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).

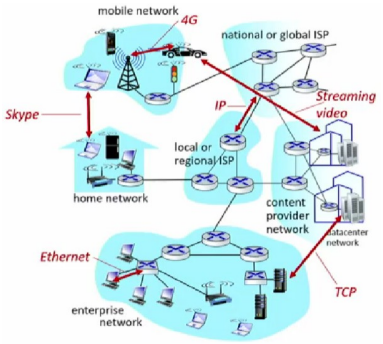
2.1.1 Visione nuts and bolts

hosts , sono dispositivi *end system* ovvero terminali.

packet switches o *commutatori di pacchetto*, che 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

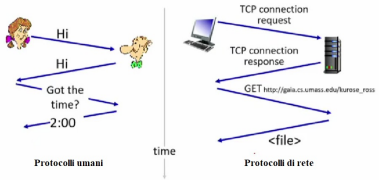
Protocolli umani
I protocolli umani sono le regole che seguono gli esseri umani quando comunicano tra loro.
Esempio: incontro una persona, scambio di convenevoli, poi chiedo: "Che ore sono?" e in base alla risposta ho delle reazioni diverse.
Altro esempio: "Ho una domanda" durante una lezione, così che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

Protocolli di rete
Emulano il funzionamento dei protocolli umani.
Tutte le comunicazioni di rete sono gestite da protocolli.

DEFINIZIONE

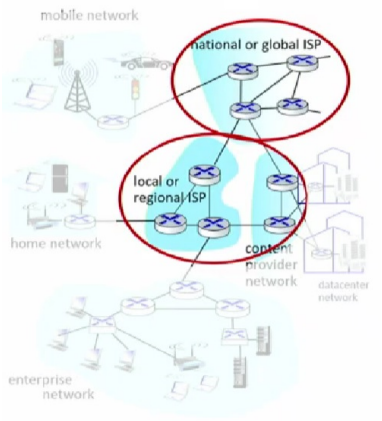
I protocolli di rete definiscono le regole per i messaggi inviati in rete. In particolare definiscono il formato dei messaggi, l'ordine in cui vengono inviati e ricevuti tra le entità di rete (host, commutatori, ...) e le azioni da intraprendere una volta che questi messaggi vengono inviati e ricevuti.

2.2.1 Cos'è un protocollo?



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.

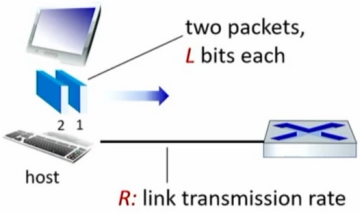
Ovviamente più è alta la banda, più è alta la velocità di trasmissione.

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

DEFINIZIONE

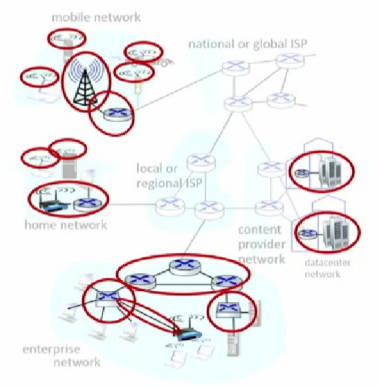
Il *ritardo di trasmissione* è il ritardo che intercorre tra l'invio del primo bit di un pacchetto di L bit alla ricezione dall'altro lato del link dell'ultimo bit dello stesso pacchetto.

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



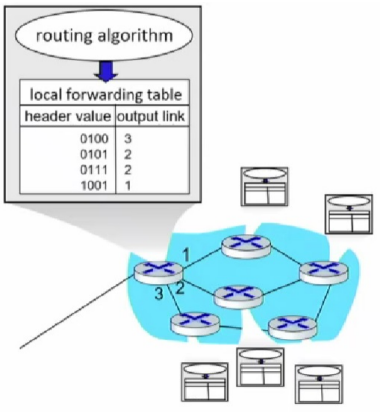
2.3.2 Access networks

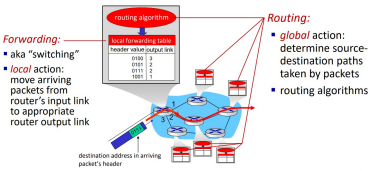
Abbiamo detto che Internet è un *packet switching network*, una rete a commutazione di pacchetto. L'unità informativa che viene scambiata in rete è il **pacchetto** che appunto viaggiano in rete, ricevuti e trasmessi da più dispositivi.
Il compito della rete è di inoltrare i pacchetti verso i router 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.

Swtinching: 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.

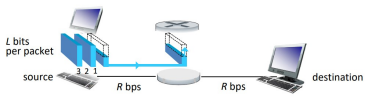
2.4 Modalità packet-switching

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 appartenga.

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.

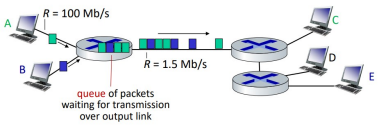
2.4.2 Esempio di esercizio

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 rinscontrabili 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**.

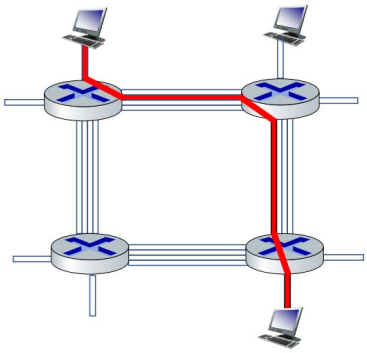


Nella maggior parte dei casi quello che può verificarsi è che il rate a cui io posso trasmettere in uscita sul mio link è più basso rispetto a quello con cui ricevo i pacchetti. Nell'immagine ho un R pari a $100\frac{Mb}{s}$ sui link in ingresso e un R pari a $1.5\frac{Mb}{s}$ sui link in uscita, *molto* 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.

2.5 Packet-switching vs circuit-switching

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

2.5.1 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.

Problemi

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:

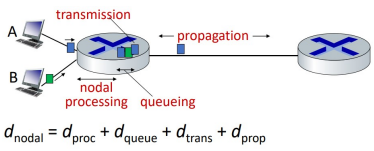
- **regional ISP:** collegamento fra **una** ISP e diverse reti di accesso a cui fornisce accesso.
- **Multi-homing:** collegamento fra **diverse** ISP e diverse reti di accesso a cui fornisce accesso, migliore perché nel caso cada la connessione con un ISP ci sono gli altri a mantenere attiva la connessione.

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 cambia da pacchetto a pacchetto.
Uno che non abbiamo visto è il **ritardo di elaborazione** ho perso roba vedi la registrazione.
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.
Tutte queste operazioni creano il **ritardo di elaborazione**, questo tempo è dell'ordine di grandezza di 10^{-9} secondi.
Un dato decisamente più importante (2-3 ordini di grandezza più grande) è il **ritardo di accodamento**.
Slide.
Poi abbiamo il **ritardo di propagamento** che dipende dal mezzo di trasmissione, lunghezza del cavo e velocità di propagazione.
A sommare tutte queste componenti, ottengo il tempo complessivo che il pacchetto ci mette a passare da sorgente a destinazione.
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: $d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$.

2.5.4 Di più sul ritardo di queueing

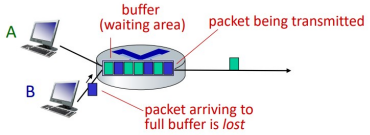
Il ritardo di queueing cambia da pacchetto a pacchetto.
Nel caso più generico possibile dipende dal tasso di intensità di traffico. Come si

calcola? 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 $\frac{L \bullet \bar{a}}{R}$.
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?



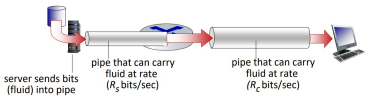
I pacchetti con errori di trasmissione vengono scartati. Potrebbe succedere che in qualche modo si è perso il pacchetto e decidere di ri-inviarlo ma non è scontato.

2.5.6 Throughput

velocità, rate (bits/time unit) a cui i bits sono inviati da sorgente a destinazione (a volte chiamato ***end-to-end throughput***) ed è:

istantaneo: rate ad un certo punto dato nel tempo

medio: rate in un periodo più esteso di tempo



Domande:

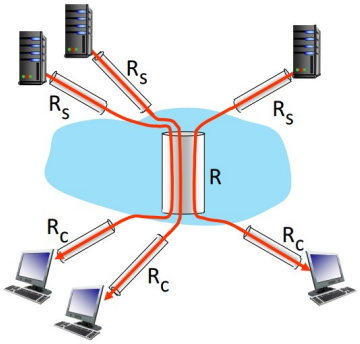
$R_s < R_c$**:** quale è il throughput medio?
R_s perché il rate di trasmissione è più basso del rate di ricezione.

$R_s > R_c$**:** quale è il throughput medio?
R_c perché il rate di trasmissione è più alto del rate di ricezione.

Parliamo di **textitbottleneck link**, link on end-end path that constrains end-end throughput.

Throughput: network scenario

Boh non ho sentito tanto sono un po' cotta cerca la registrazione.



Capitolo 3

Reti e livelli di rete

3.1 Architettura a strati

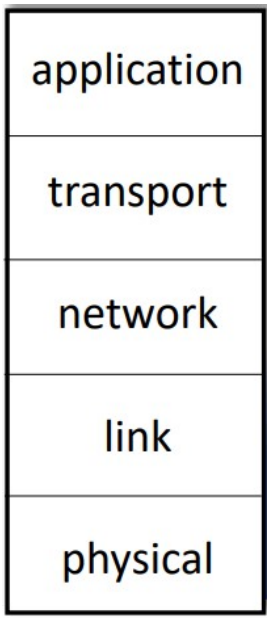
N.B: livelli e strati sono sinonimi qua.
Partiamo dal presupposto che le reti hanno una struttura complessa. Esiste una possibilità di studiare e organizzare questa struttura?
Un modo è quello di suddividere la complessa struttura in strati, ognuno dei quali si occupa di servizi diversi.

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'implementzione di una componente, questo risulta essere trasparente a tutto il sistema.

Architettura a strati dell'Internet

Anche nota come *Layered Internet protocol stack*.



applicativo: supporta applicazioni di rete.
Es.: HTTP, SMTP, FTP.

trasporto: trasferisce i dati tra protocollo e protocollo.
Es.: TCP, UDP.

rete: instrada i pacchetti da sorgente a destinazione.
Es.: IP, routing protocols.

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

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

3.1.1 Servizi, stratificazione, incapsulamento

Sorgente

I livelli **applicativi** si scambiano messaggi per implementare specifici servizi. Questi messaggi derivano dal livello di **trasporto**. Questi livelli di *trasporto* L'intestazione è Header, bit organizzati in **campi**, mi sono persa 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. Al livello di link ottengo un messaggio pieno di headers che a me non interessano ma sono utili per il funzionamento del collegamento in rete e che prende il nome di overhead.
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.

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.

A livello di commutatori

Negli host 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.
Il router toglie l'header del livello di link e inoltra il messaggio al prossimo router, incapsulandolo in un header diverso.
Quindi in due punti diversi del percorso il messaggio è incapsulato in due header diversi.

3.2 Livello applicativo

3.2.1 DNS: Domain Name System

Questo servizio permette di mappare un nome di dominio in un indirizzo IP.
Ci serve:

- host name
- indirizzo IP
  Tecnicamente associato all'interfaccia e non al server.

Struttura gerarchica dei DNS

root servers

Top-level domain e DNS autoritativi

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

Local DNS name servers

host fanno dns query, vengono inoltrate a un local DNS server.

- i DNS locali rispondono direttamente alle query che hanno già in cache ma che ne so senti la registrazione

Come funziona la risoluzione di DNS

iterativa: quella che vedremo

ricorsiva: quella che non vedremo (no shit)

In quella iterativa 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. In quella ricorsiva chi l'ha capito.
Ma è meglio quella iterativa perché il DNS locale gestisce meglio.

Capitolo 4

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

4.1 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...

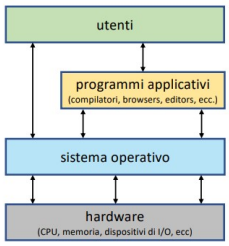
Cos'è un sistema opertativo?

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...).

4.1.1 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.

4.2 Struttura dei sistemi operativi

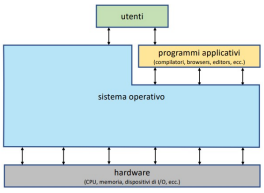
Non c'è una definizione universalmente accettata di quali programmi fanno parte di un s.o..
In generale però un s.o. comprende almeno:

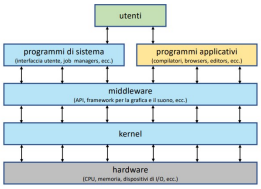
Kernel: il "programma sempre presente", che si "impadronisce" dell'hardware, lo gestisce, ed offre ai programmi i servizi per poterlo usare in maniera condivisa ed astratta.

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.





4.3 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).

4.3.1 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.

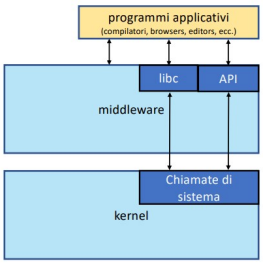
Allocazione delle risorse: alloca le risorse hardware (CPU, memoria, dispositivi di I/O) ai programmi in esecuzione in maniera equa ed efficiente.

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.

4.4 Chiamate di sistema e API

I programmi applicativi accedono a questi servizi che abbiamo appena elencato tramite chiamate di sistema o API.
Un programma applicativo che vuole accedere ai servizi offerti dal kernel non chiama direttamente dal kernel ma passa da un Middleware.
Il kernel offre i propri servizi ai programmi come chiamate di sistema, ossia di funzioni invocabili in un determinato linguaggio di programmazione (C, C++...).
I programmi però non utilizzano direttamente le chiamate di sistema, ma delle librerie di middleware dette Application Program Interface (API) implementate invocando le chiamate di sistema.
Questo perché le chiamate di sistema dipendono dal linguaggio mentre le API sono più standardizzate (es. le POSIX, tipicamente in C). Spesso le API sono fortemente legate con le librerie standard del linguaggio di implementazione (es. libc se le API sono implementate in C), al punto che anche queste diventano una parte implicita dell'API.

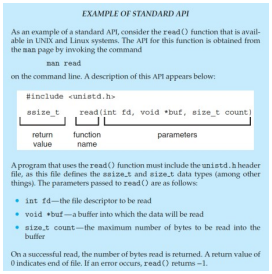


Differenza tra chiamate di sistema ed API

Le API sono esposte dal middleware, le chiamate di sistema dal kernel.
Le API usano le chiamate di sistema nella loro implementazione.
Le API sono standardizzate (esempio: standard POSIX e Win32), le chiamate di sistema no (ogni kernel ha le sue).
Le API sono stabili, le chiamate di sistema possono variare al variare della versione del s.o..
Le API offrono funzionalità più ad alto livello e più semplici da usare, le chiamate di sistema offrono funzionalità più elementari e più complesse da usare.

Esempio confronto API Win32 e POSIX

Esempio API POSIX



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t    read(int fd, void *buf, size_t count)
```

return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

4.5 I programmi di sistema

La maggior parte degli utenti utilizza i servizi del s.o. attraverso i programmi di sistema.
Questi permettono agli utenti di avere un ambiente più conveniente per l'esecuzione dei programmi, il loro sviluppo, e la gestione delle risorse del sistema.

4.5.1 Tipologie di programmi di sistema

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.

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

Modifica dei file: editor di testo, programmi per la manipolazione del contenuto dei file.

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

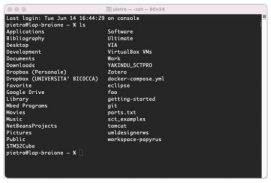
Interfaccia utente: l'interprete dei comandi

L'interprete dei comandi permette agli utenti di impartire in maniera testuale delle istruzioni al s.o..
In molti sistemi operativi è possibile configurare quale interprete dei comandi usare, nel qual caso è detto **shell**.
Due modi per implementare un comando:

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

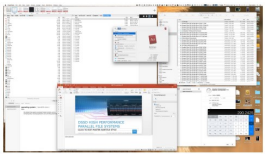
Come programma di sistema: l'interprete manda in esecuzione il programma (tipico delle shell Unix e Unix-like)

Spesso riconosce un vero e proprio linguaggio di programmazione con variabili, condizionali, cicli...



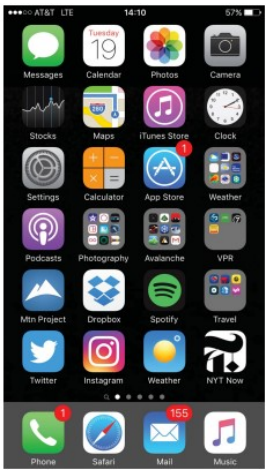
Interfaccia utente: le interfacce grafiche

L'interfaccia grafica (GUI) è di solito basata sulla metafora della scrivania, delle icone e delle cartelle (corrispondenti alle directory).
Nate dalla ricerca presso lo Xerox PARC lab negli anni 70, popolarizzate dal computer Apple Macintosh negli anni 80.
Su Linux le più popolari sono Gnome e KDE.



Interfaccia utente: le interfacce touch-screen

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



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 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.
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

Capitolo 5

Processi e Threads

Vedremo sostanzialmente come i s.o. gestiscono **processi** e **thread** e cosa si intende con questi due termini.

5.1 Argomenti

- Concetto di processo, operazioni e impplementazione
- Comunicazione interprocesso e comunicazione interprocesso in POSIX
- Multithreading, API e implementazione

5.2 Concetto di *processo*

5.2.1 Cos'è un processo

Un s.o. ha come obiettivo il fatto di prendere un sistema di elaborazione e fare in modo che su di esso possano essere eseguiti più programmi in modo concorrente. Il loro numero può essere arbitrariamente elevato, di solito molto maggiore del numero di processori, di CPU del sistema. Il s.o. ci permette però di eseguire il n° di programmi che vogliamo indipendentemente dal n° di CPU che abbiamo. Come?
Il s.o. realizza e mette a disposizione una macchina astratta, un'astrazione detta **processo**.

DEFINIZIONE

Un **processo** è un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma.

All'interno del s.o. il processo è l'unità esecutiva del programma. Es.: su un pc da 4 CPU, 50 programmi in esecuzione vuole dire 50 processi.
Supporremo **per ora** che l'esecuzione di un processo sia sequenziale e quindi non ci sia concorrenza.

5.2.2 Programmi e processi

Notare la differenza tra *programma* e *processo*!

- Un programma è un'entità passiva (un insieme di istruzioni, tipicamente contenuto in un file sorgente o eseguibile)

- Un processo è un'entità attiva (è un *esecutore di un programma*, o un *programma in esecuzione*)

Uno stesso programma può dare origine a diversi processi:

- Diversi utenti eseguono lo stesso programma

- Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

5.2.3 *Multiprogrammazione* e *multitasking*

Obiettivi del sistema operativo:

- Mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi (se ci sono programmi da eseguire)

- Dare l'illusione che ogni processo abbia una CPU dedicata

Due tecniche adottate nei sistemi operativi sono la **multiprogrammazione** e il **multitasking** (o **time-sharing**, sono sinonimi):

Obiettivo della multiprogrammazione: impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU

Obiettivo del multitasking: far si che un programma interattivo reagisca agli input utente in un tempo accettabile (è una tecnica **non** rilevante per i sistemi puramente *batch*, ma lo è principalmente per i programmi interattivi, che continuano a prendere dati dall'utente che li usa)

Multiprogrammazione: l'implementazione

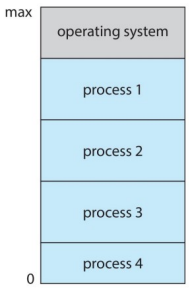
Il sistema operativo mantiene in memoria i processi da eseguire.
Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU.
Quando un processo non può proseguire l'esecuzione (ad es. perché deve attendere il termine dell'input di dati che gli servono per proseguire), la sua CPU viene assegnata ad un altro processo non in esecuzione.
Come risultato, se i programmi da eseguire sono più delle CPU, queste saranno impegnate nell'esecuzione di qualche processo per la maggior parte del tempo.
Questo risolve il problema dell'efficienza: *non tenere le CPU ferme.*

Multiprogrammazione: la memoria

La multiprogrammazione richiede che tutte le immagini di tutti i processi siano in memoria perché questi possano essere eseguibili.
Se i processi sono troppi non possono essere contenuti tutti in memoria: la tecnica dello **swapping** può essere usata per spostare le immagini dentro/fuori dalla memoria.

DEFINIZIONE

Swapping significa che il s.o. può togliere la memoria (anche detta *immagine*) di un processo dalla memoria centrale e caricare l'immagine di un altro processo.



La memoria virtuale è un'ulteriore tecnica che permette di eseguire un processo la cui immagine non è completamente in memoria.
Queste tecniche aumentano il numero di processi che possono essere eseguiti in multiprogrammazione, ossia il grado di multiprogrammazione.
Es.: se in un istante sto eseguendo 100 processi, il grado di multiprogrammazione è 100. Se in un altro istante sto eseguendo 50 processi, in quell'istante il grado di multiprogrammazione sarà 50.

Multitasking: l'implementazione

È un'*estensione della multiprogrammazione* per i sistemi interattivi.
La CPU viene "sottratta" periodicamente al programma in esecuzione ed assegnata ad un altro programma. Quindi non più solo quando il programma non è più in grado di proseguire ma anche ogni tot, così tutti i programmi hanno la possibilità di progredire.
In questo modo tutti i programmi progrediscono in maniera continuativa nella propria esecuzione, anziché solo nei momenti in cui il programma che detiene la CPU si mette in attesa.
Questo permette che i programmi batch, che effettuano poco I/O (e quindi vanno poco in attesa), non monopolizzino la CPU a discapito dei programmi interattivi.

5.3 Operazioni sui processi

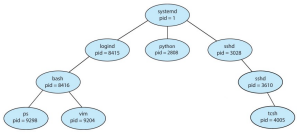
I sistemi operativi di solito forniscono delle chiamate di sistema con le quali un processo può creare/terminare/manipolare altri processi.
Dal momento che *solo un processo può creare un altro processo*, all'avvio il sistema operativo crea dei processi "primordiali" dai quali tutti i processi utente e di sistema vengono progressivamente creati.
Vedremo ora le operazioni fondamentali per:

- Creare processi

- Terminare processi

5.4 Creazione di processi

Chi crea il primo processo? Il s.o., in fase di booth. Il fatto che un processo crei un altro processo dà origine ad una **gerarchia**. Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica.
Un processo (padre) può creare altri processi (figli). Questi a loro volta possono essere padri di altri processi figli, creando un albero di processi.
Un albero di processi in Linux:



La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi.
Possibili politiche di condivisione di risorse:

- Padre e figlio condividono tutte le risorse ...
- ... o un opportuno sottoinsieme ...
- ... o nessuna

Possibili politiche di creazione spazio di memoria/indirizzi:

- Il figlio è un duplicato del padre (stessa memoria e programma) ...
- ... oppure no, e bisogna specificare quale programma deve eseguire il figlio Questo succede per le API Posix e non per le ?

Possibili politiche di coordinazione padre/figli:

- Il padre è sospeso finché i figli non terminano ...
- ... oppure eseguono in maniera concorrente

5.5 Terminazione di processi

I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo.
Un processo padre può attendere o meno la terminazione di un figlio.
Un processo padre può forzare la terminazione di un figlio. È sempre brutto terminare forzatamente un processo piuttosto che lasciarlo terminare spontaneamente, ma a volte si rende necessario. Possibili ragioni:

- Il figlio sta usando risorse in eccesso (tempo, memoria...).
- Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera "ordinata" tramite IPC).

- Il padre termina prima che il figlio termini (in alcuni sistemi operativi).

Riguardo all'ultimo punto, alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre:

- **Terminazione in cascata:** anche i nipoti, pronipoti... devono essere terminati (non c'è nelle API Posix).

- La terminazione viene iniziata dal sistema operativo.

Es.: API Posix

`fork()` crea un nuovo processo figlio; il figlio è un duplicato del padre ed esegue concorrentemente ad esso, ritorna al padre un numero identificatore (PID) del processo figlio e al figlio il PID 0.

`exec()` sostituisce il programma in esecuzione da un processo con un altro programma, che viene eseguito dall'inizio; viene tipicamente usata dopo una `fork()` dal figlio per iniziare ad eseguire un programma diverso da quello del padre.

`wait()` viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio; ritorna:

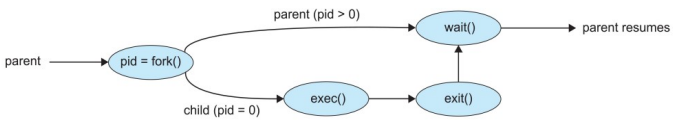
- Il PID del figlio che è terminato.
- Il codice di ritorno del figlio (passato come parametro alla `exit()`).

`exit()` fa terminare il processo che la invoca:

- Accetta come parametro un codice di ritorno numerico che dice come è andata
  (es. tipicamente se il figlio ritorna 0 allora è terminato normalmente, se invece è un numero $\neq 0$ c'è un errore di qualche tipo e il padre deve essere in grado di capire e gestire il codice di errore).
- Il sistema operativo elimina il processo e recupera le sue risorse.
- Quindi restituisce al processo padre il codice di ritorno (se ha invocato `wait()`, altrimenti lo memorizza per quando l'invocherà).
- Viene implicitamente invocata se il processo esce dalla funzione `main`.

`abort()` fa terminare forzatamente un processo figlio.

La sequenza `fork-exec` nelle API POSIX:



5.5.1 Processi *zombie* e *orfani*

Ricordiamo che abbiamo detto che nelle API Posix non c'è la terminazione a cascata.

P. zombie: Se un processo termina ma il suo padre non lo sta aspettando (non ha invocato `wait()`) il processo è detto essere ***zombie***.
Praticamente il processo ritorna un codice di errore ma non c'è nessuno a prendere e gestire il codice di errore. Il processo viene allora riallocato ma il codice rimane, quindi il processo non è effettivamente morto. Per questo *zombie*.

P. orfano: Se un processo padre termina prima di un suo figlio, il figlio è detto essere ***orfano*** (non vi è terminazione a cascata).

5.6 Implementazione dei processi

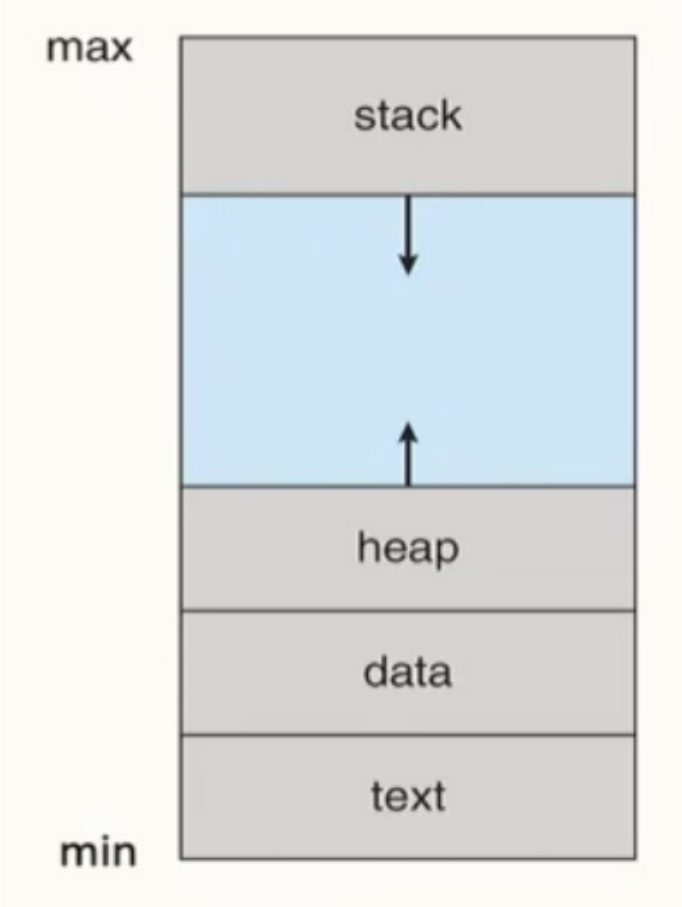
5.6.1 Struttura di un processo

Un processo è composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il program counter.
- Lo stato della regione di memoria centrale usata dal programma, o **immagine** del processo.
- Lo stato del processo stesso.
- Le risorse del sistema operativo in uso al programma (files, semafori, regioni di memoria condivisa...).

Processi distinti hanno *immagini distinte*.
Le risorse del sistema operativo invece possono essere condivise tra processi (a seconda del tipo di risorsa).

Immagine di un processo



L'immagine di un processo è formata da:

- Il codice del programma (**text section**)
- La **data section**, contenente le variabili globali
- Lo **heap**, contenente la memoria allocata dinamicamente durante l'esecuzione
- Lo stack delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno

Text e *data section* hanno dimensioni costanti per tutta la vita del processo. *Stack* e *heap* invece crescono/decrescono durante la vita del processo, smetteranno solo quando una delle due sezioni raggiungerà la fine della memoria allocata per il processo, ovvero collidono tra loro.