. Questo perché impiega $\frac{L}{R}$ secondi per trasmettere il pacchetto al commutatore e altri $\frac{L}{R}$ secondi per trasmettere il pacchetto dal commutatore alla destinazione.
Se invece non usassi questa modalità e ad esempio facessi passare un bit alla volta attraverso il commutatore, impiegherei meno tempo: agirebbe come se il commutatore non esistesse e i due link di velocità R fossero direttamente collegati tra loro, come un unico link più lungo, quindi la velocità complessiva sarebbe comunque R e il tempo di trasmissione **totale** sarebbe $\frac{L}{R}$ secondi. Ovviamente avrei meno ritardi ma altri problemi che vedremo in seguito.

2.4.2 Esempio di esercizio

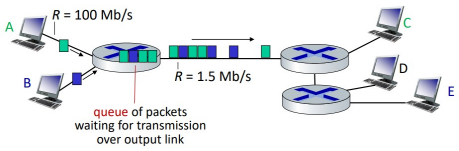


Se come in figura avessi tre pacchetti alla sorgente in attesa di essere inviati (P1, P2 e P3), quanto tempo (in multipli di $\frac{L}{R}$) passerebbe fra l'invio dalla sorgente del primo bit di P1 alla ricezione della destinazione dell'ultimo bit di P3?
La risposta è $4\frac{L}{R}$. Perché?

1. P1 viene inviato al commutatore in $\frac{L}{R}$ secondi.
2. P1 viene inviato alla destinazione e P2 viene inviato al commutatore in $\frac{L}{R}$ secondi.
3. P2 viene inviato alla destinazione e P3 viene inviato al commutatore in $\frac{L}{R}$ secondi.
4. P3 viene inviato alla destinazione in $\frac{L}{R}$ secondi.

2.4.3 Queueing

Abbiamo parlato di possibili problemi riscontrabili con lo store-and-forward. Uno dei principali svantaggi è che in una rete a commutazione di pacchetto si verifica quello che si chiama **accodamento di pacchetti** o **queueing**.

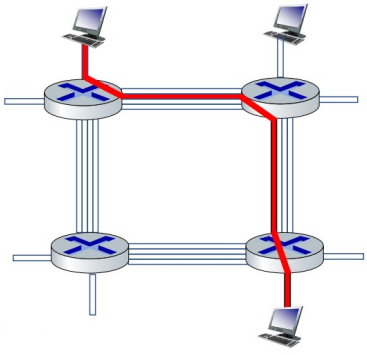


Nella maggior parte dei casi quello che può verificarsi è che il rate a cui io posso trasmettere in uscita sul mio link è più basso rispetto a quello con cui ricevo i pacchetti. Nell'immagine ho un R pari a $100^{Mb}/_{s}$ sui link in ingresso e un R

pari a $1.5^{Mb}/_{s}$ sui link in uscita, *molto* più basso. Arriverò ad un punto in cui non riesco a smaltire i miei pacchetti che si accumulano sul router ad un ritmo sufficientemente sostenuto da tenere il passo del ritmo con cui li ricevo. Allora comninciano ad accumularsi in *buffer*, in zone di memoria, e si crea una coda di pacchetti in attesa di essere trasmessi sul link in uscita.
Queste zone di memoria del commutatore hanno però una capacità limitata (ovviamente) e quando il rate del link in uscita è troppo inferiore rispetto al rate in ingresso al commutatore e quindi si accumulano i pacchetti, rischio di andare in ***buffer* overflow**: è rischioso perché quando lo raggiungo poi perdo i pacchetti che incuranti continuano ad arrivare, e che vengono scartati.
Questo problema è chiamato **il problema della perdita nelle reti a commutazione di pacchetto**.
Un altro problema è il **ritardo di accodamento**.

2.4.4 Alternativa alla modalità packet-switching: circuit-switching

Una rete a commutazione di circuito si basava su principi molto diversi da quelli visti finora.
Al giorno d'oggi non è più usata, ma è importante conoscerla perché è stata la prima modalità di funzionamento delle reti di telecomunicazione. Un esempio è la vecchia rete telefonica (fissa di casa).
Si hanno delle risorse dedicate esclusivamente alla chiamata o al circuito, quando si stabilisce una chiamata si stabilisce un circuito dedicato che rimane dedicato per tutta la durata della chiamata. Una certa quantità di banda viene riservata esclusivamente alla comunicazione: stabilendo un circuito end-to-end di risorse dedicate alla comunicazione.



Vantaggi: non ho i problemi tipo l'accodamento, non ho ritardi di accodamento, non ho perdita di pacchetti. Quindi migliori prestazioni.

Svantaggi: scarsa capacità, ho per esempio nell'immagine un numero massimo di 4 dispositivi collegabili. Quindi peggior utilizzo e scarsa condivisione delle risorse.

2.4.5 Packet-switching vs circuit-switching

Boh non c'è tanto da scrivere, spesso diceva che non lo vediamo nel corso quindi salterei.

2.5 Struttura della rete: reti di reti

Abbiamo detto che gli *hosts* sono connessi da *links* e *ISPs*.
Come più volte abbiamo visto, la rete ha una struttura di tipo **gerarchico**: diversi provider forniscono connettività di tipi diversi. Ci sono nel mondo milioni di reti che fanno da punto di accesso a diversi clients/hosts, e devono essere connesse tra loro.

Prima ipotesi

Creare un punto di accesso verso tutte le altre reti di accesso. Non è affatto fattibile!! Non scala: il numero di collegamenti diretti che io dovrei avere fra le reti di accesso è dell'ordine di $O(n^2)$.

Seconda ipotesi

Creare un rete di un ISP che connette tutte le reti di accesso. Questo è quello che succede oggi giorno. Si interconnette a tutte le reti di accesso e grazie ad una rete di commutatori permette la commutazione fra tutte quelle reti di accesso. Si crea una rete in cui ogni rete di accesso paga l'ISP per poter avere l'accesso ed essere messo in comunicazione con tutte le altre.

2.5.1 Problemi e soluzioni

Una rete così fatta dovrebbe essere geometricamente estesissima, e questo non è fattibile. Si va quindi a creare una rete di reti di ISP, geometricamente più limitate ma anche numerose. Così una rete può scegliere da chi comprare l'accesso.
Il problema che sorge ora è che se una rete di ISP non è in comunicazione con un'altra rete di ISP, come faccio a mettere in comunicazione due reti di accesso che sono collegate a due reti di ISP diverse?
Si vanno a creare allora dei link che le mettano in comunicazione, collegamenti:

peering link: link fra pari, tra un ISP di una rete di ISP e un altro di un'altra rete.
Vantaggioso ad entrambi.
Due tipi diversi:

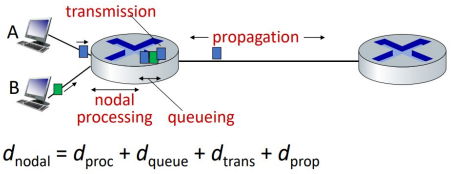
- **regional ISP:** collegamento fra **una** ISP e diverse reti di accesso a cui fornisce accesso.
- **Multi-homing:** collegamento fra **diverse** ISP e diverse reti di accesso a cui fornisce accesso, migliore perché nel caso cada la connessione con un ISP ci sono gli altri a mantenere attiva la connessione.

IXP: internet exchange point, interconnessioni fra ISP di reti diverse. Punti di contatto fra diversi ISPs.
Il MIX è quello di Milano e il più importante d'Italia.

Content provider network: esempio Google, straordinariamente geograficamente estesa in modo da prendere quanti più ISPs possibili. Ha un vantaggio per tutti, comprese le reti di accesso, hanno tutti accessi privilegiati.

2.5.2 Ritardi introdotti dalla packet-switching

4 tipi diversi di ritardo:



Il ritardo di queueing è l'unico che può variare notevolmente da pacchetto a pacchetto. Gli altri sono (all'incirca) costanti per pacchetti diversi. Uno che non abbiamo visto è il **ritardo di elaborazione al nodo**. Ritardo introdotto dal commutatore per effettuare alcune operazioni, tra cui la lettura della tabella di inoltro (*look up* la *table*) per recuperare le informazioni sul link di output. Ogni volta che un pacchetto arriva al commutatore, il commutatore deve analizzare l'intestazione del pacchetto per capire se ci sono stati problemi di trasmissione. Ci sono tutti dei codici/meccanismi per controllare ed eventualmente correggere questi eventuali errori (se non li correggo rischio di perdere il pacchetto, se li correggo lo posso salvare). Questo prende tempo, che diventa il **ritardo di elaborazione (d_{proc})**, questo tempo è dell'ordine di grandezza di 10^{-9} secondi. Un altro dato decisamente più importante (2-3 ordini di grandezza più grande, $10^{-6} - 10^{-3}$ secondi) è il **ritardo di accodamento (d_{queue})**. L'abbiamo già visto. È il tempo che un pacchetto aspetta sul mezzo di trasmissione di uscita (cavo) prima di essere trasmesso, dipende dal livello di congestione del router. Poi abbiamo il **ritardo di trasmissione ($d_{trans} = {}^{L}/_{R}$)**, anche questo già visto, che dipende dalla lunghezza del pacchetto ($L, bits$) e velocità di trasmissione (R, bps). Poi abbiamo il **ritardo di propagazione ($d_{prop} = {}^{d}/_{s}$)** tempo che un pacchetto impiega praticamente a fluire da un'estremità all'altra del pacchetto. Dipende dal mezzo di trasmissione, quindi lunghezza del cavo (d) e velocità di propagazione (s, circa pari alla velocità della luce, $\sim 2 \times 10^8 \frac{m}{sec}$).
A sommare tutte queste componenti, ottengo il tempo complessivo che il pacchetto ci mette a passare da sorgente a destinazione.

Recap:

Nome del ritardo	Sigla	Tempo
R. di elaborazione	d_{proc}	10^{-9} sec
R. di accodamento	d_{queue}	$10^{-6} - 10^{-3}$ sec
R. di trasmissione	d_{trans}	${}^{L}/_{R}$
R. di propagazione	d_{prop}	${}^{d}/_{s}$

Dove:

L:	lunghezza del pacchetto	$[bit]$
R:	velocità di trasmissione	$[{}^{bit}/_{sec}]$
d:	lunghezza del mezzo	$[m]$
s:	velocità di propagazione	$[{}^{m}/_{sec}] = \sim 2 \times 10^8 {}^{m}/_{sec}$

Una cosa fondamentale è la differenza fra *ritardo di trasmissione* e *ritardo di propagazione*: entrambi dipendono dal mezzo su cui il pacchetto viene trasferito, ma sono diversi. Il primo dipende dalla dimensione del pacchetto e il rate di trasmissione del commutatore; il secondo dalla lunghezza del mezzo propagativo (cavo, fibra ottica, whatever) e la sua capacità di propagazione, che è una velocità ma non la stessa dell'altro ritardo.

Es.

Immaginiamo di essere in tangenziale. Arrivo alla barriera e mi fermo al casello di boh Sesto, poi devo arrivare alla barriera di Legnano. Immaginiamo di avere un certo numero di macchine (i bit) che fanno parte di uno stesso pacchetto.
Per semplicità un solo casello. La differenza fra r. di trasmissione e r. di propagazione è: mettiamo di avere un casellante che fa passare 4 macchine al minuto, che portano ad un ritardo di trasmissione R_T (che ovviamente diminuisce all'aumentare della capacità del trasmittente). Se il casellante si sbriga, passano più macchine (più bit al minuto) e quindi il ritardo di trasmissione è più basso. Ovviamente c'è un limite fisico di velocità. Una volta che il casellante mi fa passare, io devo percorrere la tangenziale fino al casello successivo, che è un ritardo di propagazione R_P che dipende dalla lunghezza della tangenziale e dalla velocità massima che posso raggiungere.

2.5.3 Ritardo *end-to-end*

È il ritardo che intercorre per il processing fra sorgente e destinazione, introdotto da tutti i commutatori di pacchetto fra uno e l'altro e i ritardi introdotti dai sistemi periferici.
Abbiamo detto che è la somma delle 4 sorgenti di ritardo: $d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$.
Quando parliamo di *ping* però facciamo riferimento al tempo che ci mette un pacchetto a passare da sorgente a destinazione e di nuovo indietro da destinazione alla sorgente, prende il nome di ritardo *round-trip-time*.

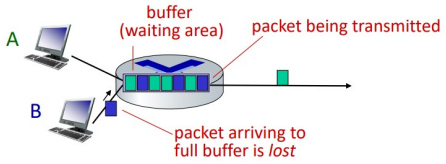
2.5.4 Di più sul ritardo di queueing

Il ritardo di queueing cambia da pacchetto a pacchetto.
Nel caso più generico possibile dipende dal tasso di **intensità di traffico**. Come si calcola? Con strumenti statistici. Abbiamo un rate medio di arrivo dei bit ($\bar{a}$), le lunghezze dei pacchetti (L) e la velocità di trasmissione del mezzo (R).
L'intensità di traffico è definita come $L \cdot \bar{a} / R$.
Se l'intensità di traffico tende a 0, allora il ritardo di queueing tende ad essere molto piccolo.
Se l'intensità di traffico tende a 1, allora il ritardo di queueing tende ad essere larghino.
Se l'intensità di traffico è maggiore di 1, allora il ritardo di queueing tende a infinito ed è ingestibile.
N.B.: stiamo parlando di valori medi!!!

2.5.5 Perdita di pacchetti

Più cause:

- capacità del commutatore satura (buffer overflow) che porta alla perdita di pacchetti inviati dopo che la capacità finita è stata riempita.
- problemi sul mezzo trasmissivo



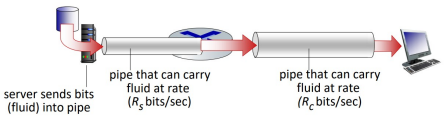
I pacchetti con errori di trasmissione vengono scartati. Potrebbe succedere che in qualche modo si è perso il pacchetto e decidere di ri-inviarlo ma non è scontato.

2.5.6 Throughput

Calcolato come una velocità, rate (bits/time unit) a cui i bits sono inviati da sorgente a destinazione (a volte chiamato ***end-to-end throughput***, anche se questi due termini sono abbastanza simili e quindi possono essere usati singolarmente) ed è:

istantaneo: rate ad un certo punto dato nel tempo

medio: rate in un periodo più esteso di tempo



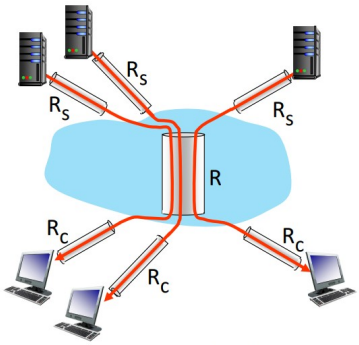
Domande:

$R_s < R_c$**:** quale è il throughput medio?
R_s perché il rate di trasmissione è più basso del rate di ricezione.
In questo caso non c'è accodamento sul commutatore.

$R_s > R_c$**:** quale è il throughput medio?
R_c perché il rate di trasmissione è più alto del rate di ricezione.
In questo caso c'è accodamento sul commutatore.

Parliamo di ***bottleneck link***, che è quel link sul percorso tra sorgente e destinazione (ovvero il percorso end-to-end) che vincola il throughput end-end. Nel primo caso è R_s, nel secondo è R_c.

Throughput: network scenario



Mettiamo di avere una rete come nella situazione in immagine. Abbiamo un certo numero (qua 10) di server, sopra nell'immagine, che acquistano un accesso alla rete per mezzo di un provider di una rete di accesso e hanno quindi link con velocità R_s. Immaginiamo vogliano comunicare con i client (diciamo ancora 10) che si trovano in basso, e che hanno accesso alla rete per mezzo di altri link con dimensione R_c.
Immaginiamo ora di modellare la rete di core come un link di dimensione R (oppure, posso dire che nella rete di core ho un link di dimensione R condiviso da tutte le connessioni in modo **fair**, ovvero equamente, quindi con rate $R/10$). Il throughput sarà dato dal minimo fra questi, ovvero $min(R_c, R_s, R/10)$.
Difficilmente $R/10$ sarà il bottleneck link: è decisamente sottodimensionato rispetto agli altri, tipicamente il bottleneck link sarà R_c o R_s.

Capitolo 3

Reti e livelli di rete

3.1 Architettura a strati

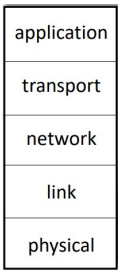
N.B: livelli e strati sono sinonimi qua. Partiamo dal presupposto che le reti hanno una struttura complessa. Esiste una possibilità di studiare e organizzare questa struttura? Un modo è quello di suddividere la complessa struttura in strati, ognuno dei quali si occupa di servizi diversi ed è in comunicazione solo con quelli ***immediatamente*** **adiacenti**.

Perché fare strati?

La stratificazione di un sistema permette di andare a definire un modello di riferimento e permette un'identificazione semplice delle relazioni dei vari pezzi del nostro sistema. La modellizzazione è vantaggiosa perché se vado a modificare l'implementazione di una componente, questo risulta essere trasparente a tutto il sistema e non va ad impattare sull'implementazione degli altri.

3.1.1 Architettura a strati dell'Internet

Anche nota come *Layered Internet protocol stack*, lo stack protocollare si legge dall'alto verso il basso.



applicativo: supporta applicazioni di rete.
Es.: HTTP (non vedremo), SMTP (protocollo per l'invio di mail), DNS (vedremo fra un attimo).

trasporto: trasferisce i dati tra processi diversi eseguiti (di solito) su macchine diverse.
Es.: TCP, UDP.

rete: instrada i datagrammi (pacchetti a livello di rete) da sorgente a destinazione.
Es.: IP, routing protocols.

link: trasferisce i dati tra nodi, elementi di rete, adiacenti.
Es.: Ethernet, 802.11 (WiFi).

fisico: trasferisce i bit su un canale fisico.
È il livello che non vedremo.

3.2 Servizi, stratificazione, incapsulamento

In che modo posso andare effettivamente ad implementare i vari servizi nei vari livelli?

3.2.1 A livello di sorgente

I livelli **applicativi** si scambiano messaggi per implementare specifici servizi. Questi messaggi vengono trasportati sfruttando i servizi offerti dai livelli di **trasporto**. Questi livelli di *trasporto* hanno il compito di far comunicare diversi processi su macchine diverse e possono trasferire i messaggi (nel caso di TCP) in maniera affidabile, garantendo che il messaggio arrivi in maniera corretta a destinazione.

Incapsulamento

Per implementare i servizi del livello di trasporto si fa in questo modo:

- Prendo i bit del mio messaggio (M) e ci attacco un'intestazione chiamata ***Header*** (H_t), insieme di bit organizzati in **campi**.
- Il messaggio (M) prende il nome di ***payload***.
- Il messaggio così ottenuto ($H_t + M$), che prende il nome di ***segmento***, viene passato al *livello di rete* che è immediatamente sotto.
- Lo strato di rete (*network*) prende il *payload* e ci aggiunge un **suo header**, il messaggio diventa così $H_n + H_t + M$, prende il nome di ***datagram*** e viene passato al livello di link.
- Lo strato di link prende il *datagramma* e ci aggiunge un **suo header**, il messaggio diventa così $H_l + H_n + H_t + M$, prende il nome di ***frame*** *(trama)* e viene passato al livello fisico.

Così si crea la separazione fra i livelli, ogni livello vede i pochi bit passati dal livello precedente.
Alla fine arrivo ad avere un messaggio molto più lungo di quello originale, ma che contiene tutte le informazioni necessarie non ai fini del messaggio ma piuttosto per il funzionamento del sistema. Tutta la parte in eccesso (oltre al *payload* M) prende il nome di ***overhead*** ($H_l + H_n + H_t$). In breve, l'incapsulamento, prende e aggiunge l'intestazione (unica cosa che guarderò) senza preoccuparmi di ciò che ci sta prima (payload). Il succo è che ad ogni livello aggiungi un header.
Il messaggio non viene inviato finché non è stato completamente incapsulato. Deve attraversare tutto il percorso stratificato fino ad arrivare allo strato fisico e poi viene inviato al destinatario.

3.2.2 A livello di destinazione

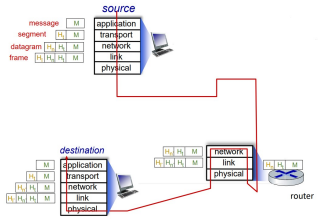
Percorso inverso, partendo dal livello fisico e continuando fino ad arrivare al livello applicativo, si tolgono gli header e si ottiene il messaggio originale.

- Lo strato di link prende il **frame** e toglie il suo header H_l, il **frame** diventa quindi $H_n + H_t + M$ e viene passato allo strato di rete.

- Lo strato di rete prende il **datagram** e toglie il suo header H_n, il **datagram** diventa quindi $H_t + M$ e viene passato allo strato di trasporto.
- Lo strato di trasporto prende il **segmento** e toglie il suo header H_t, il **segmento** diventa quindi M ovvero il mio *payload* e viene passato allo strato applicativo.

A livello di commutatori

Negli *end-host* (sistemi periferici) ovviamente ci sono **tutti e 5** gli strati, a livello di commutatori invece è diverso: ci sono solo i primi 2 (switch) o 3 strati (router), ovvero il *livello fisico*, quello *di link* e quello *di rete*.
I router sono dotati di *commutatori di livello 3*.
Es. semplice:



In questo caso:

- il *frame* $(H_l + H_n + H_t + M)$ creato dal mittente passa al router, che abbiamo detto avere *solo strato fisico, di link, di rete*.
- Il router toglie l'header (H_l) del livello *di link* e inoltra il messaggio al livello *di rete*.
- Il livello *di rete* prende il messaggio $(H_n + H_t + M)$ e lo incapsula con un header diverso (che noi chiamiamo H'_n).
- Il *datagramma* così ottenuto $(H'_n + H_n + H_t + M)$ viene incapsulato in una nuova *trama* e inviato al destinatario.

Quindi in due punti diversi del percorso il messaggio potrebbe essere incapsulato in due header diversi perché magari vengono usati due protocolli internet diversi.

Capitolo 4

Livello applicativo

Vedremo ben poca roba, verrà approfondito meglio in altri corsi.

4.1 DNS: Domain Name System

Funzione importante del protocollo internet, implementato a livello applicativo. La sua complessità per questo motivo è limitata ai bordi della rete (a livello applicativo sono coinvolti sono gli end-users, clients e servers, non commutatori). Così come le persone hanno degli identificatori (quali nome, cognome, carta d'identità, codice fiscale, passaporto etc.), anche in rete avviene la stessa cosa. In particolare ci serve:

- host name (es. "www.unimib.it")
- indirizzo IP (indirizzo di 32 bit, tecnicamente associato all'interfaccia e non al server).

Quando scrivo e invio un host name, quello che concretamente faccio è generare una richiesta di DNS che mi porti a risolvere quell'host name. Questo è ciò che fa il DNS: permette di restituire sulla base dell'host name l'indirizzo IP su cui io posso reperire quello specifico contenuto a livello di interfaccia di rete del server.

4.1.1 Mapping tra indirizzi IP e host names

Per mezzo del **Domain Name System** (DNS) è possibile mappare gli indirizzi IP con gli host names.

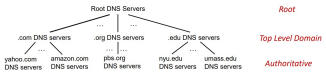
4.1.2 Cos'è un DNS?

È un database distribuito implementato seguendo una gerarchia e utilizzando diversi server che prendono il nome di ***DNS*** *(o* ***nameserver****)*.
Il servizio di traslazione degli indirizzi non è facilmente implementabile tramite un unico server (che per ogni singolo host name restituisce l'indirizzo IP associato), non sarebbe scalabile (in internet ho una quantità gigantesca di host names, una singola entità non potrebbe possibilmente risolverli tutti).

Per questo è stata creata la gerarchia per distribuire questo carico su più nodi con diverse responsabilità.

4.1.3 Come funziona un DNS?

Struttura gerarchica dei DNS



Abbiamo:

root DNS servers: sono al top della gerarchia, gestiti da ICANN (Internet Corporation for Assigned Names and Numbers).
Sono una funzione internet importantissima. Contengono tutte le informazioni relative ai top-level domain (TLD) servers o i DNS servers che possono essere contattati.

top-level domain (TLD) DNS servers: gestiscono i domini di primo livello (es. .com, .org, .net, .it, .fr, .uk, .de, ...).
Due casi:

- organizzazioni grandi gestiscono e mantengono in modo diretto i propri server autoritativi (es.: Bicocca)
- organizzazioni piccole si appoggiano a provider di servizi DNS

authoritative DNS servers: organizzazioni che gestiscono i propri DNS (es. Bicocca).
Sono quelli che vanno a fare il mapping vero e proprio.

local DNS name servers: ISP, università, aziende, home network che forniscono accesso a internet ai client.

Local DNS name servers

Quando gli host fanno query DNS, queste vengono inoltrate a un **local DNS server**.

- i DNS locali, se la cache è vuota, creano l'associazione nome-indirizzo che viene storata e conservata per un limitato intervallo di tempo
- altrimenti (se non trova riscontro) interrogano la gerarchia e rispondono direttamente alle query che hanno già in cache, magari non con valori aggiornati ma che comunque non si discostano tanto ma che ne so senti la registrazione

Come funziona la risoluzione di DNS

iterativa: quella che vedremo
Il ruolo centrale ce l'ha il DNS locale che va a contattare gli altri della gerarchia e scopre quale server successivo da contattare per ottenere il vero indirizzo IP da cui ottenere l'informazione richiesta. Tutto trasparente per il server ma non per il requesting host.
Meglio quella iterativa perché il DNS locale gestisce meglio.

ricorsiva: quella che non vedremo (no shit, CoP)

Si richiede più carico ai DNS all'interno della gerarchia e si toglie un po' di carico a livello del DNS locale.

Non è l'ideale perché vado ad aggiungere complessità ai messaggi che devono essere scambiati fra questi nodi e vista la grande richiesta di scambio di messaggi sulla rete è un casino.

Capitolo 5

Strato di trasporto

5.1 Servizi e protocolli di trasporto

Forniscono una comunicazione logica tra **processi applicativi** che risiedono su host diversi.

Cos'è una comunicazione logica? Fa sì che i processi che stanno comunicando lo facciano, si scambino cioè dati, come se fossero sullo stesso host (in realtà non lo sono). Di fatto *trasparente al livello superiore*. Come il livello applicativo, interpretato esplusivamente dagli host (utenti periferici), non dai dispositivi rete (switch, router, etc.).

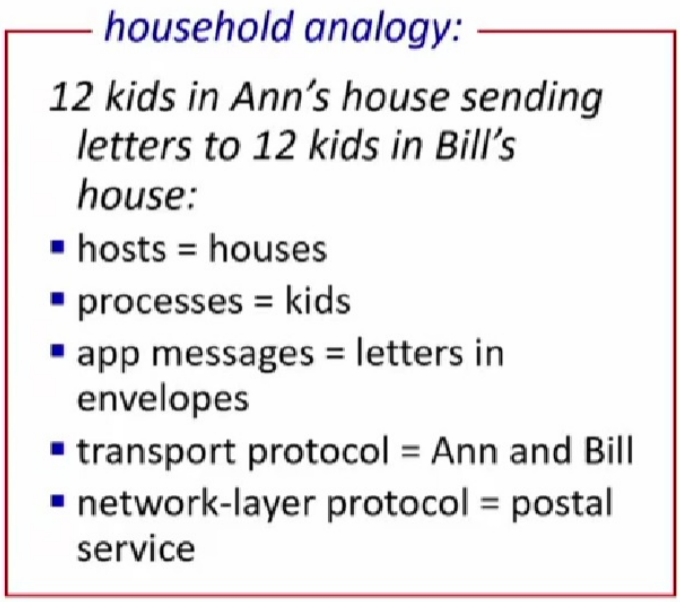
Come si comporta: spezzetta l'informazione in segmenti che trasporta poi a livello di rete.

Due tipi che vedremo: i due principali protocolli a livello di trasporto che sono disponibili per le applicazioni internet sono:

TCP: Transmission Control Protocol
Funzionalità aggiuntiva: *affidabilità*. I messaggi vengono inviati e se arrivano bene, se non arrivano vengono ritrasmessi.

UDP: User Datagram Protocol
Best-effort: I messaggi vengono inviati, se arrivano bene, se non arrivano va bene lo stesso.

Es. pratico:



5.1.1 Differenze livello di trasporto e livello di rete

l. di rete: comunicazione logica fra host

l. di trasporto: comunicazione logica fra processi

5.1.2 Socket

Quando si parla di protocolli a livello di trasporto si deve introdurre il concetto di ***socket***. Quando un processo deve comunicare con un altro processo, deve avere un'interfaccia di comunicazione. Questa interfaccia è il socket.

DEFINIZIONE

Una socket è un'interfaccia verso cui vengono spinti e ricevuti i messaggi che devono essere ricevuti da/inviati a un processo.

Permette di mettere in comunicazione i processi eseguiti a livello applicativo e il livello di trasporto.
N.B.: un processo può gestire più di una socket contemporaneamente.
Avremo bisogno di

- 1 socket per **sender**
- 1 socket per **receiver**

La pila protocollare da livello fisico a livello di trasporto è gestita dal *sistema operativo*. Lo sviluppatore può programmare quella parte di socket esposta verso il processo applicativo.

Per ricapitolare, il messaggio farà: `livello applicativo` → `socket` → `livello di trasporto` → `livello di rete` → `livello fisico` → `livello fisico` → `livello di rete` → `livello di trasporto` → `socket` → `livello applicativo`.

5.2 I due principali protolli Internet a livello di trasporto

5.2.1 TCP, Transmission Control Protocol

Si assicura che tutti i *segmenti* di informazione vengano consegnati *correttamente* e *in ordine*.
Garantisce la consegna affidabile e la connessione affidabile fra mittente e destinatario.
Offre altri servizi:

controllo della congestione: per evitare di contribuire alla congestione della rete fa capire al mittente di diminuire il numero di segmenti inviati.

controllo di flusso: non riguarda la rete ma il destinatario. Se il destinatario non è in grado di ricevere i segmenti, il mittente non li invia.

è connection-oriented: chiede che vengano istituite connessioni fra le parti prima di poter inviare i dati.

È molto importante per le *applicazioni elastiche*, che non sono in alcun modo tolleranti per quanto riguarda la perdita di dati ma molto più tolleranti per quanto riguarda i ritardi. TCP è perfetta.
Nessuno di questi due protocolli garantisce:

- un limite sui ritardi
- garanzie sulla banda (es. non posso dire "ho almeno 10 Mbit di banda per questa connessione").

TCP, principi della trasmissione affidabile dei dati

Cosa ci serve? Un canale di comunicazione affidabile. Ma nella realtà affidabile non lo è mai. A livello di trasporto bisogna implementare una tecnologia per rendere affidabile la comunicazione.
Servono servizi di feedback, il destinatario deve dirmi se gli sono arrivati i dati. Ho perso cose.

5.2.2 Reliable data transfer

Raffigurato c'è un diagramma da spiegare (inserisci qua screen). A sinistra ho l'asse dei tempi, a sinistra avrò `A` e a destra `B` e `A` comunica con `B`. Da `A` a `B` ho frecce che vanno in diagonale perché? Senti.
Consideriamo un modello di rete ideale, ovvero:

- non ci sono errori di bit
- perdite di segmenti/pacchetti
- ritardi di trasmissione dei segmenti/pacchetti

Così diventa superfluo il modello di reliable data transfer perché la rete è già ottimale e affidabile, nessun bisogno di meccanismi di feedback etc.
Non è così ovviamente in verità, quindi bisogna implementare un reliable data transfer.
Avendo errori, una cosa che posso fare è introdurre *due messaggi*:

positive acknowledgment: `ACK` (messaggio di conferma positiva)

negative acknowledgment `NACK` (messaggio di conferma negativa)

```
IF ack,
THEN M_(i+1)
ELSE IF nack
    THEN M_i
    ELSE ?
```

Non funziona, perché anche `ACK` e `NACK` possono essere soggetti ad errore.
Se i miei segmenti *fossero numerati*, riesco a capire se ho perso qualcosa e anche nel caso di duplicati.

```
IF ack
THEN M_(i+1)
ELSE M_i
```

Se vado a numerare anche gli *acknowledgments*, riesco anche ad evitare l'uso di `NACK`.
Nel momento in cu ricevo un ACK_i per il segmento `M_i`, so che tutti i segmenti precedenti sono stati ricevuti correttamente. Se ricevo un ACK_i per il segmento `M_i`, ovvero mando l'`ack` del segmento precedente, posso capire che il segmento `M_i` è stato perso.

```
IF ack
THEN M_(i+1)
ELSE M_i
```

Le reti però non introducono solo perdite di segmenti, ma anche ritardi.
La perdita può capitare al messaggio che sto inviando o all'ack che sto ricevendo.
Come gestisco situazioni in cui li perdo entrambi? Devo introdurre un *timer*.
Parliamo di protocolli ***stop-and-wait***.

Come settiamo questo timer? È una cosa complicata ma fondamentale. Se è troppo alto, se mando un time out spreco un sacco di tempo. Se è troppo breve, rischio di creare ritrasmissioni inutili, invece di usare il mio canale per comunicare normalmente lo riempio di messaggi inutili. Nelle slide un esempio di ciò, timer troppo breve.

Misurare la performance

Transmitting a segment at a time and waiting for its ack before a further transmission (stopand-wait) significantly limits performance.
Example: $RTT = 100ms, L = 1kbyte, R = 100Mbit/s \rightarrow U = 0,008$.

Sliding windows protocols

Sliding window protocols can transit up to W segments while waiting the ack of the first one Ci sarebbe da rivedere l'inizio di sta lezione.
Qua ho aumentando molto l'efficienza dei protocolli stop-and-wait. Se vado a dimensionare la finestra, posso avere un numero di segmenti che posso inviare prima di ricevere l'ack, quindi molta informazione utile.
Perché la trasmissione sia continua (ed evitare gli spazi vuoti) il tempo di invio del primo segmento + il tempo di ricezione dell'ack del primo segmento deve essere minore del tempo di trasmissione di W segmenti. Ovvero:

$$w \bullet \frac{L}{R} \geq RTT + \frac{L}{R}$$

$$W \geq \frac{RTT \bullet R}{L} + 1$$

Go-back-N protocol

Cosa ritrasmetto in caso di errore?

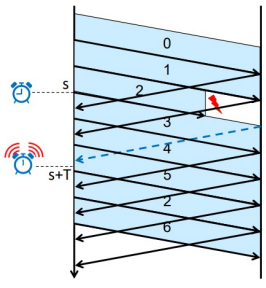
Go-back-N: ritrasmetto tutti i segmenti a partire da quello che ha avuto problemi.
Ma come funziona il Go-back-N?

Lato ricevitore: se ho problemi al segmento 2 (come da immagine), il segmento 3 lo scarto; poi dal segmento 4 mando ack negativi per dire che ho perso qualcosa prima e scarta i segmenti che riceve. Nell'immagine infatti vedo che dopo aver scartato il 5, inizia ad accettare il 2 quando viene ritrasmesso.

Lato mittente: quando riceve un ack negativo, ritrasmette tutti i segmenti a partire da quello che ha avuto problemi. Comunque riguarda la registrazione qua.

Non c'è bisogno di implementare un buffering al ricevitore.
Si dice che gli *ack* sono *cumulativi*. Nel momento in cui non ho un segmento mando un ack per l'ultimo segmento ricevuto e uno negativo per tutti i segmenti che non ho ricevuto.
Per come è fatto il protocollo, ho un solo timer, ogni volta che viene ricevuto correttamente un ack viene resettato e riportato a 0. Questo semplifica di molto l'implementazione.

Selective repeat: ritrasmetto solo il segmento che ha avuto problemi.



Gli ack **non** sono cumulativi. Quando scade il

timer, ritrasmetto solo il segmento che ha portato al timeout, ovvero alla scadenza di quel timer.

5.2.3 TCP protocol

Overview

point-to-point: un mittente, un destinatario

affidabile, byte-stream in ordine: garantisce che i dati arrivino correttamente e in ordine, non ci sono "message boundaries"

full duplex data: flow di dati in entrambe le direzioni nella stessa connessione (MSS: maximum segment size)

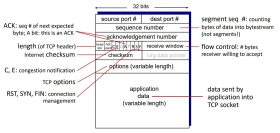
***acks* cumulativi:**

pipelining: TCP congestion and flow control set window size

connection-oriented: handshaking (exchange of control messages) initializes sender, receiver state before data exchange

flow controlled: sender will not overwhelm receiver

TCP, struttura di un segmento



source port #: + dest. port #: 16 bit + 16 bit

sequence number: 32 bit, numero di sequenza del primo byte di dati nel bytestream

acknowledgment number: 32 bit, numero di sequenza del prossimo byte di dati che il mittente si aspetta di ricevere
Sotto, non ho capito il nome, a sinistra del receive window. Noi non la vedremo, più opzioni con lunghezza variabile, dovrebbe contenere l'*header length* di 4 bit, lunghezza dell'header in parole da 32 bit, poi RST, SYN, FIN che si occupando di connection management

receive window: flow control, # bytes che il ricevente è disposto ad accettare

checksum: NOI NON VEDREMO, 16 bit, controllo degli errori

urgent pointer: NOI NON VEDREMO, 16 bit, se il flag URG è impostato, indica il byte successivo al byte urgente

data: NOI NON VEDREMO, 0 o più byte di dati

TCP numeri di sequenza e ACKs

Sequence numbers: byte stream "number" of first byte in segment's data

Acknowledgements: seq # of next byte expected from other side, cumulative ACK

Q: how receiver handles out-oforder segments?

A: TCP spec doesn't say, - up to implementor

TCP, round trip time e timeout

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

Il senso di tutto ciò è che RTT non è fisso ma bisogna cercare di settare un timeout sufficiente a permettere al pacchetto di arrivare e al ricevente di rispondere.

Q: how to estimate RTT?

- SampleRTT:measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT "smoother"

– average several recent measurements (EstimatedRTT),not just current SampleRTT (quindi una media degli ultimi RTT misurati)

Come lo misuro?

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

senti la lezione per capire l'equazione.
Per il timeout, aggiungo un margine di sicurezza, tempo aggiuntivo per evitare la situazione in cui metto un timeout troppo breve e ritrasmetto inutilmente.

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

Dove DevRTT è il mio margine di sicurezza (perché 4* non lo sa nessuno, neanche il prof, credo non lo dicano proprio le specifiche).
L'intervallo varia anche in base al valore del RTT stimato. N.B.: quando apri una connessione e sei appena partito, non hai nessun RTT stimato, quindi di solito si dà il SampleRTT come valore all'EstimatedRTT.

TCP Sender (simplified)

data received from application create segment with seq # seq # is byte-stream number of first data byte in segment

start timer
- think of timer as for oldest unACKed segment
- expiration interval: TimeoutInterval

timeout
- retransmit segment that caused timeout
- restart timer

ACK received: if ACK acknowledges previously unACKed segments

- update what is known to be ACKed
- start timer if there are still unACKed segments

TCP: retransmission scenarios

Me lo sono persa

TCP: fast retransmit

Me lo sono persa

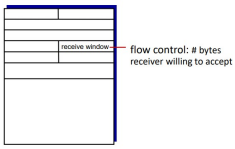
DEFINIZIONE

If sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq # (likely that unACKed segment lost, so don't wait for timeout).

Smallest seq # perché può capitare una perdita multipla.
importantissimo!!! Unico caso in cui non aspetto il timeout per ritrasmettere.

TCP controllo di flusso

Abbiamo detto che avviene nella parte del segmento TCP "receiver window", che si occupa di flow control (# bytes che il ricevente è in grado di accettare).



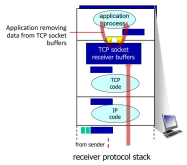
DEFINIZIONE

Flow control: receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast.

receive buffer: fixed amount of buffer space, allocated for a TCP socket

receiver advertises free buffer space by including value of RcvWindow in segment

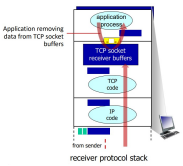
sender limits unACKed data to receiver's RcvWindow



- TCP receiver "advertises" free buffer space in *rwnd* (receive window) field in TCP header (many operating systems autoadjust RcvBuffer)
- sender limits amount of unACKed ("in-flight") data to received *rwnd*
- guarantees receive buffer will not overflow

TCP connection management

before exchanging data, sender/receiver "handshake":



- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)

risorse (tipo variabili al centro dell'immagine) da allocare prima dell'handshake.

TCP 3-way handshake, per aprire una connessione

TCP chiudere una connessione

- client, server each close their side of connection
- send TCP segment with FINbit = 1
- respond to received FIN with ACK
- on receiving FIN, ACK can be combined with own FIN

TCP: controllo congestione

DEFINIZIONE

Congestion: informally, "*too many sources sending too much data too fast for network to handle*".

- manifestations: • long delays (queueing in router buffers) • packet loss (buffer overflow at routers) - different from flow control! NB: controllo di flusso esclusivo del ricevente, controllo di congestione è un problema di tutti i nodi della rete. When network gets closer to saturation of available resources, delay and loss percentage increase - If the transport layer retransmits messages, the average number of messages retransmissions increases too - While throughput (i.e., messages traversing the network) is close to 100% of capacity, "goodput" experienced by the application decreases! Il goodput sono le informazioni che a me effettivamente interessano e droppa così nel grafico perché gran parte delle informazioni che io ricevo sono ritrasmissioni a me non utili.

Approaches towards congestion control

Non andremo a vederla più di tanto e comunque è vagamente fatto da quale parte del segmento TCP?
End-to-end congestion control: - no explicit feedback from network - congestion inferred from observed loss, delay Approaches towards congestion control data data ACKs ACKs - approach taken by TCP

Network-assisted congestion control: - routers provide direct feedback to sending/receiving hosts with flows passing through congested router - may indicate congestion level or explicitly set sending rate

Usa un approccio *Additive Increase Multiplicative Decrease*: senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event.

Additive Increase
- increase sending rate by 1
- maximum segment size (MSS)
- every RTT until loss detected

Multiplicative Decrease • cut sending rate in half at each loss event

Parliamo di *comportamento a dente di sega* di AIDM, vedi il grafico.
TCP sending behavior:

- roughly: send `mincwnd,rwnd` bytes, wait RTT for ACKS, then send more bytes
- When `cwnd < rwnd`, congestion control is dominant w.r.t. flow control
- TCP sender limits transmission: $LastByteSent - LastByteAcked < mincwnd, rwnd$
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)
- `rwnd` is instead adjusted according to the received value by the receiver (implementing TCP flow control)

Tre fasi:

slow start: quando inizia la connessione, il rate di invio viene aumentato in maniera esponenziale. Nome un po' infelice (per citare il prof) visto che è tutto tranne che slow.

congestion avoindance: **Q:** when should the exponential increase switch to linear?

A: when cwnd gets to 1/2 of its value before last timeout

Implementation: • variable **ssthresh**

- on loss event, **ssthresh** is set to 1/2 of the value of **cwnd** just before loss event

TCP Reno: it adopts the Fast Recovery mechanism after three duplicates acks

- set `cwnd` to `ssthresh + 3 MSS` (congestion avoidance)
- same as TCP Tahoe when timeout is reached

5.2.4 UDP, User Datagram Protocol

Non offre nessuno dei servizi di TCP. È un protocollo *best-effort*.
Decisamente meno affidabile di TCP per quanto riguarda perdita di dati e ordine di invio e ricezione.
Meglio per le *applicazioni interattive*, che sono tolleranti per quanto riguarda la perdita di dati ma non tolleranti per quanto riguarda i ritardi. UDP è perfetta.
Protocollo tipico: connectionless. Ogni segmento inoltre gestito in modo indipendente dagli altri. Non è così in TCP.
Ha dei vantaggi:

- non c'è necessità di stabilire una connessione (cosa che introduce ritardi)
- è più semplice (no controllo stato sender/receiver)
- header più piccolo
- non c'è controllo di flusso e di congestione

Nessuno di questi due protocolli garantisce:

- un limite sui ritardi
- garanzie sulla banda (es. non posso dire "ho almeno 10 Mbit di banda per questa connessione").

UDP segment header

Senti lezione per questo pezzetto. E metti screen.

source port #: 16 bit?

dest. port #: 16 bit?

length: in bytes, lunghezza totale del segmento (header + dati).
16 bit?

checksum: controllo degli errori, wth is he saying, complemento a 1?!
16 bit?

5.2.5 Multiplazione e demultiplazione

Entrambi i protocolli hanno questi (?) servizi (?):

Multiplazione: l'operazione di multiplexing permette di gestire l'invio di più flussi di dati contemporaneamente verso più socket diverse, andando a inserire valori specifici negli header di trasporto. Operazione fondamentale per la *demultiplazione*.

Demultiplazione: comunicazione nella direzione opposta: nello stack protocollare vedo arrivare dati provenienti da processi (veramente socket) differenti. Il receiver ha il compito di andare a vedere (una volta tolti gli header) a quale protocollo mandare i dati.
Usa **indirizzi IP** (livello di *rete*) e **numero di porte** (livello di *trasporto*) per indirizzare i dati alle socket corrette.
L'host riceve l'IP da cui arriva il messaggio.

connectionless demultiplexing: UDP

- Quando creo la socket, devo assegnare un numero di porta #.
- Quando creo il datagram da inviare tramite UDP socket, devo specificare l'intirizzo IP del ricevente e il suo numero di porta #. Servono entrambe!! Questo perché:

  - due host diversi possono avere lo stesso numero di porta #.
  - un host può avere più socket aperte, ognuna con un numero di porta # diverso.
- Quando arriva il datagram, il livello di trasporto estrae l'indirizzo IP e il numero di porta # (li prende dall'header) e li usa per consegnare il datagram alla socket appropriata.

  Sulle slide c'è un esempio molto chiaro: inserisci screen.

connection-oriented demultiplexing: TCP

Socket identificata da 4-tuple:

- source IP address.
- source port number.
- dest IP address.
- dest port number.

Sulle slide c'è un esempio molto chiaro: inserisci screen.
Socket strettamente associata alla coppia IP-porta: nell'esempio comunicando da C a B, avrò bisogno di due socket diverse.

Capitolo 6

Sistemi operativi: struttura e servizi

Argomenti

- A che servono i sistemi operativi?
- Requisiti per i sistemi operativi
- Struttura e servizi dei sistemi operativi
- Chiamate di sistema ed API
- I programmi di sistema

6.1 I sistemi operativi

Un s.o. è una certa quantità di software che viene dato assieme all'hardware perché quest'ultimo da solo non funziona. Per esempio quando compri un telefonino apri e ti trovi Android o iOS, mentre su un laptop abbiamo Windows, macOS e Linux.
Parliamo del modello teorico della macchina di Von Neumann. Questo modello si basa su cinque componenti fondamentali:

- Unità centrale di elaborazione (**CPU**), che si divide a sua volta in unità aritmetica e logica (ALU o unità di calcolo) e unità di controllo;
- Unità di **memoria**, intesa come memoria di lavoro o memoria principale (RAM, Random Access Memory);
- Unità di **input**, tramite la quale i dati vengono inseriti nel calcolatore per essere elaborati;
- Unità di **output**, necessaria affinché i dati elaborati possano essere restituiti all'operatore;
- **Bus**, un canale che collega tutti i componenti fra loro.

Un computer quando lo accendiamo inizia ad eseguire un programma. Se non c'è un programma da eseguire è solo un mucchio di ferraglia che si blocca e non fa niente. Perciò possiamo dire che un s.o. è il **primo** programma che viene eseguito all'accensione del computer e che permette di eseguire altri programmi.
È una cosa molto diversa dalle *applicazioni*, che sono quelle che "fanno le cose utili" cit prof. Infatti la prima cosa che facciamo quando per esempio compriamo un telefono è installare WhatsApp o altre app che ci servono e rendono comomda la vita. Un s.o. di per sé non è *"utile"*, lo diventa in relazione al funzionamento delle app che ci interessano.
Un s.o. fornisce un ambiente a finestre (laptop) o icone (smartphone) che permette di eseguire le applicazioni utili: le possiamo installare, lanciare, far eseguire (anche più di una contemporaneamente), interromperne l'esecuzione...
La macchina di Von Neumann esegue un programma alla volta, ma noi di solito su un computer eseguiamo più di un'applicazione alla volta (es. WebEx per la lezione e VSCode per gli appunti). Questa cosa ci è permessa dal s.o., che ci permette ci gestire n applicazioni contemporaneamente, anche più dei processori di cui dispone il mio computer. Es.: mettiamo di avere un computer con un processore a 32 core, io però posso eseguire anche centinaia di programmi contemporaneamente, non massimo 32.
Il s.o. crea un'ambiente in cui le applicazioni possono collaborare assieme. Se avessi più applicazioni che usano contemporaneamente lo stesso hardware (es. lo schermo)? Il s.o. si occupa di gestire le risorse hardware e di farle usare alle applicazioni. Il s.o. è il software *intermediario* tra le applicazioni e l'hardware.
Un'altra cosa che fa il s.o. è organizzare i nostri file in un sistema ordinato di file e cartelle (anche memorizzandoli su dispositivi secondari di memoria e storaging). Il s.o. si occupa di gestire i file e le cartelle, di crearli, cancellarli, rinominarli, spostarli...

Cos'è un sistema operativo?

DEFINIZIONE

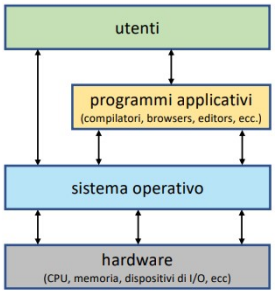
È un insieme di programmi (software) che gestiscono gli elementi fisici di un computer (hardware).

Fornisce una piattaforma di sviluppo per le applicazioni, che permette loro di condividere ed astrarre le risorse hardware.
Agisce da intermediario tra utenti e computer, permettendo agli utenti di controllare l'esecuzione dei programmi applicativi e l'assegnazione delle risorse hardware ad essi.
Protegge le risorse degli utenti (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali attori esterni.
Un s.o. in primo luogo è una piattaforma di sviluppo, ossia **un insieme di funzionalità software** che i programmi applicativi possono usare.
Tali funzionalità permettono ai programmi di poter usare in maniera conveniente le risorse hardware, e di condividerle:

- Da un lato, il s.o. **astrae** le risorse hardware, presentando agli sviluppatori dei programmi applicativi una visione delle risorse hardware più facile da usare e più potente rispetto alle risorse hardware ≪native≫.

- Dall'altro, il s.o. **condivide** le risorse hardware tra molti programmi contemporaneamente in esecuzione, suddividendole tra i programmi in maniera equa ed efficiente e controllando che questi le usino correttamente.

Componenti di un sistema di elaborazione



Utenti: persone, macchine, altri computer...
Programmi applicativi: risolvono i problemi di calcolo degli utenti.
s.o.: coordina e controlla l'uso delle risorse hardware.
Hardware: risorse di calcolo (CPU, periferiche, memorie di massa...).

6.1.1 Requisiti per i sistemi operativi

Cosa si richiede ad un s.o.?

Oggigiorno i computer sono ovunque: vi sono molteplici tipologie di computer utilizzati in scenari applicativi diversi (i nostri laptop, i nostri smartphones, i computer detti "embedded" che non hanno lo scopo di interagire con persone ma sono "cyber-fisici", cioè che fanno parte di servizi ad es. quelli che controllano le automobili ad es. il sistema ABS che fa in modo che la ruota non si blocchi e quindi non slitti quando noi inchiodiamo e freniamo a fondo, etc...).
In quasi tutti i tipi di computer si tende ad installare un s.o. allo scopo di gestire l'hardware e semplificare la programmazione.
Ma ogni scenario applicativo in cui viene usato un computer richiede che il s.o. che vi viene installato abbia caratteristiche ben determinate (es. laptop molto diverso dai sistemi embedded). Che cosa si richiede ad un s.o. per supportare un certo scenario applicativo?
A seconda che sia:

Server, mainframe: massimizzare la performance, rendere equa la condivisione delle risorse tra molti utenti

Laptop, PC, tablet: massimizzare la facilità d'uso e la produttività della singola persona che lo usa

Dispositivi mobili: ottimizzare i consumi energetici e la connettività

Sistemi embedded: funzionare senza, o con minimo, intervento umano e reagire in tempo reale agli stimoli esterni (interrupt)

La maledizione della generalità

Nella storia (ed anche oggi) alcuni sistemi operativi sono stati utilizzati per scenari applicativi diversi,
Ad esempio, Linux è usato oggi nei server, nei computer desktop e nei dispositivi mobili (come parte di Android).
La **maledizione della generalità** afferma che, se un s.o. deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportare nessuno di tali scenari particolarmente bene. Praticamente io quando cerco di dare più di un esame per sessione.
Questo si è visto con OS/360, il primo s.o. che doveva supportare una famiglia di computer diversi (la linea 360 dell'IBM).
Quella **maledizione della generalità** non avviene sempre e necessariamente, è un potenziale rischio; può essere tuttavia aggirata, ma non ho capito come.

6.2 Struttura dei sistemi operativi

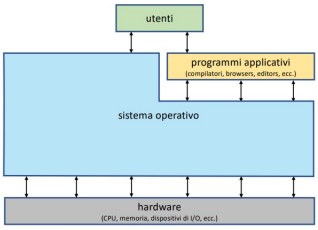
Non c'è una definizione universalmente accettata di quali programmi fanno parte di un s.o..
In generale però un s.o. comprende almeno:

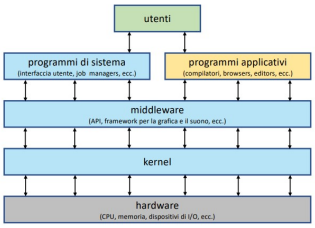
Kernel: il "programma sempre presente", che si "impadronisce" dell'hardware, lo gestisce, ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta.

Middleware: servizi di alto livello che astraggono ulteriormente i servizi del kernel e semplificano la programmazione di applicazioni (API, framework per grafica e per suono...)

Programmi di sistema: non sempre in esecuzione, offrono ulteriori funzionalità di supporto e di interazione utente con il sistema (gestione di jobs e processi, interfaccia utente...)

Alcuni sistemi operativi forniscono anche dei programmi applicativi (editor, word processor, fogli di calcolo...), ma non li considereremo parti del s.o. stesso.





6.3 Servizi dei sistemi operativi

Un s.o. offre un certo numero di **servizi**:

Per i programmi applicativi: perché possano eseguire sul sistema di elaborazione usando le risorse astratte esposte dal s.o..

Per gli utenti: per gestire l'esecuzione dei programmi e stabilire a quali risorse hardware i programmi (e gli altri utenti) hanno diritto.

Per garantire che il sistema di elaborazione funzioni in maniera efficiente.

Gli utenti però interagiscono con il s.o. attraverso i programmi di sistema...
...i quali utilizzano gli stessi servizi dei programmi applicativi...
Quindi, in definitiva, il s.o. ha bisogno di esporre i suoi servizi esclusivamente ai programmi (applicativi o di sistema).

6.3.1 Principali servizi:

Controllo processi: questi servizi permettono di caricare in memoria un programma, eseguirlo, identificare la sua terminazione e registrarne la condizione di terminazione (normale o erronea).

Gestione file: questi servizi permettono di leggere, scrivere, e manipolare files e directory

Gestione dispositivi: questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale.

Comunicazione tra processi: i programmi in esecuzione possono collaborare tra di loro scambiandosi informazioni: questi servizi permettono ai programmi in esecuzione di comunicare.

Protezione e sicurezza: permette ai proprietari delle informazioni in un sistema multiutente o in rete di controllarne l'uso da parte di altri utenti e di difendere il sistema dagli accessi illegali.

Allocazione delle risorse: alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente.

Rilevamento errori: gli errori possono avvenire nell'hardware o nel software (es. divisione per zero); quando avvengono il s.o. deve intraprendere opportune azioni (recupero, terminazione del programma o segnalazione della condizione di errore al programma).

Logging: mantiene traccia di quali programmi usano quali risorse, allo scopo di contabilizzarle, ovvero fare sì che un programma o un utente non usi troppe risorse sottraendole ad altri programmi o utenti.

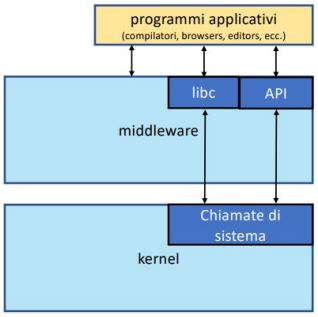
6.4 Chiamate di sistema ed API

6.4.1 Cosa sono?

kernel: programma sempre presente che è la parte centrale del s.o. e che permette di astrarre l'hardware e di fornire servizi ai programmi utente.

Middleware: fornisce le API, le chiamate di procedura che ci permettono di costruire i programmi.

Il kernel fornisce un insieme di servizi detti **chiamate di sistema**, funzioni invocabili in un determinato linguaggio di programmazione (ad es. C, C++...), anche se si tende a **non** invocare direttamente le chiamate di sistema. Ma si tende a frapporre uno strato (il *middleware*) tra il kernel e i programmi applicativi, in modo che questi ultimi chiamino le **API**, funzioni pensate appositamente per permettere ai programmi applicativi di usare i servizi offerti dal kernel in modo migliore rispetto alle chiamate di sistema.
Utili anche perché ogni linguaggio di programmazione ha una sua libreria, suoi oggetti...
Per evitare commistioni (mescolanze, contaminazioni...), si vanno a creare due strati distinti, uno per la libreria del L.d.P. e quello delle API.



Come visibile nell'immagine:

- Strato della libreria del L.d.P. può accedere alle chiamate di sistema.
- Strato della libreria delle API può accedere alle chiamate di sistema.
- Programmi applicativi possono accedere al middleware.
- Programmi applicativi possono accedere alla libreria del L.d.P..
- Programmi applicativi possono accedere alle API.

E basta.

6.4.2 Differenze fra chiamate di sistema ed API

API	Chiamate di sistema
Esposte dal **middleware**	Esposte dal **kernel**
Usano le c.d.s. nella loro implementazione	
Sono standardizzate (es. POSIX e Win32)	Non sono standardizzate (ogni kernel ha le sue)
Sono stabili	Possono variare al variare della versione del s.o.
Funzionalità più ad **alto livello** e più semplici da usare	Funzionalità più **elementari** e più complesse da usare

6.4.3 I programmi di sistema

I programmi di sistema sono programmi che usano le chiamate di sistema per offrire servizi di alto livello agli utenti. I più importanti:

Interfaccia utente (UI): permette agli utenti di interagire con il sistema stesso; può essere grafica (GUI) o a riga di comando(CLI). I sistemi mobili hanno un'interfaccia touch.

UI - l'interprete dei comandi:

Permette agli utenti di interagire con il sistema tramite **istruzioni**, comandi testuali.

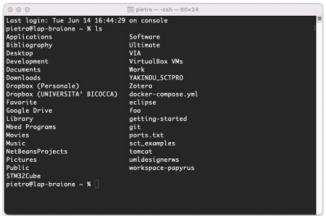
In molti sistemi operativi è possibile configurare quale interprete dei comandi usare, in quel caso è detto **shell**.

Ci sono due modi per implementare un comando:

▷ *Built-in:* l'interprete esegue direttamente il comando (tipico nell'interprete dei comandi di Windows).

▷ *Come programma di sistema:* l'interprete manda in esecuzione il programma (tipico delle shell Unix e Unix-like).

Spesso riconosce un vero e proprio l.d.p. con variabili, condizionali, cicli...

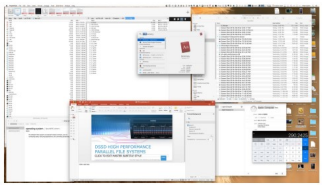


UI - le interfacce grafiche:

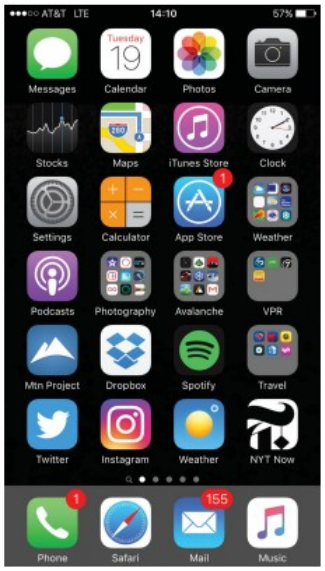
L'interfaccia grafica (GUI) è di solito basata sulla metafora della scrivania, delle icone e delle cartelle (le directory).

Nate dalla ricerca presso lo Xerox PARC lab negli anni '70, sono state popolarizzate poi da Apple con il Macintosh negli anni '80.

Su Linux le più popolari sono GNOME e KDE.



UI - le interfacce touch:
I dispositivi mobili richiedono interfacce di nuovo tipo.
Nessun dispositivo di puntamento (mouse)
Uso dei gesti (gestures)
Tastiere virtuali
Comandi vocali



Gestione file: creazione, modifica e cancellazione file e directory.

Modifica dei file: editor di testo, programmi per la manipolazione del contenuto dei file.
Il confine fra *programmi di sistema* e *programmi applicativi* è spesso labile.

Visualizzazione e modifica informazioni di stato: data, ora, memoria disponibile, processi, utenti. . . fino a informazioni complesse su prestazioni, accessi al sistema e debug. Alcuni sistemi implementano un registry, ovvero un database delle informazioni di configurazione.

Caricamento ed esecuzione dei programmi: loader assoluti e rilocabili, linker, debugger.

Ambienti di supporto alla programmazione: compilatori, assemblatori, debugger, interpreti per diversi linguaggi di programmazione.

Comunicazione: forniscono i meccanismi per creare connessioni tra utenti, programmi e sistemi; permettono di inviare messaggi agli schermi di un altro utente, di navigare il web, di inviare messaggi di posta elettronica, di accedere remotamente ad un altro computer, di trasferire file...

Servizi in background: lanciati all'avvio, alcuni terminano, altri continuano l'esecuzione fino allo shutdown. Forniscono servizi quali verifica dello stato dei dischi, scheduling di jobs, logging...
I servizi in background sono ad esempio i cosiddetti *demoni di sistema* nei sistemi Unix-like, sono programmi di sistema che di solito sono inattivi per la maggior parte del tempo ma attivati all'avvio del sistema e che rimangono in esecuzione fino allo spegnimento del sistema. Rimangono in attesa finché non si verifica un certo evento, compiono una certa operazione e si disattivano di nuovo in attesa del prossimo evento.
Utile per servizi temporali, ad esempio utilissimi per i backup, tipo quelli mensili.

L'implementazione dei programmi di sistema

I programmi di sistema sono implementati utilizzando le API, esattamente come i programmi applicativi.
Consideriamo ad esempio il comando cp delle shell dei sistemi operativi Unix-like:

- cp (copy a file) in.txt out.txt
- Copia il contenuto del file in.txt in un file out.txt
- Se il file out.txt esiste, il contenuto precedente viene cancellato, altrimenti out.txt viene creato

È implementato come programma di sistema tramite API (tutte POSIX).
Una possibile struttura del codice è riportata sulla destra (le invocazione delle API sono riportate in grassetto).

- **Apri** in.txt in lettura
- Se non esiste
  - **Scrivi** un messaggio di errore su termimale
  - **Termina** il programma con codice errore
- **Apri** out.txt in scrittura
- Se non esiste, **crea** out.txt
- Loop
  - **Leggi** da in.txt
  - **Scrivi** su out.txt
- End loop
- **Chiudi** in.txt
- **Chiudi** out.txt
- **Termina** normalmente