Dario Olianas A.A. 2013/2014

Appunti di Sistemi di Elaborazione e Trasmissione (SET)

Sistemi operativi (prof. Chiola):

Reti (prof.ssa Ribaudo):

1. Virtualizzazione, standardizzazione, sicurezza ed efficienza

1.1 Virtualizzazione

Abbiamo già visto alla fine del corso di SEI la virtualizzazione della memoria, utilizzata per motivi di protezione e di efficienza. Si può infatti definire un segmento in cui è contenuto il codice da eseguire e un altro in cui sono contenuti i dati. La MMU (Memory Management Unit) è il dispositivo hardware che si occupa di gestire la memoria virtuale e impedisce accessi non consentiti alla memoria: ad esempio in caso di tentativo di esecuzione di un segmento dati.
La MMU si occupa inoltre della traduzione degli indirizzi di memoria virtuale in indirizzi fisici, attraverso la segmentation table, e quando viene fornito per la traduzione un indirizzo virtuale non valido lo segnala con una trap.
Il codice di sistema e il codice delle applicazioni devono risiedere in segmenti diversi, poiché per ipotesi il codice di sistema è esente da errori, e quindi può (e deve) essere eseguito con pieni permessi di accesso ad ogni risorsa. In realtà non è per niente vero che il codice di sistema è privo di bug, ma senza questa assunzione non potremmo avere un sistema operativo (il SO deve avere pieno accesso alle risorse per poterle gestire). Il codice delle applicazioni invece è considerato inaffidabile, e viene eseguito sotto il controllo del SO.
Quando un'applicazione tenta di eseguire un accesso non consentito alla memoria, la MMU se ne accorge e genera una trap, che interrompe l'esecuzione dell'istruzione corrente e invoca il trap handler, contenuto nel codice di sistema (quindi fidato, con pieni permessi di esecuzione).
Uno degli scopi principali del sistema operativo è proprio quello di virtualizzare le risorse, rendendole facilmente disponibili ai vari software applicativi senza che debbano gestirle direttamente e senza conflitti in caso di più applicazioni eseguite contemporaneamente.
Tramite la virtualizzazione, posso fare in modo che ogni processo creda di avere a disposizione un'intera macchina, con il suo processore, la sua memoria, il suo disco e le sue periferiche.
La virtualizzazione è fondamentale non solo per i sistemi operativi: anche il cloud è basato sulla virtualizzazione, poiché consente di vedere dispositivi remoti come se fossero parte della mia macchina.

/* La virtualizzazione è spiegata molto più approfonditamente negli appunti di SEI del 2012 */

1.2 Standardizzazione

La **standardizzazione** è un altro concetto chiave dei sistemi operativi, fondamentale per l'interoperabilità: un sistema operativo dovrà far funzionare assieme svariati dispositivi e software di diversi produttori: ciò è possibile solo operando con standard comuni.

In questo corso faremo riferimento principalmente a sistemi operativi UNIX based. Ci sono moltissimi sistemi derivati da UNIX, ognuno con le sue peculiarità ma l'interoperabilità tra di loro è garantita dallo standard **POSIX**, che fornisce una serie di primitive implementate allo stesso modo su tutti i sistemi che lo rispettano. Persino Windows, entro certi limiti, è POSIX compliant.

1.3 Sicurezza

Alcuni punti fondamentali per la sicurezza di un sistema operativo:

- **controllo degli accessi**
- **integrità**
- **disponibilità**
- **affidabilità**
- **segretezza**

Il **controllo degli accessi** consente di sapere sempre chi sta utilizzando la macchina, tramite richiesta di username e password per il riconoscimento dei diversi utenti e assegnazione di diversi privilegi a ciascun utente. Il controllo degli accessi non serve solo a difendersi da attacchi esterni, ma anche da errori dell'utente: assegnando ad un utente minori privilegi si riducono anche le sue possibilità di danneggiare il sistema.

L'**integrità** consiste nell'impedire che un software installato perda le sue caratteristiche di affidabilità per interventi più o meno volontari dell'utente o di altre applicazioni.

La **disponibilità** consiste nella risposta immediata ai comandi, senza inutili tempi di attesa. Anche se non sembra una caratteristica di sicurezza, basta pensare ad un computer incaricato di gestire un aereo o una centrale nucleare per capire quanto la reattività ai comandi sia fondamentale per la sicurezza del sistema: se il pilota corregge la rotta, la rotta deve essere corretta immediatamente. La disponibilità si misura in termini di tempo.

L'**affidabilità** è un concetto simile alla disponibilità, ma meno stringente: un sistema affidabile potrà subire dei guasti per cui diventa non disponibile, ma mi da garanzia di recupero per cui tornerà ad esserlo. Un esempio è un comune sistema operativo: può bloccarsi, e quindi non essere più disponibile, ma se riavvio la macchina il sistema riprenderà a funzionare come prima. È una caratteristica fondamentale perché un calcolatore, in quanto sistema fisico, può rompersi, ma se sostituisco la parte danneggiata devo avere la garanzia che la macchina ritorni a funzionare come prima. Tipicamente l'affidabilità viene raggiunta attraverso la **ridondanza**: utilizzare più risorse di quelle necessarie in modo che in caso di guasto di un componente nulla vada perduto. Ad esempio: un computer con un singolo hard disk non è affidabile perché in caso di rottura del disco avrò perso tutti i miei dati. Se ho un secondo hard disk di backup con una copia identica del primo avrò un sistema affidabile.

La ridondanza può essere usata anche per aumentare la disponibilità: un sistema real time che deve darmi la risposta in un secondo, in condizioni normali dovrà darmela in meno di un secondo, in modo che in caso di guasti ci sia tempo di utilizzare i sistemi di emergenza per fornire la risposta.

1.4 Efficienza e prestazioni

Come abbiamo detto, durante il suo funzionamento un sistema operativo moltiplica, attraverso la virtualizzazione, le risorse disponibili in modo da fornirne una copia ad ogni processo. Ma anche il sistema operativo è un processo in esecuzione sulla macchina, e dovrà quindi usare il minimo delle risorse possibili in modo da lasciarne di più per le applicazioni che dovrà eseguire. Qualche decina di anni fa il problema dell'efficienza era molto sentito, a causa del maggior costo delle risorse hardware. Ma l'efficienza di un sistema operativo è tutt'ora importante, perché è aumentata la quantità di risorse disponibili ma sono aumentate anche le aspettative, e anche perché virtualizzare una grande quantità di risorse è più complicato di virtualizzarne poche.

2. Componenti di un sistema operativo

Prenderemo come riferimento la struttura di un sistema operativo UNIX.

2.1 Kernel

Il **kernel** è lo strato di base che implementa la virtualizzazione e solitamente anche il controllo degli accessi. Appena la macchina viene accesa, la RAM contiene una serie di bit casuali e non predicibili. Ma nello stesso spazio di indirizzamento della RAM c'è anche una memoria EPROM, non volatile. Il program counter della CPU al momento dell'accensione contiene l'indirizzo della EPROM, il cui contenuto sarà il primo programma ad essere eseguito dalla CPU.

Il BIOS (Basic Input/Output System) tra le altre cose identifica il dispositivo (generalmente l'hard disk) dal quale effettuare il **bootstrap**. È utile che il BIOS consenta di modificare questo dispositivo, per consentire ad esempio l'installazione di un nuovo sistema.

2.1.1 Bootstrap

L'operazione di **bootstrap** è quella che mi consente di caricare in memoria nuovo codice eseguibile. Il primo programma ad essere caricato in memoria dal BIOS è il **boot loader**: un piccolo programma che consente più opzioni di avvio di quelle fornite dal BIOS: ad esempio può consentirmi di scegliere tra diversi sistemi operativi.

Il boot loader carica una versione preliminare del sistema operativo, incaricata di predisporre la virtualizzazione della memoria. Per fare questo, il processore dovrà essere in modalità privilegiata, quella in cui si accede direttamente alla memoria con gli indirizzi fisici.

Tutte le caratteristiche di sicurezza descritte prima sono disponibili solo dopo l'attivazione della memoria virtuale, quindi è opportuno che la predisposizione della memoria virtuale sia il più veloce possibile.

Il kernel viene lanciato solo dopo l'attivazione della memoria virtuale.

Un **processo** è un'applicazione mandata in esecuzione sotto il controllo del sistema operativo, con il suo segmento di codice diviso in codice di sistema e codice applicazione. Quando viene eseguita un'applicazione, il sistema fa una chiamata alla funzione **`main()`** della mia applicazione, e passa il processore in modalità non privilegiata: qualunque istruzione privilegiata all'interno dell'applicazione genererà una **trap** per la cui gestione sarà invocato il **trap handler** del sistema operativo, che è parte del kernel e viene dunque eseguito in modalità privilegiata. In caso di più processi in esecuzione, ognuno avrà vritualizzato il suo segmento di codice di sistema, uguale per tutti. Questa caratteristica mi consente il **process switch**: quando un processo è in esecuzione, il sistema può assegnargli un tempo di esecuzione dopo il quale viene inviato un interrupt: questo provocherà l'interruzione dell'applicazione e l'invocazione dell'interrupt handler, che riconosciuta la natura dell'interrupt salverà lo stato attuale dell'applicazione e ne manderà in esecuzione un'altra.

Cosa succede quando un'applicazione chiede più risorse di quelle che le sono state assegnate?

Supponiamo che il sistema abbia assegnato ad un'applicazione 1 MB di stack, e che l'applicazione chiami una funzione ricorsiva che esaurisce rapidamente lo spazio. Una volta esaurito, l'applicazione chiederà di accedere ad un indirizzo che non fa parte del suo segmento stack. La MMU se ne accorge e genera una trap, il trap handler del sistema operativo la riconosce e aumenta la dimensione del segmento stack dell'applicaizone.

2.1.2 Kernel monolitici e modulari

Il kernel è la parte di software di sistema che virtualizza le risorse hardware, per effettuare questa virtualizzazione c'è la necessità di avere un alto grado di integrazione tra kernel e hardware: i componenti del kernel saranno quindi molto legati ai dispositivi fisici che costutuiscono la macchina. Ma uno stesso sistema operativo deve poter essere installato su macchine diverse.
Nei sistemi UNIX solitamente si usa un **kernel monolitico**: un kernel che viene configurato prima della compilazione in modo da avere tutte le componenti necessarie per l'hardware sul quale dovrà essere installato. Ma questo è uno svantaggio in caso di aggiornamenti hardware alla macchina: dovrò ricompilare il kernel per avere i componenti che mi servono.
La soluzione è un **kernel modulare**: un kernel al quale possono essere aggiunti e tolti moduli che danno il supporto a particolari dispositivi, anche durante l'esecuzione. I kernel moderni sono di tipo modulare.
Esistono anche altri tipi di kernel, che vedremo successivamente: i **microkernel** sono kernel estremamente semplificati che fanno solo il minimo indispensabile (virtualizzazione) e demandano tutte le altre necessità a processi di tipo applicativo.

2.2 Librerie

Le librerie vanno ad aggiungere codice nel segmento applicativo dei processi. Attraverso queste vengono implementate le **system call**: chiamate che permettono ad un'applicazione di chiedere funzionalità al sistema. Abbiamo già visto che per passare dalla modalità utente alla modalità sistema c'è bisogno delle trap. Ma l'utilizzo delle trap non è semplicissimo per il programmatore, perché richiede la conoscenza del funzionamento del trap handler del sistema operativo, e sarebbe inoltre ridotta la portabilità, poiché ogni sistema operativo ha un suo trap handler. Lo scopo delle system call è proprio quello di evitare al programmatore l'uso diretto delle trap: dovrà solo invocare una funzione che passa il controllo al sistema, inoltrandogli le richieste dell'applicazione. Lo standard POSIX stabilisce nomi e parametri delle system call. In un certo senso potremmo dire che le librerie virtualizzano per le applicazioni il vero funzionamento del sistema operativo.

2.3 Altre componenti

Log

Nei file di log vengono registrati tutti gli eventi che accadono sulla macchina: avvio, login, login errati, errori ecc. Utili in caso di malfunzionamenti per capire quale sequenza di eventi ha portato all'errore. Poiché registrando tutto raggiungono molto presto grandi dimensioni, sarà utile avere programmi per gestirli.

Applicazioni di gestione

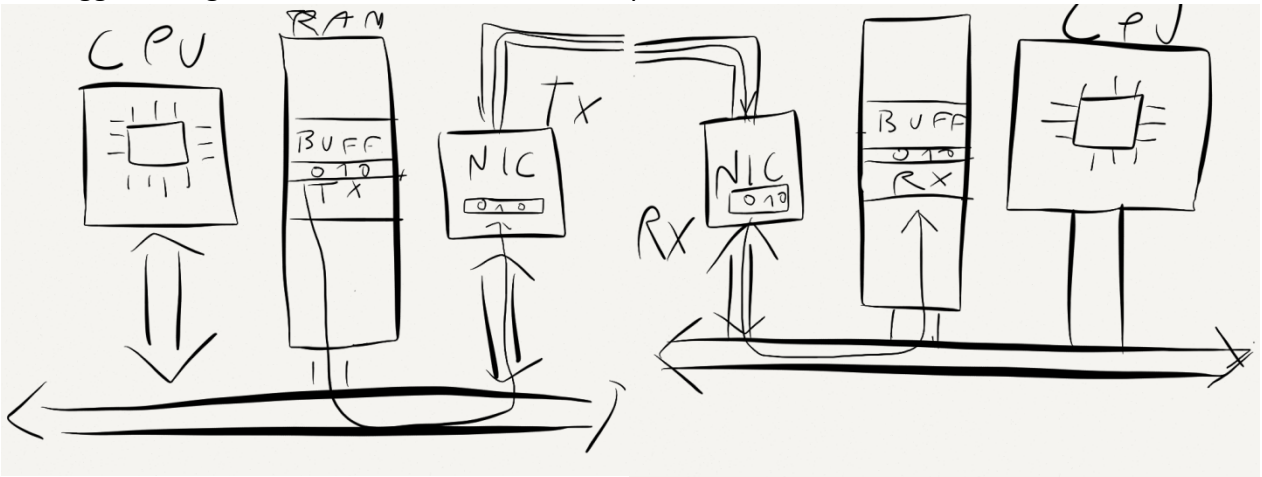
Per il funzionamento di un sistema operativo avremo bisogno di applicazioni che permettano all'utente di gestirlo: applicazioni per accedere ai file, aggiungere utenti, installare applicazioni, gestione dei log.

Applicazioni per lo sviluppo

Compilatori, debugger, IDE, editor.....

3. Scambio di dati via rete

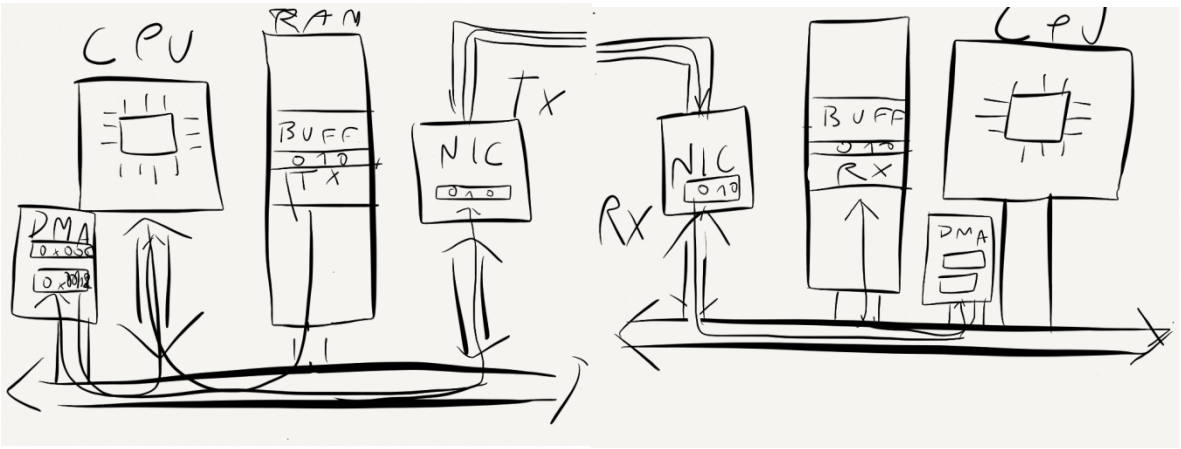
Immaginiamo di voler far comunicare due programmi in esecuzione su macchine diverse. Per farlo ho innanzitutto bisogno che entrambe le macchine abbiano un'interfaccia di rete. Serve poi che nella macchina ricevente ci sia un'area di memoria libera abbastanza grande da contenere i dati da ricevere (**buffer di ricezione**). Anche il mittente avrà un buffer, detto **buffer di trasmissione**.
Ci sarà un programma che legge dal buffer di trasmissione e lo copia, parola per parola, in un registro dell'interfaccia di rete, che vedendo dati in ingresso si attiverà e invierà i dati attraverso un canale di comunicazione verso l'interfaccia del ricevente. Nel ricevente ci sarà un programma che legge dal registro dell'interfaccia di rete e copia nel buffer di ricezione.



Che problemi può dare un'implementazione simile? Beh, intanto il bus che trasferisce i dati dalla RAM all'interfaccia di rete è molto più veloce della connessione di rete: dovrà quindi essere opportunamente sincronizzato. Bisognerà anche coordinare l'attività del programma che trasmette e del programma che riceve, per evitare perdite di dati: i due programmi sono eseguiti su due macchine diverse, quindi con diversa frequenza di clock, e se uno va più veloce dell'altro ci saranno perdite di dati.

3.1 Trasmissione

Questo è un modello molto semplificato, che implementato nella realtà sarebbe molto inefficiente. Il bus infatti è progettato per avere la più alta velocità possibile, in modo da accelerare i trasferimenti di dati e quindi la velocità generale della macchina. La rete a confronto è molto più lenta, quindi per essere mantenuto sincrono il programma che trasmette sulla rete dovrebbe essere estremamente rallentato, poiché dovrebe scrivere il dato da trasmettere nel registro della NIC, aspettare che venga trasmesso (se ne accorge perché il buffer di trasmissione si svuota) e scrivere il dato successivo. Il programma passerebbe quindi la maggior parte del tempo in attesa.
Per ovviare a questo problema è stato introdotto il **DMA channel**: un coprocessore in grado di eseguire un piccolo sottoinsieme dell'instruction set del processore, tipicamente istruzioni relative alla copia di dati. Essendo un processore, il canale DMA conterrà dei registri in cui la CPU può scrivere. Il nostro programma di trasmissione potrà scrivere una serie di valori nei registri del canale DMA, che si occuperà della trasmissione. I valori richiesti dal DMA channel sono gli indirizzi di inizio e fine del buffer di trasmissione: una volta ricevuti si occuperà lui di trasmetterlo senza ricorrere al processore principale, il quale dovrebbe occuparsi solo della programmazione del canale DMA.



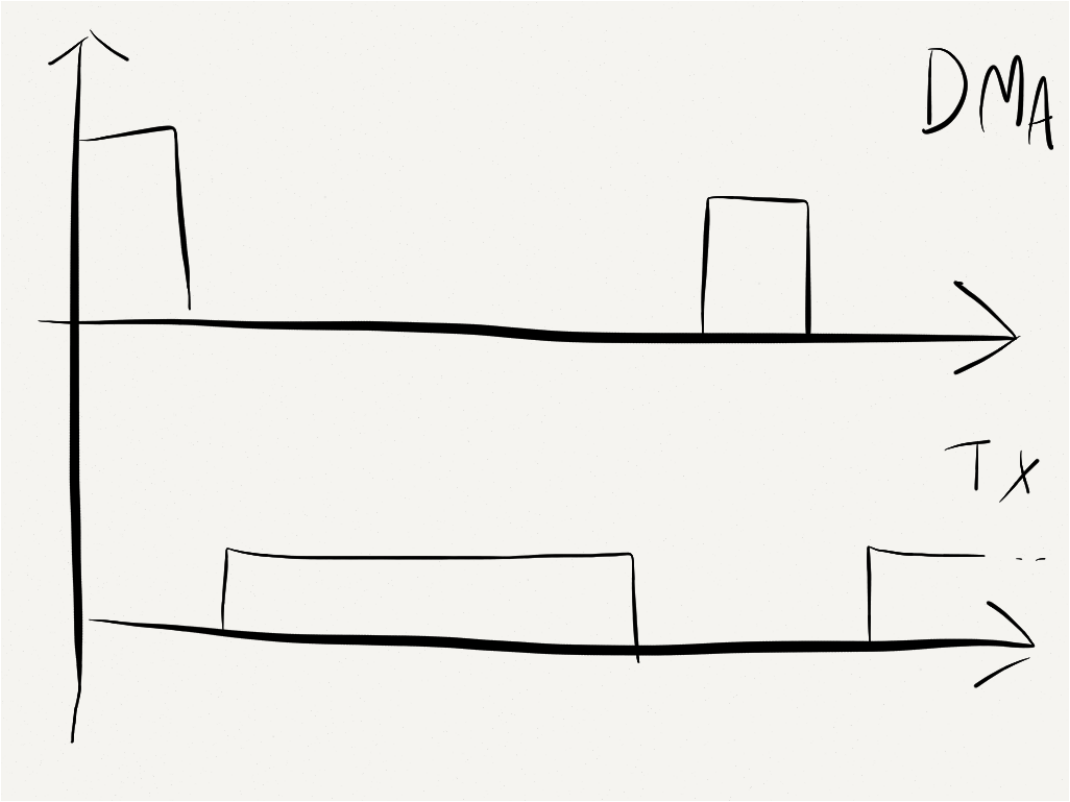
L'uso del DMA introduce un altro problema: se prima il bus era gestito solo dal processore, ora avrò due dispositivi che possono usarlo e devo assicurarmi che non vadano in conflitto. Servirà quindi un arbitro che "diriga il traffico", decidendo ogni volta chi può utilizzare il bus. Quando l'arbitro nega l'utilizzo del bus al processore, questo potrà andare avanti nel suo funzionamento con i dati di cui dispone nella cache. È infatti raro che il processore richieda l'accesso alla RAM: mediamente una volta su 20, le altre volte i dati necessari sono già in cache. Per segnalare al processore che è finita la trasmissione, il DMA invia al processore un interrupt.

3.2 Ricezione

In ricezione, alla NIC arriveranno dati dalla rete, che verranno scritti nei suoi registri. Quando i registri sono pieni, il processore deve copiare i dati nel buffer di ricezione in modo da liberare i registri per i nuovi dati in arrivo. I registri pieni vengono segnalati alla CPU attraverso un interrupt, e questo rallenta notevolmente il processore che sarà continuamente interrotto nel suo funzionamento. Inoltre, se al momento in cui viene inviato l'interrupt il processore sta funzionando in modalità sistema, non sentirà gli interrupt (perché potrebbe essere in corso la gestione di un altro interrupt o di una trap). Se il processore non risponde all'interrupt e continuano ad arrivare dati dalla rete, la NIC non sa più dove metterli e rischierò di perderli.
Anche in questo caso risolvo il problema con il DMA: il DMA channel della maccina ricevente sarà dedicato ad aspettare i dati dalla NIC, prenderli quando sono pronti e copiarli nel buffer di ricezione. Anche in questa implementazione avrò bisogno di un interrupt, la prima volta, per attivare il DMA. Ma se la NIC invia l'interrupt appena arrivano dati e non quando i registri sono pieni, avrò più tempo utile per segnalare l'arrivo di dati e la necessità di attivazione del DMA alla CPU.
Una soluzione più efficiente è integrare il DMA nell'interfaccia di rete, in modo da utilizzare un solo ciclo di lettura sul bus invece di due (si elimina la fase di trasmissione dal DMA alla NIC). È questa la soluzione usata nei computer moderni: il DMA channel unico per tutto il sistema era usato una trenitna di anni fa, adesso si usa il **DMA bus mastering**, in cui ogni dispositivo ha il suo DMA.

3.3 Sincronizzazione

Abbiamo già detto che uno dei problemi della comunicazione tra macchine remote è che ogni parte del percorso dalla macchina trasmittente a quella ricevente ha velocità diversa: il clock del processore del trasmittente ha velocità diversa dal bus, che a sua volta ha velocità diversa dal canale di comunicazione. Anche la macchina ricevente avrà una frequenza di clock della CPU diversa, e quando il mittente è molto più veloce del ricevente il rischio è la perdita di dati. Bisogna quindi trovare una soluzione per sincronizzare tutti. Questo viene effettuato sul canale di comunicazione attraverso algoritmi di **flow control** (controllo di flusso), che cercano di regolare la velocità di mittente e destinatario rispetto al canale di comunicazione.
Supponiamo di avere un mittente più veloce del destinatario: con un controllo di flusso i dati vengono rallentati in fase di trasmissione in modo da dare tempo al ricevente.
Avremo quindi un primo trasferimento dalla RAM al DMA dell'interfaccia di rete con la prima parola da trasmettere. Segue un lungo intervallo in attesa che il trasferimento verso la rete sia terminato. A questo punto ci sarà un altro trasferimento DMA dalla RAM per la seconda parola, e così via.



Vediamo che ci sono tempi morti anche in trasmissione, quando la NIC sta aspettando dal DMA la prossima parola da trasmettere. Come li posso evitare? Posso aggiungere un secondo registro alla NIC, che conterrà un'altra parola. Una volta finito di trasmettere il contenuto del primo registro, inizierò subito a trasmettere il secondo mentre il DMA carica un nuovo valore da sovrascrivere a quello del registro già trasmesso. Con due o più registri nella NIC, vi abbiamo implementato un **buffer**.
Lo stesso discorso vale per la ricezione: con un solo registro avrei tempi morti nell'attesa che un valore appena ricevuto venga scritto in RAM dal DMA: con un buffer di ricezione posso continuare a ricevere mentre il DMA trasferisce i dati ricevuti dal buffer della NIC alla RAM. In genere le interfacce di rete hanno un **firmware**, sono piccole macchine programmabili.

Finora abbiamo definito i buffer di trasmissione e ricezione in RAM come porizioni di memoria identificate da un indirizzo di inizio e uno di fine, quindi contigue. Ma cosa succede se voglio trasmettere dati non contigui? Dovrò riprogrammare i registri del DMA più volte, una per ogni frammento contiguo di dati, e la cosa può diventare poco efficiente per dati molto frammentati. Si risolve complicando ulteriormente il DMA della NIC, rendendolo in grado di gestire strutture dati dette **ring**.

3.3.1 Ring based DMA

Supponiamo di voler ricevere su due buffer non contigui: come lo comunico al DMA della scheda di rete? La CPU crea in memoria una struttura dati, il **ring**, che conterrà le coordinate di uno dei due buffer, e passa l'indirizzo del mio ring al DMA, che leggerà gli indirizzi contenuti nel ring e inizierà a ricevere nel buffer corrispondente. Il DMA è sensibile ai cambiamenti del ring: se il processore modifica in corsa un indirizzo del ring, il DMA se ne accorge e inizia a utilizzare il nuovo buffer. I ring possono essere usati per programmare più trasferimenti consecutivi su buffer diversi, scrivendone consecutivamente gli indirizzi nel ring. L'utilizzo dei ring è quindi una bufferizzazione della programmazione del DMA.
Queste porzioni di memoria contenenti gli indirizzi dei buffer sono chiamate ring perché vengono lette in modo circolare.
Se il canale di comunicazione è **full duplex** si può inviare e ricevere contemporaneamente, se è half duplex no.

3.4 Virtualizzazione dell'attività di rete

Abbiamo visto come funziona la trasmissione e la ricezione dei dati da una macchina all'altra: il passo successivo è virtualizzare tutto per le applicazioni.
Se ho tre processi che vogliono trasmettere sullo stesso canale, posso farli trasmettere uno per volta, in coda (**multiplexing**). In ricezione (**demultiplexing**) è più complicato perché non so con che criteri sia stato fatto il multiplexing sul trasmittente, i dati trasmessi dovranno includere informazioni che mi dicono quali dati corridpondono a quale processo. Per fare questo posso usare dei buffer generici, gestiti dal sistema operativo, in cui mettere i dati in attesa di smistamento. Una volta ricevuti i dati li andrò a leggere e saprò a quale processo appartengono. La dimensione di questi buffer generici dovrà essere la massima possibile, in modo da poter ricevere anche i messaggi lunghi. Ma essendo decisa dal sistema operativo, le applicazioni dovranno adattarsi a non trasmettere o ricevere messaggi più lunghi della dimensione massima impostata dal sistema: per trasferimenti più lunghi i dati da trasmettere andranno scomposti in pacchetti della dimensione del buffer da trasmettere uno per volta. La dimensione dei buffer di ricezione deve essere convenzionale: altrimenti il mittente potrebbe inviare pacchetti più grossi.
Il demultiplexing dei dati ricevuti è effettuato dal sistema operativo.
In pratica, come avviene il demultiplexing? Bisogna riassegnare il segmento desiderato, che per ora appartiene al sistema operativo, al processo cui i dati erano destinati. Per fare questo bisogna agire sulla memoria virtuale. Oppure, si possono copiare i dati dal segmento del sistema operativo a quello dell'applicazione. Quale delle due soluzioni sia la più efficiente è un dettaglio implementativo: su alcuni sistemi può essere più veloce riassegnare un segmento, su altri copiare i dati.

Memoria condivisa e scambio di messaggi
All'interno di un sistema, la memoria è condivisa (**shared memory**) tra il processore e tutti i dispositivi dotati di DMA, che possono usarla sia per conservare i loro dati sia per comunicare tra di loro (ring). Tra due computer connessi in rete si usa un approccio diverso, quello del **message passing** (scambio di messaggi): nessuno ha accesso alla memoria dell'altro, ma le due macchine possono inviarsi messaggi che saranno gestiti esclusivamente dalla macchina ricevente. Questa differenza di approccio è motivata da problemi fisici: con una maggiore distanza, un sistema a RAM condivisa sarebbe inefficiente.

API per la comunicazione di rete
Abbiamo visto che la comunicazione di rete è un problema piuttosto complesso: sarà utile allora virtualizzarne il funzionamento per le applicazioni. Questo avviene attraverso delle API (Application Program Interface) standard, in modo da garantire la portabilità.
Pensiamo al problema della computazione distribuita: devo usare le risorse di vari computer sparsi per il mondo, anche a grande distanza, ma connessi in rete. La soluzione è stata trovata nel protocollo MPI (Message Passing Interface). Sviluppando software che dovrà essere eseguito in un sistema distribuito, devo fare in modo che il numero di processori da usare non sia definito ma possa cambiare ad ogni esecuzione. Questo è possibile definendo un numero, il **rank**, che identifica i vari processi della mia applicazione. Se ne lancio solo uno il suo rank sarà 0, se ne lancio due avrò due processi 0 e 1 e così via. Questi numeri potrò poi utilizzarli come identificativo del mittente e identificativo del destinatario per lo scambio di messaggi tra i vari processi in esecuzione su macchine diverse. È compito dell'interfaccia MPI mandare in esecuzione tante copie del programma su tante macchine diverse e metterle in comunicazione tra loro.
Può essere gestita una comunicazione uno a uno, uno a molti, molti a uno.

3.5 Socket
L'API di comunicazione per eccellenza è il **socket**. L'idea del socket è quella di "fare un buco" nel perimetro che delimita il nostro processo in modo da far entrare e uscire flussi di informazione. Due processi in esecuzione su macchine diverse dovranno avere entrambe un socket aperto per comunicare. Non è indispensabile che i due processi siano su macchine diverse: possono anche essere sulla stessa. Come si fa a fare in modo che un processo sappia che c'è un altro processo che vuole comunicare con lui e quindi fargli aprire un socket?
Per poter riconoscere univocamente ogni computer sulla rete è necessario che pgni computer abbia un suo **indirizzo di rete** che lo identifichi univocamente. Il protocollo IP esiste in due versioni, **IPv4** e **IPv6**, differenti per il numero di bit da cui è composto l'indirizzo. Tramite l'indirizzo possiamo far arrivare il messaggio alla macchina di destinazione: manca il demultiplexing per capire a quale processo della macchina ricevente è destinato il messaggio. A livello di sistema, il demultiplexing può essere gestito tramite **porte**, identificate da un numero di 16 bit. Ad ogni porta va assegnato un socket, che riceverà i dati da essa. Questa associazione è chiamata **bind**.
Come fa il mittente a sapere su quale porta troverà il ricevente? Deve essere deciso in precedenza. Ma essendo le porte gestite dal sistema, un processo potrebbe chiadere il bind su una porta e il sistema potrebbe negarglielo, magari perché la porta è già occupata. Il processo dovrebbe allora comunicare al mittente il cambiamento di porta. Ma anche il mittente può avere di questi problemi, quindi l'unico modo per garantire la comunicazione con questo sistema è necessario che almeno uno dei due abbia un'associazione fissa con una porta. È un esempio di architettura **client/server**: chi ha un bind fisso con la porta viene chiamato server, l'altro client.

L'assegnazione delle porte è uno standard, che le suddivide in **well known** ed **ephimeral**: le well known sono assegnate ai server, le ephimeral ai client. In questo modo i server sono sempre contattabili, avendo porte assegnate dallo standard, e per prima cosa il client dovrà comunicare su quale porta ephimeral ha fatto il bind in modo che il server possa rispondergli. Dopo questa fase non ha più senso parlare di client e server: una volta che entrambi i processi sono reciprocamente raggiungibili la comunicazione diventa perfettamente simmetrica.
I socket possono essere utilizzati in due modi: **datagram** e **stream**. La datagram è la modalità più semplice, ma con meno funzionalità: dopo che il server ha fatto il bind il client può subito iniziare la comunicazione tramite la funzione **send**.

Funzioni POSIX utilizzate per la gestione dei socket:

- **socket:** crea un socket
- **bind:** chiamata dal server per farsi assegnare una porta
- **send:** invia dati. In modalità datagram si chiama **send_to**
- **recv:** riceve dati. Prende come parametro un puntatore al buffer di ricezione. In modalità datagram si chiama **recv_to**

Gli indirizzi di rete sono suddivisi in **network** e **host**: una parte dell'indirizzo identifica la rete a cui il computer è connesso, l'host identifica il computer.

4. File descriptor

Un file descriptor è un numero intero che rappresenta una risorsa messa a disposizione dal sistema per un processo che ne fa richiesta. Il sistema mantiene un array, in cui ogni posizione contiene le informazioni per raggiungere uno specifico file e il file descriptor è un indice di questo array. Da notare che per "file" non si intende per forza un file nel classico senso del termine, ma ouò essere anche, ad esempio, un socket, un dispositivo o un altro tipo di risorsa. Le prime tre posizioni di questo array, quindi i file descriptor 0, 1 e 2 sono associate ai file **sdtin, stdout** e **stderr**, che sono a loro volta collegati a tasitera e terminale. Stdout e stderr fanno entrambi rifetimento al terminale, ma con una differenza logica: stdout è gestito in **modalità bufferizzata**, stderr no. I dati stampati su stdout vanno prima a finire in un buffer, e potrebbero quindi non venire stampati in caso di errori, mentre i dati stampati su stderr vengono immediatamente mostrati a terminale.
I file descriptor agiscono quindi da porta tra un processo e il mondo esterno: se non esistessero un processo potrebbe lavorare solo sulle sue variabili interne. Ogni volta che un processo vuole aprire un nuovo canale di comunicazione con l'esterno, deve farne richiesta al sistema operativo, attraverso system call come read(), write(), open(), socket(). Il sistema andrà a prendere la prima posizione libera nell'array dei file descriptor, vi metterà le informazioni necessarie ad accedere alla risorsa richiesta e ne restituirà l'indice al processo. L'idea che sta dietro a questa soluzione è che il processo possa accedere all'array dei file descriptor, che sarà quindi allocato nella memoria utente, ma non al contenuto delle sue celle, che dovrà essere gestito esclusivamente dal sistema operativo poiché sono risorse esterne al processo, la cui modifica potrebbe influenzare l'intero sistema. Quando un processo ha bisogno di accedere alla risorsa, ne passerà il file descriptor come parametro ad una system call, che chiederà al sistema operativo di svolgere l'operazione richiesta.

Queste system call dovranno attivare una trap al loro interno per passare il controllo al sistema operativo.
Ci sono primitive generiche, come read() e write(), che permettono di agire su diversi tipi di file descriptor, e primitive specifiche come quelle per i socket (send(), recv(), connect()).
I socket, quando vengono utilizzati dal programmatore, possono essere visti come buffer indefinitamente grandi in cui posso andare a scrivere quanti dati voglio. In realtà, i socket virtualizzano un sistema di buffer più complesso, in cui ci sono buffer di trasmissione e buffer di ricezione. Se io continuo ad inviare dati ma il ricevente non svuota i buffer, arriverò ad un punto in cui i buffer di ricezione e dei nodi intermedi saranno saturi. Occorre quindi un meccanismo di **controllo di flusso** che notifichi al trasmittente quando i buffer di ricezione sono saturi.
Le primitive read() e write() sono **bloccanti**, ovvero bloccano il processo che le ha lanciate se non sono in condizione di portare a termine il compito per mancanza di risorse. Per evitare blocchi e sblocchi troppo frequenti, che peggiorerebbero le prestazioni, il blocco viene effettuato quando il 90% dei buffer è saturo e lo sblocco non viene effettuato finché non si liberano almeno al 50%.

5. Processi

In UNIX i processi si possono duplicare. Quando noi abbiamo un processo in esecuzione, possiamo usare la system call **`fork()`** che ha l'effetto di fare una copia identica del processo che l'ha chiamata. Ma chi crea il primo processo? In fase di boostrap, dopo che la memoria virtuale è stata inizializzata correttamente, il kernel crea il primo processo, init, che potrà avviarne altri tramite la fork.
Il primo processo viene autogenerato passando dalla modalità reale (indirizzamento diretto della memoria) a quella protetta (memoria virtuale): il programma suddivide la sua memoria in memoria utente e memoria di sistema, diventando un processo. Questo processo, chiamando la fork nel suo codice utente, potrà crearne altri.
Cosa vuol dire che un processo duplica se stesso?

- le strutture dati che rappresentano il processo nel kernel vengono replicate.
- i segmenti dati e codice del processo vengono replicati

Avrò così un secondo processo uguale identico al primo. Identico anche nello stato: il processo figlio non partirà dall'inizio del main, ma dal momento in cui è stata chiamata la fork, perché anche il program counter è stato copiato dal padre. Come faccio a differenziarne il funzionamento? Bisogna guardare il valore ritornato da **`fork()`**, un intero che sarà maggiore di zero nel padre e zero nel figlio: è l'unico modo che hanno per differenziarsi, essendo due copie identiche dello stesso processo. La fork può anche ritornare -1 in caso di errore, se ad esempio non c'è abbastanza memoria per avviare un nuovo processo: questo sarà ovviamente ritornato nel padre (il figlio non è stato creato) che potrà agire di conseguenza. Il processo pong dell'esercitazione, ad esempio, fa una fork ogni volta che accetta una connessione poiché deve creare un altro processo che interagisca col ping, mentre quello in ascolto sulla pota 1491 deve restare in ascolto nel caso arrivino altre richieste di connessione. I vari processi sono identificati dal sistema da un **PID (process identifier)**, un numero positivo diverso per ogni processo, corrispondente al valore ritornato dalla fork al padre del processo.
Un altra funzione fondamentale per la creazione di nuovi processi è **`int execve(filename, argv, envp)`**, che serve a mandare in esecuzione il file eseguibile **`filename`**. Quando viene

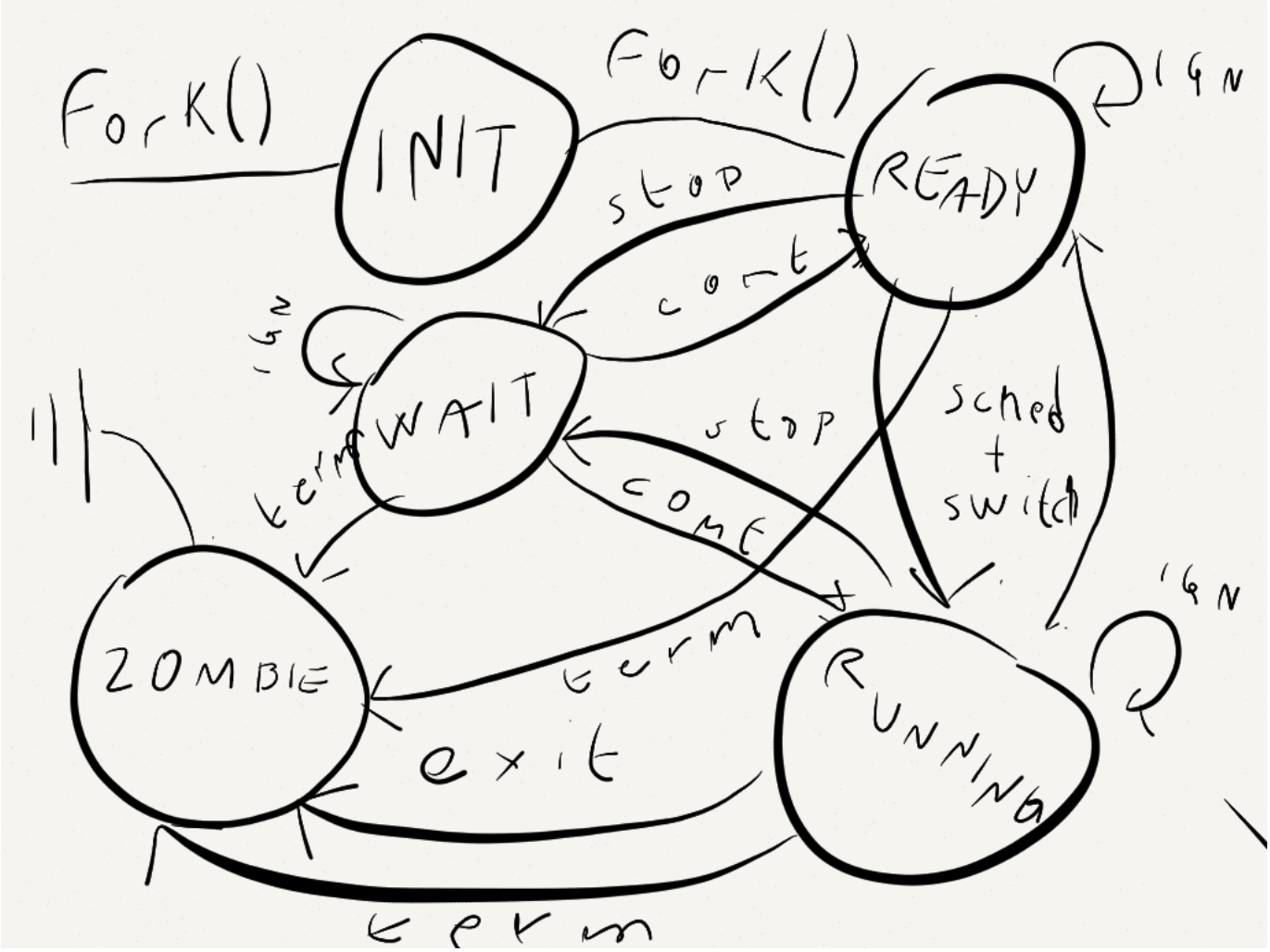
invocata, il loader del sistema operativo va a cercare il file specificato, e se è un eseguibile valido modifica i segmenti del processo che l'ha chiamata per prepararla all'esecuzione del nuovo processo. Questo include il ridimensionamento dei segmenti dati e codice adattandoli a contenere dati e codice del nuovo processo, e la loro corretta inizializzazione. In questo caso, l'esecuzione non dovrà cominciare da un punto precedente, ma dovrà ricominciare da zero. Il parametro argv corrisponde all'argv del main. Il parametro envp contiene delle informazioni di ambiente della shell, che possono essere informazioni di geolocalizzazione, lingua, ecc.
I file descriptor possono essere settati tramite flag (O_CLOEXEC) in modo da essere chiusi non appena viene invocata la execve.
Fork ed execve sono usate principalmente dalla shell: si duplica con una fork e trasforma il processo figlio nel processo che l'utente ha richiesto tramite execve.
Quando viene invocata la exit, il processo che l'ha invocata viene terminato, eliminando tutte le sue strutture dati, a meno che non siano ancora in uso (ad esempio, se l'applicazione stava usando un socket e l'invio dei dati non è ancora terminato). Un processo che ha invocato la exit e sta aspettando di poter essere terminato viene chiamato **zombie**. In questo stato il PID del processo è ancora bloccato, non può essere riassegnato.
La exit ha un parametro di tipo intero: un codice di terminazione che viene inviato al processo padre, che leggendo questo codice avrà informazioni sull'esito dell'esecuzione del processo figlio. Questo valore non viene inviato automaticamente al processo padre, come potrebbe essere nel caso di un'eccezione sollevata, ma è il processo padre che deve andarlo a leggere. Finché il processo padre non legge il codice di terminazione, il figlio rimane in stato di zombie. Per evitare sprechi di risorse è necessario che il padre legga il codice di terminazione il prima possibile.

5.1 Stato di un processo e segnalazioni

Un nuovo processo appena generato dalla fork è in stato **init**, e passerà in stato **ready** non appena i suoi segmenti di memoria saranno inizializzati.
Lo scheduler del sistema operativo decide, tra tutti i processi in stato ready, quale mandare in esecuzione. La fase in cui vengono cambiati i valori dei registri interni del processore per passare da un processo all'altro è detta **commutazione di processo** (**switch**). L'operazione inversa, cioè il passaggio dallo stato running allo stato ready, può avvenire sia per volontà del processo stesso, sia per ordine del sistema operativo, che assegna un tempo di esecuzione ad ogni processo, terminato il quale lo interrompe per eseguirne un altro. Il processo passerà dallo stato ready allo stato running per tutta la sua esistenza, finché non invoca la exit per terminare: passerà quindi in stato **zombie** finché il sistema operativo non decide che può terminare e lo elimina.

Segnalazioni
Un processo, per invocare la exit e terminare volontariamente, deve per forza di cose essere in stato running. Se voglio forzare un processo a terminare quando è in stato ready, posso farlo attraverso le **segnalazioni** (**signal**), una virtualizzazione del meccanismo di interrupt a livello di sistema operativo. Tramite la system call **`signal (int signum, sighandler_t handler)`** (deprecata, adesso si usa la più complicata **`sigaction`**) posso stabilire corrispondenze tra la segnalazione signum,un intero che indica al processo il motivo della segnalazione, e la funzione handler, da eseguire nel caso in cui venga inviata la segnalazione signum. È una funzione che non fa parte del normale flusso di esecuzione del programma.
Per inviare una segnalazione si usa la system call **`kill()`**. Con un handler programmato per eseguire la exit, la kill può essere facilmente usata per terminare un processo contro la sua volontà. Gli handler di default del sistema sono:

- **term**: invoca la exit terminando il processo
- **ign**: handler vuoto, ignora la segnalazione
- **core**: fa il core dump dello stato dell'applicazione
- **stop**: mette il processo in attesa (passa in stato **wait**)
- **cont**: riprende il processo che è attualmente in attesa (esce dallo stato wait)



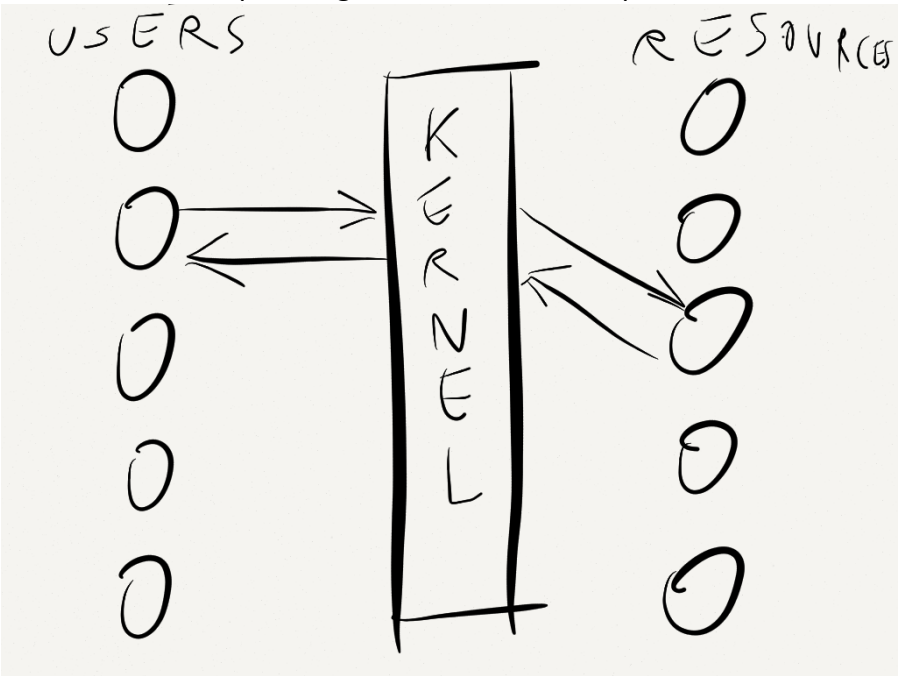
Schema dei passaggi di stato di un processo.

Alcune segnalazioni non possono essere ridefinite dal programmatore: ad esempio SIGKILL sarà sempre associata all'handler term.
La kill prende come parametro il PID del processo a cui inviare la segnalazione. Occorrono alcuni controlli di sicurezza però: ad esempio in un sistema multiutente non voglio che un utente possa terminare i processi di un altro (a meno che non sia root).
Un processo può essere lanciato in **foreground** o in **background**: un processo invocato in background (nella shell bash si fa col carattere & alla fine del comando) non può interagire col terminale che lo ha lanciato, un processo inviato in foreground invece ne prende il controllo.
Quando sto eseguendo un processo in foreground, la combinazione **CTRL+C** ha l'effetto di inviare una **`SIGKILL`** al processo. Se invece voglio mettere in attesa un processo posso usare la combinazione **CTRL+Z**, che invia una **`SIGSTOP`**.
L'utente tramite shell può inviare una **`SIGKILL`** ad un processo, tramite il comando **`kill`** *PID* del processo. per conoscere il PID dei processi in esecuzione c'è il comando **`ps`**.

Abbiamo detto che un processo rimane in stato zombie finché ha risorse ancora utilizzate o finché il padre non legge il suo codice di terminazione (il valore ritornato dalla exit). Esistono delle system call per leggere il codice di terminazione di un processo: **`wait()`** e **`waitpid()`**. Waitpid consente di specificare il PID di cui vogliamo sapere il codice di terminazione, la wait() di uno qualunque dei processi che il processo che la invoca ha mandato in esecuzione. Sono entrambe system call bloccanti.
Per evitare che un processo rimanga zombie per tanto tempo, esiste la segnalazione **`SIGCHLD`**, inviata da un processo che invoca la exit a suo padre, che alla **`SIGCHLD`** avrà associato un handler che esegue la wait, leggendo il codice di terminazione del figlio che quindi potrà terminare.
In pong server, possiamo provare a implementare questo sistema in modo che i processi generati dalla fork terminino.

6. Controllo degli accessi

Vediamo un modello molto semplificato di sistema operativo: un insieme di utenti vogliono accedere ad un insieme di risorse, e il nucleo di sicurezza del sistema deve intermediare queste richieste, filtrarle, perché gli utenti non devono potervi accedere direttamente.



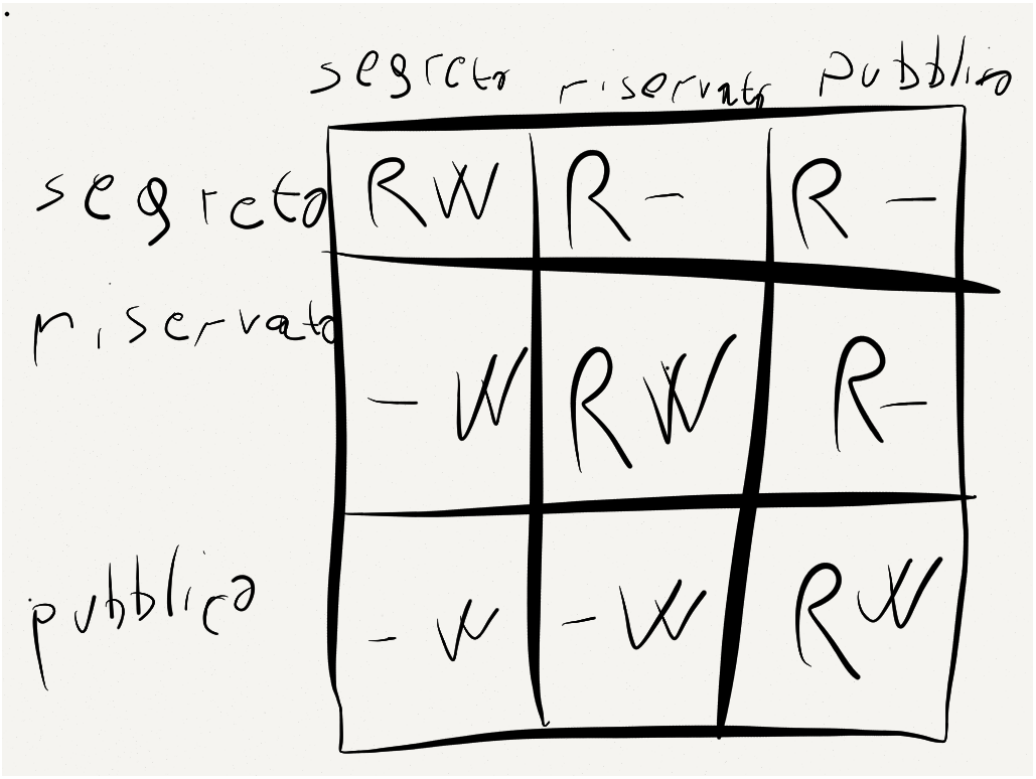
Schema di interazione utente-kernel-risorsa / risorsa-kernel-utente

Possiamo distinguere due casi: sistema aperto e sistema chiuso. In un sistema aperto c'è una quantità arbitraria di utenti che non possiamo conoscere a priori, che potrebbero arrivare in qualunque momento con richieste che non ci aspettiamo. In un sistema chiuso sappiamo da prima chi sono gli utenti del sistema e le loro possibili richieste.
In un sistema chiuso posso mantenere una matrice utenti-risorse. In questa matrice posso definire quali utenti possono accedere a quali risorse, e anche le modalità di accesso consentite (lettura, scrittura, esecuzione). In questo schema manca una fase cruciale: l'autenticazione. Al kernel infatti serve un modo per riconoscere gli utenti con assoluta sicurezza, non semplicemente fidandoci di quello che ci dicono. Servirà allora una **matrice degli accessi** (**access matrix**).

Un sistema aperto può esserlo solo per quanto riguarda gli utenti (non conosco a priori l'insieme degli utenti ma conosco quello delle risorse) o le risorse (non conosco a priori l'insieme delle risorse ma conosco quello degli utenti). Nel caso di sistema aperto dal punto di vista delle risorse userò una lista degli accessi, in cui ad ogni nuova risorsa aggiunta vado a definire i permessi di tutti gli utenti. Nel caso di sistema aperto dal punto di vista degli utenti, userò una **capability list**: ad ogni utente è associata una lista di permessi di accesso alle varie risorse. Adesso sembrano due modi equivalenti di gestire le cose: in realtà le differenze ci sono.

6.1 Assegnazione dei permessi: approccio mandatorio e approccio discrezionale

Abbiamo visto il modo in cui vengono gestiti i permessi, ma non come vengono stabiliti (cioè abbiamo visto la struttura dati che implementa il controllo degli accessi, ma non il suo contenuto). Esistono due approcci per l'assegnazione dei permessi: l'**approccio mandatorio** e l'**approccio discrezionale**. L'approccio mandatorio è quello più adatto ad un sistema chiuso: il progettista del sistema, in base alle esigenze di funzionamento, decide manualmente i permessi. L'approccio discrezionale invece delega all'amministratore di sistema la gestione dei permessi. A parità di spesa, potrebbe sembrare più vantaggioso il sistema mandatorio perché, essendo la sicurezza affidata al progettista, in caso di malfunzionamenti potrò rivalermi su di lui, cosa che non posso fare usando un sistema discrezionale in cui la responsabilità della sicurezza è dell'amministratore. I sistemi mandatori sono scelti solitamente da chi ha fortissime esigenze di sicurezza, uno dei maggiori utilizzatori di sistemi mandatori è il Department of Defense statunitense. Pensiamo ad esempio ad un sistema che deve gesitre gli accessi a documenti classificati con diversi livelli di segretezza, consentendo agli utenti di accedere solo ai documenti per cui hanno l'abilitazione. Le abilitazioni sono gerarchiche: supponendo di avere una classifica di segretezza di segreto, riservato e pubblico, chi ha accesso ai documenti segreti avrà accesso a tutto, chi ha accesso ai documenti riservati avrà accesso solo a documenti riservati e pubblici e così via. Ma potrei voler dare ad un utente con abilitazioni inferiori il permesso in scrittura per documenti che richiedono abilitazioni superiori: questo ad esempio potrebbe servire se ho necessità che un utente con qualifiche per riservato comunichi con utenti di livello segreto senza che gli altri utenti di livello riservato lo sappiano. Le qualifiche per la scrittura sono quindi inverse rispetto a quelle per la lettura: gli utenti di livello superiore possno leggere a tutti i livelli inferiori, ma non scrivere, Gli utenti di livello inferiore possono scrivere su tutti i livelli superiori, ma non leggerli. Questo per evitare fughe di informazioni da un livello più alto ad un livello più basso, in caso di utenti malintenzionati con qualifiche alte.



Buona configurazione del controllo degli accessi per un sistema di gestione delle informazioni classificate.

Questo sistema è (teoricamente) sicuro, ma non è efficiente, perché non c'è modo di passare informazioni, anche legittime, dai livelli superiori ai livelli inferiori. Immaginandolo in un sistema informatico militare, sarò protetto nel caso di generali traditori che vorrebbero passare al nemico le informazioni di livello più alto a cui hanno accesso, ma tutti i generali non potranno dare ordini perché non possono comunicare con i livelli inferiori. Non è inoltre sicuro in caso di attacchi dal basso in cui l'attaccante non è interessato a rubare le informazioni ma solo a corromperle: può farlo su tutti i livelli, avendone i permessi di scrittura.
E inoltre non è neanche completamente sicuro per quanto riguarda la segretezza, in quanto è violabile col sistema dei **canali coperti (covert channel)**, che sfrutta le risorse condivise da tutti gli utenti, come l'unità disco, per passare informazioni ai livelli inferiori. Questo perché, misurando il tempo di risposta, posso sapere se qualcun altro lo sta usando. Il generale traditore può stabilire con l'altra spia che accederà al disco per trasmettere il valore 1, non vi accederà per trasmettere il valore 0. Il ricevente in questo modo saprà che deve codificare 1 quando ha un tempo di accesso alla risorsa condivisa più alto, 0 quando è più basso. Non è un sistema veloce, il bitrate è molto basso.
Questo attacco può essere scongiurato evitando la condivisione di risorse tra gli utenti.
Abbiamo visto che l'approccio mandatorio non è per nulla pratico, e infatti nella realtà è molto poco utilizzato.

6.2 Principi di Denning

I principi di Denning sono una serie di regole per una buona gestione della sicurezza. Innanzitutto, bisogna partire dal presupposto che chi attacca è più furbo di noi.

1. **Anello debole:** se ci sono più modi in cui la sicurezza può essere compromessa, un attaccante furbo sceglierà il più semplice. La mia priorità sarà quindi sostituire i componenti peggiori, e non migliorare i componenti che già funzionano. E nella maggior parte dei casi, l'anello debole è l'uomo: sarà quindi più conveniente investire in corsi di formazione per amministratori di sistema piuttosto che in software aggiuntivi per la sicurezza.
2. **Minimo privilegio**: ad ogni utente vanno assegnati i minimi privilegi possibili per consentirgli di accedere alle risorse di cui ha bisogno nelle modalità di cui ha bisogno, e solo a quelle: devo essere il più restrittivo possibile nell'assegnazione dei privilegi, nessuno deve avere accesso a risorse non indispensabili in modalità non indispensabili. Questo mi protegge non solo da attacchi intenzionali ma anche da errori involontari.
3. **Cambio di contesto:** gli utenti possono dover svolgere compiti diversi in momenti diversi. Quindi in momenti diversi assegnerò all'utente permessi diversi, a seconda del contesto attuale, in modo da avere sempre assegnato il minimo priviliegio possibile in relazione non a tutto quello che devo fare in generale ma da quello che sto facendo al momento. È il principio del minimo privilegio applicato sul tempo.
4. **Economia/ridondanza di controllo**: la ridondanza dei sistemi di controllo mi garantisce una maggiore sicurezza: se uno fallisce, c'è sempre l'altro, e le possibilità di violazioni si riducono.

6.3 Permessi in UNIX

Non tutti i filesystem supportano la gestione dei permessi.
Nei sistemi UNIX esiste un utente **root**, con UID 0, che è proprietario di tutti i file.
L'utente proprietario del file, a meno di modifiche manuali, è quello che lo ha creato. Quando si crea un file gli devono anche venire attribuiti dei permessi, stabiliti per default per ogni nuovo file creato (impostazioni di default modificabili dall'utente). Una volta creato, l'utente potrà modificarne i permessi col comando **`chmod`**. Chmod prende come parametro una lettera **u** (user), **g** (group) od **o** (other), che indica l'utente o gruppo a cui vogliamo attribuire i permessi, + o - per dire se stiamo dando o togliendo un permesso e le lettere **r**,**w**,**x** per specificare su quale permesso stiamo agendo.
Il proprietario del file ha un permesso in più: quello di modificare i permessi sul file, ed è un privilegio che non può essere tolto. Quindi anche in caso un utente proprietario di un file si tolga il permesso di scrittura, potrà sempre ridarselo.
Perché un utente dovrebbe volersi togliere dei permessi di accesso al file? Per il secondo e terzo principio di Denning: minimo privilegio e cambio di contesto, per una sicurezza ottimale i privilegi vanno assegnati a seconda delle necessità del momento e devono essere sempre i minimi possibile. Un approccio simile mi protegge non solo dagli attacchi, ma anche dagli errori di programmazione delle applicazioni: minori sono i permessi con cui l'applicazione buggata è eseguita, minori sono i danni che potrà fare.
I permessi di accesso ai file sono discrezionali, il permesso di modificare i permessi è mandatorio: lo può fare il proprietario e basta, questa scelta non può essere modificata. Posso però cambiare il

proprietario del file con **chown**: in questo modo il proprietario originale perde il privilegio di decidere i permessi sul file in favore del nuovo proprietario. Questo può essere fatto solo da root. La presenza dell'utente root è la più grossa vulnerabilità dei sistemi UNIX: è infatti una palese violazione del principio di Denning del minimo privilegio (è un utente con tutti i privilegi, e non gli possono essere tolti senza modifiche steutturali al sistema operativo), e rischia di trasformarsi anche in anello debole: un attaccante infatti tenterà ovviamente di ottenere l'accesso come root. È infatti buona norma non usare mai l'utente root per la normale attività, ma solo quando i suoi privilegi sono necessari.

6.4 Controllo degli accessi role-based: utenti e gruppi

In UNIX ad ogni utente è assegnato un **gruppo** di appartenenza. Che non è necessariamente solo uno: un utente può essere membro di più gruppi.
Nel file **/etc/passwd** sono contenute tutte le informazioni degli utenti (ma non più le password, ora memorizzate in /etc/shadow). Sono organizzate in linee, suddivise in campi: **name:x:UID:GID:home directory:shell** . La x mi indica se l'utente è protetto o no da password: una volta conteneva anche la password in chiaro, ma ovviamente la cosa non era sicura.
Il file **/etc/group** ha struttura simile, ma riferita ai gruppi invece che agli utenti. Contiene tutti i gruppi, completi di GID, e tutti gli utenti che ne fanno parte.
Di default ad ogni creazione di un nuovo utente viene creato anche un nuovo gruppo. Il nuovo utente apparterrà a questo gruppo e basta. Posso in seguito aggiungerlo ad altri gruppi e crearne di nuovi, come ad esempio un gruppo backup i cui utenti hanno permesso in lettura a tutti i file, poiché devono fare il backup del sistema. Posso creare gruppi per dare l'accesso solo a certi utenti a determinate risorse (come un gruppo autorizzato a stampare).
Questi file di configurazione possono di norma essere modificati solo da root: posso però creare un gruppo di utenti delegati alla creazione di nuovi utenti. Questo per il principio del minimo privilegio che mi impone di usare l'utente root il meno possibile.
Abbiamo detto prima che chmod può agire anche sui gruppi, col parametro g: i comandi dati con questo parametro si riferiranno al gruppo proprietario del file in questione. Per modificare il gruppo proprietario di un file, c'è il comando **chgrp**.

Quando io eseguo un programma o uno script, questo avrà gli stessi permessi di accesso alle risorse dell'utente che lo ha lanciato. Se assegno il privilegio **s** ad un file eseguibile, questo verrà eseguito coi permessi del suo proprietario anche se a lanciarlo è un altro utente (al processo viene associato l'UID del proprietario e non dell'utente che lo ha lanciato). Questo può eseere fatto anche dall'utente root.
Il bit s può essere anche associato ad un gruppo, con la sintassi chmod g+s: serve ad utilizzare l'utente root il meno possibile.

6.5 sudo

Un altro modo per amministrare un sistema senza dovervi accedere come root è **sudo**: esegue un comando con permessi di root anche se lanciato da un altro utente. Mantiene le sue configurazioni in /etc/sudoers, che mi indica quali utenti sono autorizzati chiamare il comando sudo e i comandi che possono essere invocati tramite sudo. Può essere configurato per richiedere una password: non quella di root, ma dell'utente che lo chiama. Chiedere nuovamente la password ad un utente già autenticato è una doppia sicurezza per accertarmi che sia veramente lui e non qualcuno che si è impossessato del terminale dopo il login (l'utente legittimo ha lasciato la

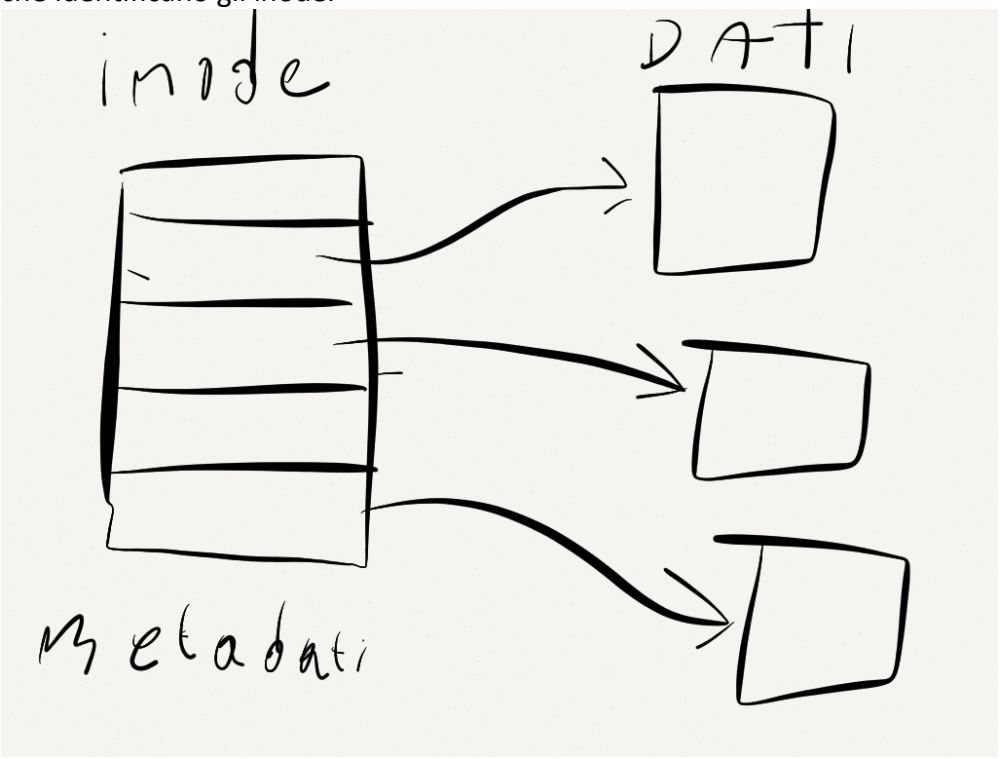
postazione incustodita). È anche un modo per rendere l'utente consapevole che è stato chiamato sudo, se ad esempio il comando è invocato dentro uno script.

7. File system

Vedremo le caratteristiche di ext2, primo filesystem sviluppato appositamente per Linux. Compito di un filesystem è di memorizzare file di diversi tipi. Ci sono 5 principali tipi di file:

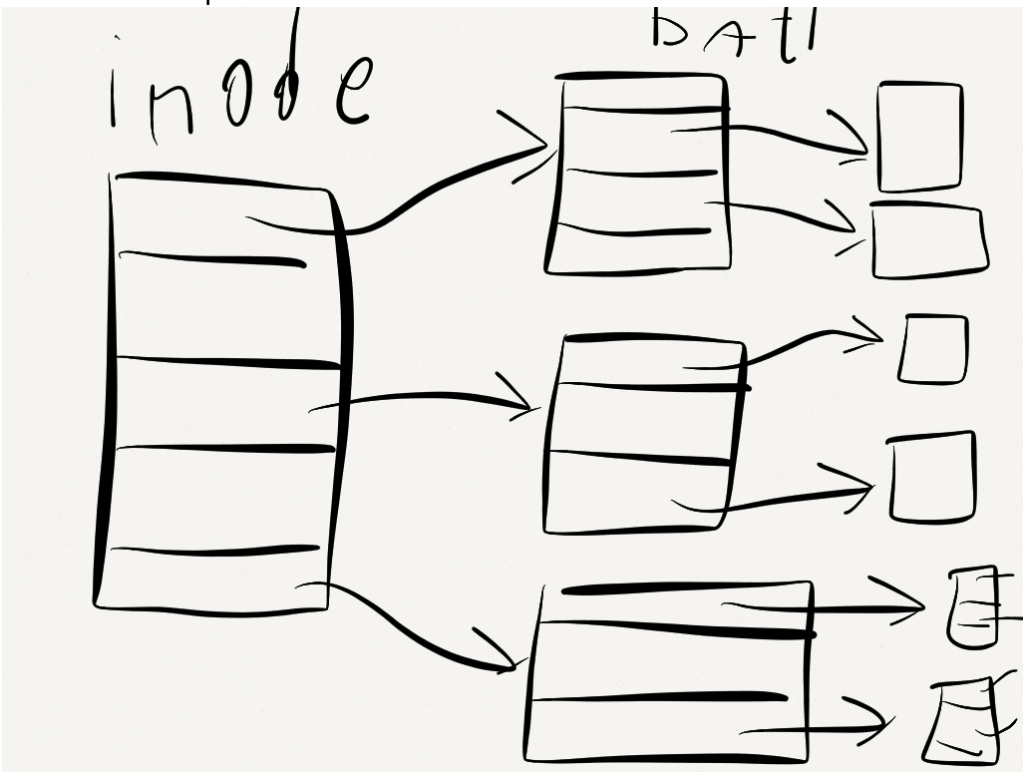
- **regolari**: possono contenere dati di qualunque tipo
- **directory**: un file che contiene altri file. La differenza con i file regolari è che, mentre i dati di un file regolare sono arbitrari e il sistema operativo non se ne interessa, il contenuto delle directory è gestito dal sistema operativo.
- **link (simbolici)**
- **speciali:** si dividono in file a blocchi (block) e file a caratteri (char), usati per accedere a dispositivi
- **pipe**, permettono la comunicazione tra processi

I file regolari sono indirizzati da una struttura dati detta **inode**, che contiene i metadati del file: informazioni come dimensione, maschera degli accessi, date di creazione, modifica, ultimo accesso e soprattutto i **puntatori ai blocchi di dati.**I dati del file infatti sono suddivisi in blocchi, la cui dimensione varia da un filesystem all'altro. L'inode memorizza gli indirizzi di questi blocchi.
L'inode viene identificato con un numero: esiste una inode table, un array i cui indici sono i numeri che identificano gli inode.



Indirizzamento diretto inode->blocchi dati

Per avere una memorizzazione efficiente dei file si usano più livelli di indirizzamento. All'interno degli inode abbiamo una quantità ridotta di puntatori ai blocchi di dati. Se il file di corto è finita qui: tutti i suoi blocchi di dati sono indirizzati dai puntatori dell'inode. Se il file è più grande entra in gioco l'**indirizzamento indiretto**: il blocco dati puntato dall'inode contiene a sua volta puntatori ad altri blocchi dati. È un modo furbo di moltiplicare i dati indirizzati da un singolo inode, ma il suo svantaggio è di aumentare i tempi di accesso, poiché dovrò attraversare più livelli di indirizzamento prima di arrivare ai dati. La dimensione degli indirizzi può essere 32 o 64 bit, a seconda dell'implementazione.



Indirizzamento indiretto inode->blocchi di indirizzamento->blocchi dati

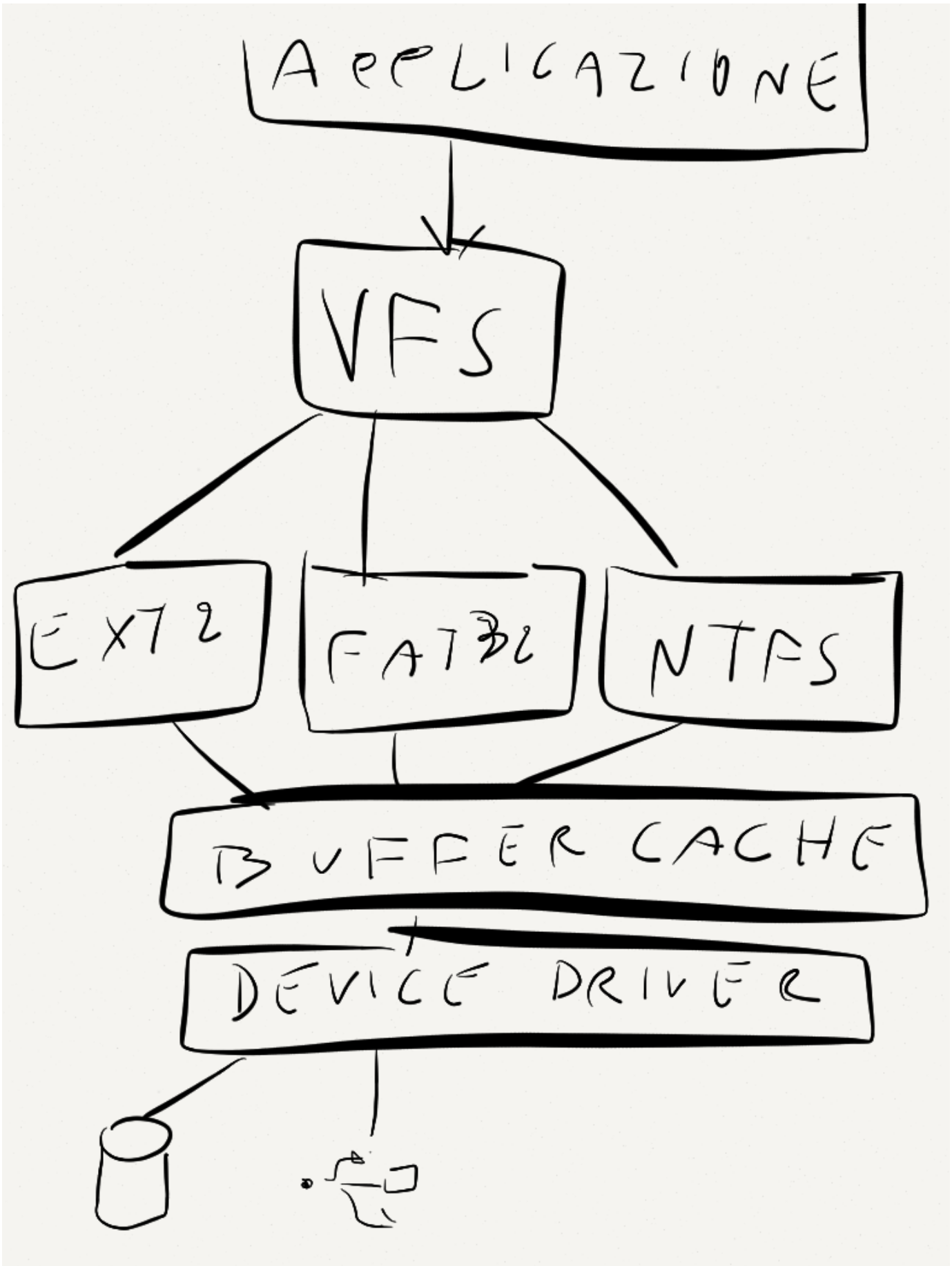
Come si decide la dimensione dei blocchi di dati? Usando blocchi troppo piccoli avrò bisogno di più puntatori, e saranno peggiorati i tempi di accesso. Usando blocchi troppo grandi sprecherò spazio. La dimensione giusta è quindi un compromesso tra le esigenze di spazio e di efficienza. Non posso usare una soluzione dinamica, decidendo volta per volta la dimensione dei blocchi, perché peggiorerebbe di molto l'efficienza.

Directory

Le directory contengono le associazioni tra i nomi dei file regolari al loro interno e i numeri degli inode che lo compongono. Il sistema deve sapere qual'è la directory radice, dalla quale si possono raggiungere tutte le altre.

VFS

Non è infrequente che un sistema debba gestire più dispositivi di memoria di massa con filesystem diversi. Il VFS (Virtual File System) è un livello di astrazione interno al kernel che permette di gestire più filesystem, nonostante ognuno di questi memorizzi i file in modo diverso. Questo è indispensabile per avere un uso semplificato delle system call: è grazie al VFS che possiamo chiamare la open su qualunque file senza doverci preoccupare di come questo file è stato memorizzato dal filesystem.


APPLICAZIONE
VFS
EXT2
FAT32
NTFS
BUFFER CACHE
DEVICE DRIVER

7.1 Dispositivi a blocchi

Un file speciale a blocchi è la virtualizzazione di un device driver. È detto a blocchi perché vi posso leggere e scrivere blocchi di dati da e verso il dispositivo. In un device a blocchi abbiamo una serie di blocchi numerati di dati, di dimensioni uguali e predefinite. Tramite un device driver io posso chiedere di leggere e scrivere su un blocco. Un file system memorizza in questi blocchi gli inode e i blocchi di dati dei file, e ogni file syetem ha un modo diverso di farlo.
A livello fisico, su un disco dopo il boot sector sono memorizzati una serie di gruppi, ognuno dei quali è diviso in sei parti:

- **superblock**: una serie di dati che dicono qual'è lo stato del file system e ne permettono il mount
- **file system descriptor:** dice al sistema operativo quale file system stiamo usando

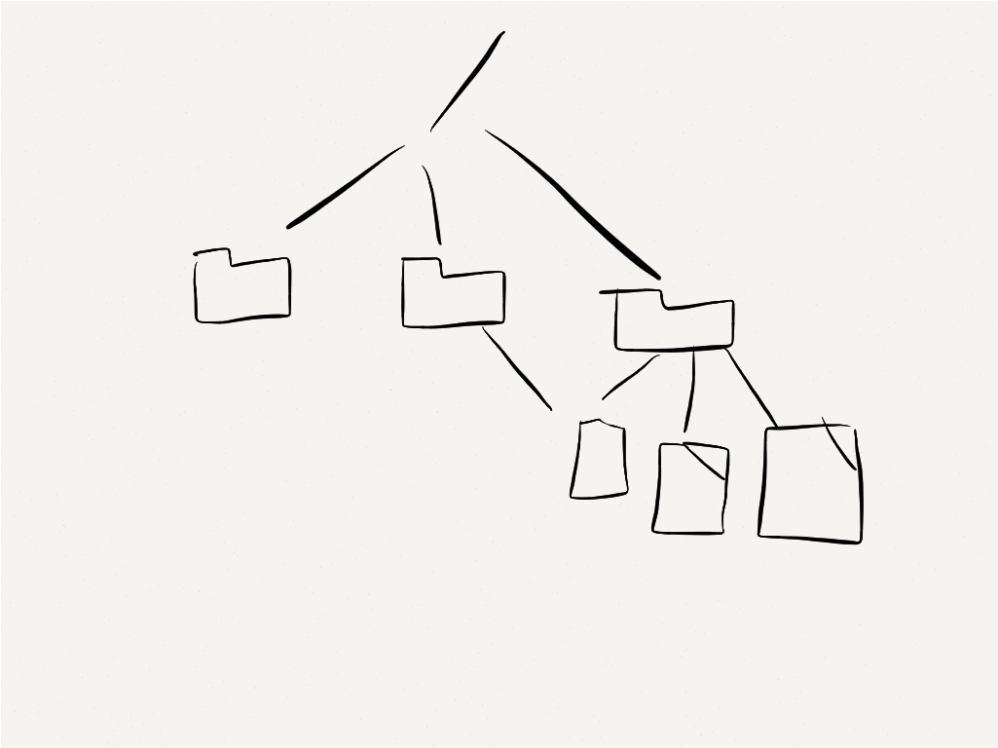
Questi due blocchi sono fondamentali, se si corrompono non posso più accedere all'unità. Proprio per questa loro criticità vengono replicati in ogni gruppo: è un meccanismo di ridondanza per cui se uno di questi blocchi diventa inutilizzabile non perdo tutta l'unità disco.

- **block bitmap**: associa un bit ad ogni blocco per sapere se è usato o no
- **inode bitmap:** associa un bit ad ogni inode per sapere se è usato o no
- **inode table:** sequenza degli inode
- **data blocks:** blocchi di dati dei file

Queste strutture non sono replicate: ogni gruppo ha i suoi blocchi e inode.
Anni fa si tentava di mantenere i blocchi dati il più vicino possibile agli inode, per ottimizzare i tempi di accesso, ma oggi gli hard disk sono talmente sofisticati che il loro comportamento è scarsamente predicibile dal software.
Questo livello di organizzazione di gruppi e blocchi è tipico di ext2, mentre la struttura in inode e blocchi è comune a molti file system, poiché è necessaria perché possano essere virtualizzati dal VFS.

7.2 Link fisici

Abbiamo visto che il file system ha una struttura gerarchica delle directory, un albero le cui foglie sono file. Per accedere ad un file, dobbiamo prima vedere nella directory che lo contiene l'associazione tra il suo nome e i suoi inode. Un link non è altro che un'associazione tra una stringa di caratteri (il nome del file) e gli indici dei suoi inode. Se vogliamo cambiare il nome di un file, o metterlo in una directory diversa, non sarà necessario agire sugli inode o sui blocchi dati ma solo sull'associazione nome-inode. Se voglio rinominare, basta cambiare la stringa associata agli inode. Senza bisogno di cancellare il nome precedente: posso avere lo stesso inode associato a più nomi diversi, e non cambierà nulla accedervi con un nome o con l'altro: tutti gli attributi (permessi, proprietario ecc.) saranno gli stessi, poiché sono informazioni contenute nell'inode. Posso anche avere due link in due directory diverse che puntano allo stesso file.



Uno stesso file ha due link in due directory diverse

Perché un file (inode) sia accessibile, è necessario che nel file system ci sia almeno un link che punta il suo inode. Quando cancelliamo un file, in realtà eliminiamo solo il link specifico, l'associazione nome-inode in una specifica directory. Non si sta agendo su inode e dati, ma solo sul link. Nell'inode è presente un contatore del numero di link che lo puntano in tutto il file system, e i dati verranno eliminati solo quando non ci saranno più link a rendere il file accessibile.
La creazione di un file comporta la creazione di un inode e di un link. Durante l'esistenza del file posso creare altri link per accedervi, con il comando **`ln`**, che prende in input il nome di un link già esistente e il nome del nuovo link. Quando lo voglio eliminare, con il comando **`rm`** elimino uno specifico link. L'applicazione rm poi controlla il contatore dei link all'inode, e se è a 1 ne elimina anche i dati.

7.3 Link simbolici

Questo schema di accesso semplifica la condivisione di file tra utenti e applicazioni diverse. Ma cosa succede in un sistema in cui ho più file system montati in un'unica struttura? Avrò diverse numerazioni degli inode per ogni file system, quindi non potrò avere link da uno all'altro, a meno che io non usi i **link simbolici**.
I link simbolici sono un livello di astrazione più alto che mi permette riferimenti incrociati da un filesystem all'altro: per la creazione dei link infatti non viene usato l'inode, ma un nome (pathname). Questo pathname farà riferimento ad un link in un altro file system, e andando a vedere quello avrò il numero di inode che mi permette di accedere al file.
Un link simbolico è un tipo speciale di file: il suo inode fa riferimento a blocchi di dati che contengono una stringa di caratteri: il pathname assoluto del file puntato dal link. Vengono creati aggiungendo l'opzione **`-s`** al comando **`ln`**.
Parecchie situazioni possono causare errori: ad esempio se il pathname puntato dal link simbolico non esiste, o se il file system che contiene il file non è ancora montato (facile se il file system in cui si trova il file è un dispositivo rimovibile). Con il comando **`ls -l`** i link simbolici ci verranno visualizzati nella forma **nome -> pathname**.

7.4 Consistenza delle informazioni

Abbiamo visto che, in ext2, ogni blocco contiene alcune informazioni ridondanti, le bitmap di inode e blocchi dati, l'insieme degli inode e l'insieme dei blocchi di dati.
Per creare un file avrò bisogno di:

- l'inode della directory che lo contiene
- l'inode del nuovo file
- almeno un blocco dati

Dovrò marcare come utilizzati nelle bitmap l'inode e il blocco desiderati: nell'inode del file vi assocerò il suo blocco dati, e nella directory che lo contiene creerò un link inode-nome file. Cosa succede se la macchina si spegne durante la creazione di un file? Dipende dall'ordine con cui faccio le operazioni: se per prima cosa segno le bitmap avrò dei blocchi e degli inode marcati come utilizzati, ma che di fatto non lo sono; se invece prima memorizzo i dati, li indirizzo sull'inode e creo il link, avrò un file salvato correttamente i cui blocchi e inode sono marcati come spazio libero dalle bitmap.
In fase di mount di un file system posso specificare se montarlo in sola lettura (read only) o in lettura e scrittura (read write). Questa informazione è memorizzata tra le informazioni ridondanti. Se un file system è montato con permessi di scrittura, tra le informazioni ridondanti viene aggiunto il flag dirty, che indica che il file system è a rischio di inconsistenza. Questo flag viene rimosso al momento del corretto spegnimento della macchina. In caso di spegnimento improvviso, all'avvio troverò ancora il flag dirty impostato, per cui saprò che il file system non è stato chiuso correttamente e potrebbe avere inconsistenze, e avvierò quindi un'applicazione di controllo e ripristino, che in UNIX è **fsck**. Questa applicazione tenta di ripristinare la consistenza dei dati, operazione che può essere fatta in tanti modi diversi: si può innanzitutto controllare se la struttura del file system dal punto di vista applicativo (albero delle directory) è coerente con le informazioni contenute nelle varie bitmap. Questa operazione richiederebbe tempi spropositati su file system di grandi dimensioni se non fosse implementata in modo estremamente ottimizzato. Richiede comunque alcuni minuti, per cui viene fatta il meno possibile: ogni volta che trovo il flag dirty all'avvio e ogni tot montaggi del file system, per controllo. Il contatore dei mount senza controlli viene memorizzato tra le informazioni ridondanti.
Fino a una decina di anni fa, se fsck trovava file di cui si era perso il link li metteva, con un nome predefinito, nella directory /lost+found.

7.4.1 Journaling

È più efficiente, piuttosto che eseguire periodicamente una lunga procedura di controllo di consistenza, avere un meccanismo che mi garantisca la consistenza dei dati sempre. È questo lo scopo del journaling, che consiste in:

- scrivere un elenco completo delle modifiche che voglio fare, prima di farle, e ci aggiungo un checksum per controllarne l'integrità. È importante che il checksum venga scritto per ultimo.
- effettuare le modifiche

Se le operazioni si interrompono a metà, fsck può prendere l'elenco delle modifiche scritto all'inizio, controllare quali sono state effettivamente compiute e fare quelle che ancora rimangono in sospeso. Che succede se l'interruzione è avvenuta durante la scrittura del journal? Me ne accorgerò subito perché mancherà il checksum, quindi mi basterà eliminare il journal, perché sono

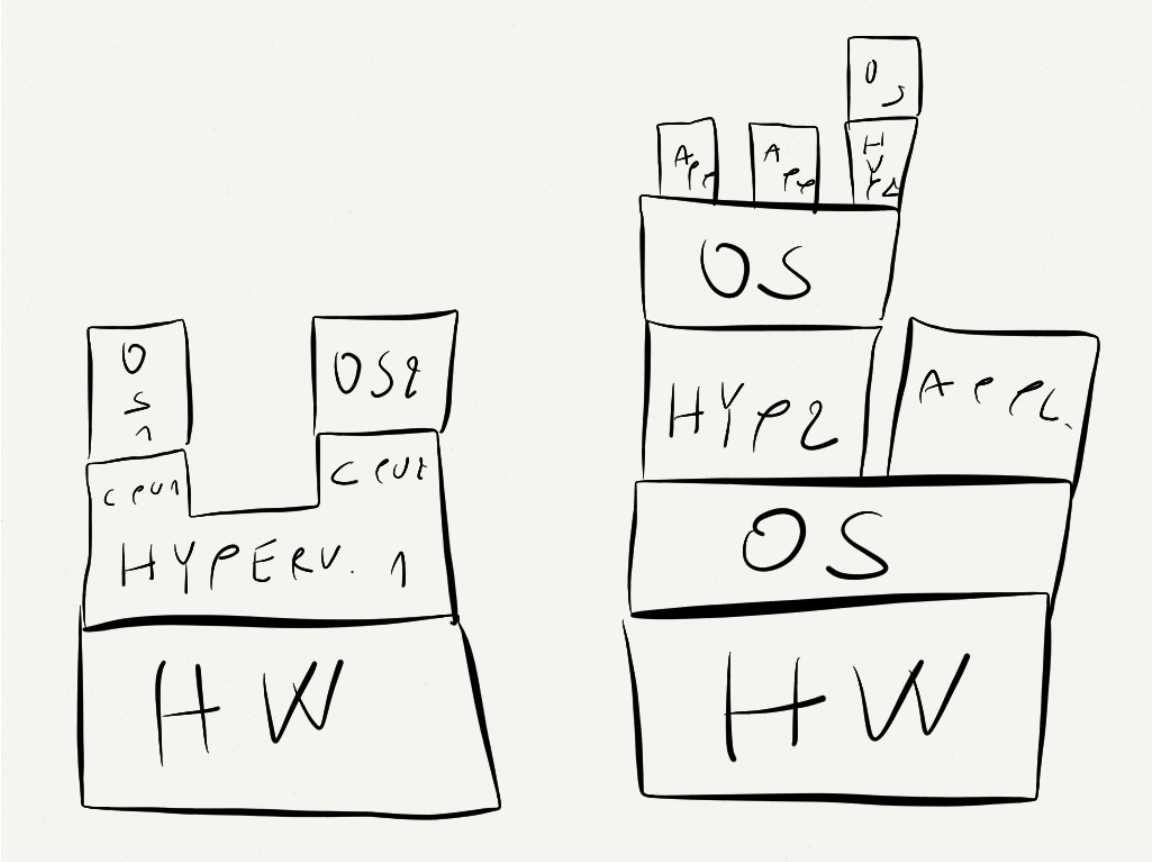
sicuro che non è stata apportata nessuna modifica dato che la procedura impone che si inizi ad operare sul file system solo dopo aver scritto journal e checksum.
C'è un problema di efficienza però: se per ogni operazione devo scrivere il journal rischio di raddoppiare i tempi di accesso all'unità. Quando si sceglie il file system da usare, bisogna valutare se è più importante l'efficienza o l'affidabilità per decidere se usare un file system con o senza journaling. C'è anche un fattore di usura dei dispositivi da considerare: ad esempio le memorie flash hanno un numero limitato di scritture che posso farci prima che si rompano, e sarà quindi preferibile usarvi un file system senza journaling.
Ext2 non implementa il journaling, che è invece usato da ext3 ed ext4.

8. Virtual machine

L'idea di virtual machine nasce insieme all'esigenza di eseguire più applicazioni contemporaneamente, ai tempi in cui le macchine erano ancora monoprogrammate.
Si tratta di avere un hypervisor al di sotto del sistema operativo che virtualizza le risorse per permettere di avere più sistemi operativi in esecuzione contemporaneamente. Ogni sistema operativo avrà la sua porzione di tempo CPU, RAM e memoria di massa, per cui crederà di essere eseguito su una machina tutta sua, e le vaire macchine vituali eseguite su una stessa macchina fisica non possono interagire tra loro. Questa caratteristica aumenta la sicurezza del sistema perché, essendo ogni virtual machine isolata dalle altre, se un'applicazione danneggia il sistema operativo il danno sarà limitato alla VM nel quale è eseguito e non all'intero sistema.
È stata una soluzione ideata da IBM per permettere ai suoi sistemi operativi di restare al passo coi tempi, offrendo la possibilità di eseguire più applicazioni sullo stesso sistema senza bisogno di riprogrammarlo da capo per implementare il multitasking. L'implementazione del multitasking nei sistemi operativi moderni ha soppiantato questa soluzione, che però rimane valida in caso di macchine grandi e costose. In molti casi infatti, per ridurre i costi di gestione, è conveniente avere una sola macchina molto potente su cui far girare molte VM, piuttosto che tante macchine a costo ridotto. Su una macchina di questo tipo avrò in esecuzione tante macchine virtuali più piccole, ma con l'affidabilità e la sicurezza della macchina fisica che le esegue. I datacenter affittano queste VM ai clienti, che dovranno occuparsi solo di usarle, senza preoccuparsi della manutenzione. Ad oggi, la maggior parte dei servizi Web è erogata da macchine virtuali

8.1 Hypervisor

Ci sono due tipi di ipervisori: il tipo 1 è quello descritto in precedenza che sta sotto al sistema operativo, il tipo 2 viene invece eseguito da un altro sistema operativo.



Ipervisori di tipo 2 sono usati tipicamente su personal computer per avere più sistemi operativi a disposizione, quelli di tipo 1 invece vengono usati sulle grandi macchine in uso dai datacenter.

Emulatore
L'ipervisore può anche emulare un'architettura hardware diversa per le macchine virtuali che esegue. Questo può servire ad esempio nello sviluppo di software, se voglio provare il codice su un'architettura diversa da quella della mia macchina fisica. Anche questa soluzione è stata ideata da IBM, per il passaggio dal System/360 al System/370. La nuova architettura hardware prevedeva delle istruzioni in più, e per questo il sistema operativo andava riscritto. Ma a quei tempi i sistemi operativi si scrivevano in assembly, e per iniziare lo sviluppo (che avrebbe richiesto alcuni anni) avrebbero dovuto aspettare di avere a disposizione le nuove macchine. Decisero quindi di emulare l'architettura del System/370 sui System/360 che già avevano e iniziare lì la programmazione del nuovo sistema.

Realizzazione di un hypervisor
Il sistema operativo eseguito da una macchina virtuale non si rende conto di essere su una VM, perché l'hypervisor implementa a livello software tutte le funzioni offerte dall'hardware: memoria virtuale, trap e interrupt, modalità privilegiata. Se la mia macchina fisica non implementa queste funzioni, non posso virtualizzare.
Quando io avvio una macchina su cui è installato un ipervisore di tipo 1, questo fa le stesse operazioni che farebbe un normale sistema operativo: predispone la memoria virtuale, il trap

handler ecc. Una volta terminato l'avvio e suddivise le risorse per le varie macchine virtuali, il processore ritorna in modalità utente. Ma i sistemi operativi delle macchine virtuali vorranno poter eseguire istruzioni privilegiate, e quando cercheranno di farlo scatterà una trap gestita dal trap handler dell'ipervisore, programmato per eseguire lui stesso le operazioni in modalità privilegiata e non lasciarle eseguire ai sistemi guest. Quando l'esecuzione del trap handler terminerà, il sistema operativo della VM sarà all'istruzione successiva, e troverà l'operazione privilegiata che ha richiesto già eseguita dal trap handler. Questo peggiora notevolmente l'efficienza dell'esecuzione di istruzioni privilegiate, poiché invece di essere eseguite direttamente vengono gestite da un trap handler.
POSIX virtualizza le trap col meccanismo delle signal, definite a livello applicativo. Le signal però non sono state pensate per la gestione di ipervisori, quindi alcuni hypervisor di tipo 2 richiedono modifiche al sistema operativo che li esegue.

8.2 Rootkit

Un rootkit è un software che altera il funzionamento del sistema per nascondere il suo comportamento reale alle applicazioni. Può essere utilizzato sia per difendersi dagli attacchi sia per farli. Un modo per implementare un rootkit è proprio con una macchina virutale. Un rootkit difensivo può essere implementato come un hypervisor, che controlla se il sistema operativo si sta comportando come dovrebbe.
Un rootkit ostile invece può essere un malware che, una volta preso il controllo del sistema operativo installa un hypervisor che, nascondendo il suo funzionamento al sistema operativo, è al riparo da eventuali patch che non avrebbero più consentito il suo funzionamento se fosse stata eseguita come normale applicazione.

9. Standard ASN.1

ASN.1 è uno standard (ITU X.680) per definire regole sintattiche usate per descrivere dati che devono essere condivisi con più entità presenti in rete, senza preoccuparci di come verranno elaborati.
Esempio: vogliamo descrivere i dati personali dei dipendenti di un'azienda.

```
PersonnelRecord::=[APPLICATION 0] SET
{
          name Name;
          title VisibleString;
          number EmployeeNumber;
          dateOfHire Date;
          nameOfSpouse Name;
          children SEQUENCE OF ChildInformation DEFAULT{}
}
```

È molto significativa la differenza tra maiuscole e minuscole: le parole che iniziano con la maiuscola sono tipi, quelle con la minuscola sono valori, mentre quelle tutte in maiuscolo sono parole chiave.
SET: indica qualcosa di simile alla struct del C, come possiamo vedere nell'esempio: un'insieme di campi racchiusi tra parentesi graffe. Questi campi sono rappresentati con la sintassi *nome Tipo*.
SEQUENCE OF: simile ad un vector del C++: un array, di dimensioni variabili.

`DEFAULT {}` indica che il campo non deve essere necessariamente definito: nel caso in cui non venga specificato verrà usato come valore di default la lista vuota.
I tipi all'interno del SET dovranno essere a loro volta definiti:

```
Name ::= [APPLICATION 1] SEQUENCE
{
            givenName VisibleString;
            initial VisibleString;
            familyName VisibleString;
}
```

Tra le parentesi quadre sono racchiuse opzioni che non verranno esaminate dall'analizzatore sintattico. È qualcosa di simile ad un commento, ma con più rigore: non posso metterci qualunque cosa, devono essere richiami a funzionalità aggiunte dall'utente (in questo caso una APPLICATION che richiede un parametro numerico).

```
EmployeeNumber::=[APPLICATION 2] INTEGER;
```

In questo caso EmployeeNumber è un numero intero, che è un tipo base: richiamo comunque APPLICATION con parametro 2, magari per controllare se il valore rientra in un range di valori consentiti.

```
Date::=[APPLICATION 3] VisibleString -- YYYY MM DD
```

Dopo il -- c'è un commento, che indica il formato con cui rappresentiamo la data.

```
ChildInformation::=SET
{
       name Name;
       dateOfBirth date;
}
```

Possiamo anche definire esplicitamente i dati, con la sintassi `{ name { givenName "John", initial "P", familyName "Smith" }, title "Director"`

ASN.1 prevede un buon numero di tipi base: non solo interi e stringhe, ma anche numeri reali, booleani e date (in formato UTC). Supporta anche qualcosa di simile agli enum del C, gli **ENUMERATED**, con cui posso definire ad esempio i giorni della settimana:
`ENUMERATED { domenica(0), lunedi(1),... }`
Meritano un discorso a parte gli spazi: a volte sono significativi e a volte no, e la cosa non è così immediata (non se la ricorda manco Chiola o.o), bisogna consultare la documentazione per sapere quando gli spazi contano e quando no.
Si possono inoltre definire set di caratteri non standard e sistemi di compressione (con PACKED).

9.1 Rappresentazione binaria

Finora ci siamo preoccupati di descrivere i dati in un modo che sia indipendente da tutto: linguaggi, macchine, implementazioni. Ma servirebbe a ben poco senza un'implementazione pratica: le informazioni scritte attraverso queste regole vanno codificate in forma binaria, e va fatto in maniera efficiente, con il minor utilizzo di spazio possibile. Per fare questo sono stati definiti altri standard che specificano il formato di rappresentazione binaria, descritti nel documento ITU X.690. L'ultima revisione di questi standard è del 2002. CER e DER hanno una rappresentazione canonica: ogni valore può essere rappresentato in un modo solo e quindi è possibile fare un confronto di uguaglianza direttamente sulla stringa binaria. E' molto importante

dal punto di vista pratico, e questa fa si che nella maggioranza dei casi venga utilizzato il formato DER.

9.1.1 BER (Basic Encoding Rules)

Lo standard BER (Basic Encoding Rules) è uno di questi standard, e permette di codificare i dati in diversi formati. I campi ID, LENGTH e CONTENT sono comuni a tutti, ma se voglio inviare dati di lunghezza variabile posso usare un valore speciale per il campo LENGTH e terminare con un byte di End of content (EOC): serve a capire quando è finito il file visto che la lunghezza non è specificata nel campo LENGTH. Questa soluzione però mi impedisce di avere il byte di EOC all'interno del contenuto del file, perché verrebbe interpretato come terminatore invece che come contenuto.
Il campo ID invece è un byte suddiviso in tre parti: due bit di CLASS più significativi che descrivono una classe di identificatori. Abbiamo quattro classi possibili:

UNIVERSAL	0 0
APPLICATION	0 1
CONTENT-SPECIFIC	1 0
PRIVATE	1 1

I bit application, content specific e private offrono la possibilità di estendere l'uso di questo formato di rappresentazione per applicazioni particolari. Il bit P/C mi indica se il tipo di dato è primitivo o definito. I 5 bit rimanenti sono chiamati TAG.
Come si può fare a codificare un valore di tipo booleano? Il primo byte assume valore 1: sette zeri seguiti da un uno. La lunghezza vale sempre 1, il contenuto, di un solo byte, conterrà o 0 (falso) o FF (vero). FF è la rappresentazione esadecimale di 8 uni, si usa questa soluzione perché l'unità minima allocabile è il byte, non posso usare un solo bit per rappresentare vero e falso.
Per codificare un valore di tipo intero invece possiamo usare una quantità di byte proporzionale al valore che vogliamo rappresentare: può bastarci un solo byte per numeri piccoli, per numeri più grandi ce ne serviranno di più. Usare la lunghezza fissa o variabile è una scelta d'implementazione: la lunghezza fissa è più efficiente perché usa un byte in meno, la lunghezza variabile invece usa un byte in più ma è più semplice da calcolare, perché non richiede all'applicazione che la usa di sapere a priori qual'è la lunghezza del contenuto. Tornando alla rappresentazione degli interi, se scegliamo di usare la lunghezza variabile possiamo mettere nel campo LENGTH il valore di lunghezza variabile, scrivere gli ottetti che compongono il numero nel campo CONTENT e quando abbiamo finito scrivere il byte di EOC. Ci evita di dover calcolare la lunghezza del numero all'inizio, per scriverla nel campo LENGTH, ma ci impone di avere tutti i byte che compongono il numero diversi dal byte di terminazione.
Supponiamo di dover rappresentare un numero >255, che quindi richiede 2 byte. Consideriamo i primi 9 bit (primo byte + primo bit del secondo): non posso averli tutti uguali a 0 o tutti uguali a 1.

9.1.2 PER (Packet Encoding Rules)

L'idea di questo formato è di andare a ridurre il numero di bit utilizzati al minimo possibile. La compressione delle informazioni è fatta in due modi, e uno è eliminare il vincolo di organizzazione in ottetti per ottimizzare l'uso dello spazio: BER, CER e DER hanno 8 bit come unità minima allocabile, anche se i dati che voglio rappresentare potrebbero occuparne meno (abbiamo visto prima l'esempio dei booleani, con PER possiamo rappresentarli in un bit). L'eliminazione della

suddivisione in byte porta alcune complicazioni: supponiamo di voler far stare all'interno di un byte più di un valore, come ad esempio un booleano da 1 bit e un esadecimale da 4 bit: ci rimarrebbero ancora 3 bit in cui potremmo mettere qualcos'altro. Però sorgono dei problemi di allineamento, dovuti al modo con cui sono implementati i processori: se considero ad esempio gruppi da 4 bit potrei pensare di indirizzare col valore 0 i 4 bit più a sinistra di un byte e col valore 1 i 4 bit più a destra, per scegliere quale gruppo da 4 bit devo considerare all'interno di un byte. Se metto il booleano nel primo bit, l'esadecimale va messo negli ultimi 4 e perderò 3 bit.
Il formato PER ammette sia la versione allineata che non allineata dei dati, che però richiede programmi più complessi per accedere ai dati visto che non potrò accedervi con le istruzioni predefinite del processore (che richiedono allineamento). Quindi la scelta tra formato allineato e non allineato è una scelta tra spazio ed efficienza dei programmi che accedono ai dati.
La seconda tecnica di compressione consiste nell'inserire solo le informazioni strettamente indispensabili per rappresentare i dati. Nella rappresentazione BER abbiamo visto che per rappresentare un valore ci sono molte codifiche aggiuntive oltre al valore vero e proprio: campo TAG, lunghezza ecc, informazioni che vengono ripetute in ogni dato di quel tipo. Si può pensare di mettere da una parte le informazioni che ci dicono come il dato è strutturato, e dall'altra i dati veri e propri: in questo avremo le informazioni sulla struttura del tipo di dato memorizzate una volta per tutte e non ci sarà bisogno di ripeterle in ogni istanza. Questa soluzione impone alcune limitazioni: dato che nei singoli dati non ci sono informazioni sulla loro struttura, abbiamo bisogno che queste informazioni ci vengano date in qualche modo per poter decodificare un dato. La codifica PER è definita nello standard ITU X.691.

Esiste anche una codifica basata su XML, XER (XML Encoding Rules).

10. Thread

I thread sono flussi di esecuzione differenti eseguiti all'interno di uno stesso processo. Il concetto di processo è pensato per avere programmi in esecuzione indipendenti uno dall'altro: ogni processo ha la sua porzione di risorse virtualizzate che usa indipendentemente da quello che fanno gli altri processi in esecuzione. Per i thread l'idea è esattamente opposta: vogliamo avere più programmi in esecuzione che possano vedersi e condividere il più possibile le risorse. Se i processi sono in competizione tra loro per ottenere le risorse, con il sistema operativo a fare da arbitro per fare in modo che tutti possano funzionare, i thread cooperano tra loro vivendo in un'ambiente comune messo a disposizione dal sistema operaitvo, ma non gestito da esso: la suddivisione delle risorse all'interno dell'ambiente è autogestita dai thread stessi. L'ambiente comune è un processo, e i file descriptor al suo interno sono visibili a tutti i thread. C'è il rischio di conflitti nell'utilizzo delle risorse, ma la loro gestione e prevenzione è affidata al programmatore e non al sistema operativo.
In un sistema POSIX quando creo un processo al suo interno viene creato un solo thread, eseguito sequenzialmente.
L'utilizzo di thread fa risparmiare nella gestione dei processi, poiché molte strutture dati non verranno replicate. Si risparmia inoltre nel costo della comunicazione tra processi: questa infatti richiede l'intervento del sistema operativo, mentre la comunicazione tra thread avviene attraverso le aree di memoria comuni.
Dal punto di vista dell'allocazione della memoria i thread devono risiedere in uno stesso segmento. Come abbiamo già visto, la parte utente di memoria di un processo è suddivisa in codice, heap, dati statici e stack. Nell'utilizzo dei thread il codice, lo heap e i dati statici sono

condivisi, mentre lo stack no: questo perché voglio che i thread vedano gli stessi dati e possano chiamare le stesse funzioni, mentre lo stack dipende dalle funzioni chiamate dai singoli thread e quindi non può essere condiviso, ne viene creato uno per ogni thread. Per comunicare informazioni contenute sullo stack con altri thread, andranno copiate in una delle aree di memoria condivisa, ad esempio tramite variabili globali.
I thread, come i processi, vengono eseguiti singolarmente, e il loro ordine di esecuzione è deciso da uno scheduler del sistema operativo, a meno che il processore non supporti l'esecuzione di più thread contemporaneamente. In questo caso l'utilizzo di thread incrementerà la velocità del sistema.
Ma se l'ordine di esecuzione non è predicibile, come fanno i thread a cooperare tra loro? Esistono delle primitive di sincronizzazione come pthread_join, che blocca il thread che la chiama finché l'esecuzione di un altro thread (specificato nei parametri) non termina. Senza questa funzione non potremmo sapere quando il risultato dell'esecuzione di un thread, generalmente memorizzato in una variabile globale, può essere letto. Un thread può essere joinable o non joinable: di default tutti i thread sono joinable, se voglio crearne uno non joinable devo specificarlo.
Non è l'unico tipo di sincronizzazione di cui ho bisogno: devo anche sincronizzare l'accesso alle variabili comuni, in modo che vengano modificate nella giusta sequenza per produrre risultati corretti. Per fare questo esistono delle primitive chiamate **mutex** (mutual exclusion), che fanno in modo che due thread non possano essere eseguiti contemporameamente.
Quando creo un thread, la funzione pthread_create ha anche il compito di passare alla funzione chiamata i parametri necessari. E dovranno naturalmente essere visibili al nuovo thread: non potranno quindi essere memorizzati nello stack del chiamante, solitamente si utilizza un array memorizzato tra i dati statici. Un array per consentire che a diverse istanze della stessa funzione possano essere passati parametri diversi.
La programmazione con i thread è particolarmente complicata, soprattutto in fase di debugging, perché non avrò sempre la stessa sequenza di esecuzione dei thread.

12.1 Primitive per l'uso dei thread

- **`pthread_create(....,funzione,...):`** crea un nuovo thread che esegue la funzione specificata. La funzione deve avere un singolo parametro di tipo **`void*`** e tipo di ritorno **`void*`**. Quando creiamo un nuovo processo, di default viene creato un solo thread che chiama la funzione main.
- **`pthread join(thread_id,):`** blocca il thread che l'ha chiamata in attesa che termini il thread specificato da thread_id. È una system call bloccante.
- **`pthread_mutex_lock/unlock`:** la lock, se chiamata da tutti i thread, consente solo ad uno l'esecuzione e blocca tutti gli altri (system call bloccante), in modo che le variabili possano essere modificate dal thread in esecuzione con la certezza che gli altri thread non gliele stanno cambiando a loro volta. Una volta finito, il thread chiama la unlock, e un altro dei processi che aveva chiamato la lock si sblocca e procede (da solo) nell'esecuzione.

12.2 Thread-safety: prevenzione delle race conditions

Analizziamo il risultato dell'esecuzione di questo codice C:

```
static int i=1;
void* t1 (void* p)  {
        int v=*(int*)p;
        v++;
        *(int*)p=v;
}

void* t2 (void* p)  {
       int v=*(*int)p;
       v--;
       *(int*)p=v;
}
```

da qualche altra parte avverranno le chiamate

```
pthread_create(myt, NULL, t1, (void*)&i);
pthread_create(myt+1, NULL, t2, (void*)&i);
for(j=0; j<2; j++)
        pthread_join(myt[j], NULL);
printf("i=%d\n", i);
```

L'esecuzione di questo codice potrebbe dare in output 0, 1 o 2 a seconda dell'ordine in cui i thread vengono eseguiti. L'ordine di esecuzione infatti dipende dallo scheduler del sistema operativo, che potrebbe anche interrompere l'esecuzione di uno per mandare in esecuzione l'altro. L'ordine in cui eseguo le pthread_create non significa nulla: non è garantito che l'esecuzione di un thread parta quando lo creo, sarà lo scheduler a decidere anche quello e potrebbe scegliere di fare continuare il thread principale invece di eseguire quello nuovo.
Le situazioni in cui il risultato dell'esecuzione di un programma dipende dall'ordine in cui parti concorrenti di codice vengono eseguite sono chiamate **race conditions** (o **corse critiche**). Sono errori di programmazione causati principalmente dal pensare come fosse sequenziale l'esecuzione di codice non sequenziale come quello di un programma che fa uso di thread. Il problema è che a causare race conditions potrebbe essere non solo il nostro codice, ma anche librerie di sistema inadatte all'utilizzo con i thread. Quando programmiamo per thread, dobbiamo assicurarci che tutte le funzioni di libreria usate nel programma siano **thread-safe** (o **reentrant**).
Un esempio di funzione thread-unsafe è la **`strtok`**. Crea problemi perché, se invocata con primo parametro NULL su una stringa che aveva già analizzato, si ricorda il punto a cui era arrivata l'ultima volta e continua la tokenizzazione da lì. Per fare questo memorizza un puntatore all'ultimo token analizzato. Ma se io la chiamassi in un altro thread su un'altra stringa questo puntatore non avrebbe più il valore atteso, e la chiamata fallirebbe (o, peggio, produrrebbe risultati errati). In generale, una funzione che mantiene memoria delle chiamate precedenti, e questa memoria ha effetto sul suo funzionamento, sarà sicuramente thread-unsafe.
Esiste una versione rientrante di strotk, la **`strtok_r`**, che prende un parametro in più: **`char** saveptr`**, che svolge la funzione di quella che nella versione non rientrante era la variabile interna statica saveptr.
Molte funzioni POSIX hanno una versione rientrante e una non rientrante: questo perché quando è stato introdotto lo standard POSIX la programmazione per thread non esisteva, e quando venne inventata è stata scritta una versione thread-safe di tutte le funzioni che causavano race conditions, identificata dal suffisso _r.

12.3 Deadlock

Abbiamo visto alcune tecniche di sincronizzazione per la prevenzione delle race conditions, come pthread_join e mutex lock/unlock. La sincronizzazione introduce un nuovo problema, il **deadlock**: due o più thread che si attendono reciprocamente, e per questo non terminano mai a meno di interventi esterni.

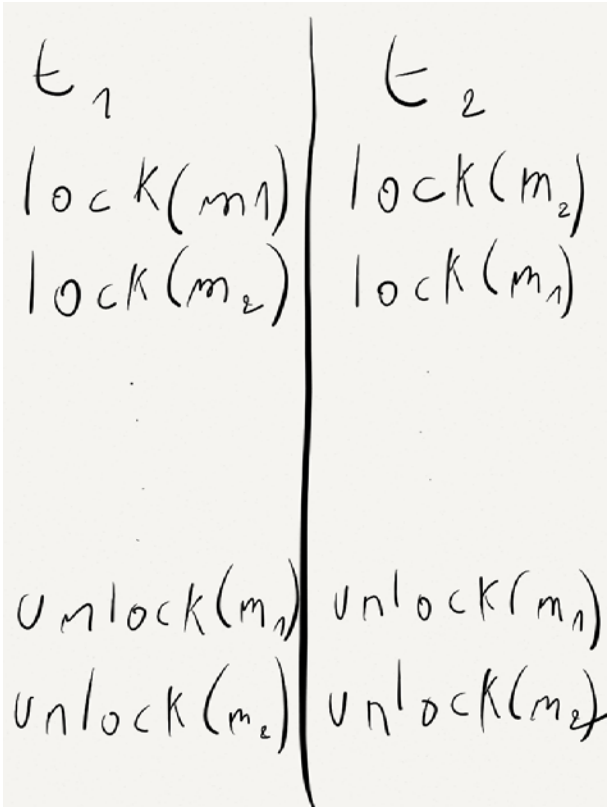
Immaginiamo di avere tre thread t1, t2 e t3. Ad un certo punto t1 fa join su t2, t2 fa join su t3 e t3 fa join su t1. Questo è un deadlock: t1 per terminare deve attendere t2, che deve attendere t3, che deve attendere t1: è un circolo vizioso che non può terminare spontaneamente.

Nel caso in cui i thread fossero stati solo due, con t1 che joina t2 e t2 che joina t1, a runtime il kernel avrebbe riconosciuto la situazione di deadlock, facendo ritornare la join con una condizione di errore EDEADLK.

Rappresentando i thread come nodi di un grafo e le join come archi, ogni volta che troviamo un ciclo abbiamo un deadlock. Questo rende verificabile la presenza di deadlock anche in caso di più di due thread, basterebbe cercare i cicli nel grafo, ma in caso di molti thread la cosa sarebbe inefficiente e quindi non è implementata: il kernel riconosce ed evita automaticamente il deadlock al massimo per due thread che si joinano a vicenda. Questo è applicabile non solo ai thread ma a qualunque sistema distribuito in cui le parti possono aspettarsi a vicenda: possono verificarsi deadlock anche tra processi, con fork e waitpid.

Il watchdog è un meccanismo di timeout per evitare i deadlock. Se ho un'applicazione a rischio di deadlock, e conosco il tuo tempo normale di esecuzione, posso impostare un timeout oltre al quale saprò che l'applicazione è andata in deadlock e potrò quindi terminarla.

Anche le primitive di mutex sono soggette a deadlock, dato che sono bloccanti.



In questo caso si verifica un deadlock perché t1 deve aspettare t2 per eseguire il lock su m2, visto che t2 lo ha fatto prima, e per lo stesso motivo t2 deve aspettare t1 per eseguire il lock su m1. Rimangono entrambi bloccati in attesa dell'altro, senza poter arrivare ad eseguire una unlock.

È una situazione più complessa della precedente perché non è detto che avremo un deadlock, dipende da come i thread vengono schedulati: avremo un deadlock solo se t1 viene interrotto dopo la prima istruzione e viene attivato t2. È una situazione in cui il verificarsi o no di deadlock è determinato da una corsa critica.
Il modo più semplice di evitare questa situazione è di modificare leggermente il codice per fare in modo che le lock sui mutex avvengano nello stesso ordine in tutti i thread: in t2 mettiamo per prima lock(m1). Ma anche questo non ci salva completamente: se all'interno di un thread che ha eseguito una lock ci sono errori di programmazione che lo portano in un loop infinito, non rimarrà bloccato solo quel thread ma anche tutti quelli in attesa che il mutex su cui quel thread aveva fatto lock venga sbloccato.
Esempio dei cinque filosofi: ci sono cinque filosofi orientali attorno ad una tavola rotonda, ed ognuno ha davanti la sua ciotola di riso. Ci sono solo cinque bacchette però, per cui ogni filosofo ha una bacchetta sulla destra e una sulla sinistra. Se tutti i filosofi si comportano allo stesso modo, ad esempio prendendo prima la bacchetta sulla destra e poi quella sulla sinistra abbiamo un deadlock: tutti rimangono in attesa della bacchetta sinistra, che non arriverà mai perché la mia bacchetta sinistra è la bacchetta destra dell'altro. Lo stallo deriva dal fatto che i filosofi prendono prima una bacchetta e poi l'altra, e lo fanno tutti nello stesso ordine. Ci sono due soluzioni: prendere contemporaneamente le due bacchette solo se ci sono tutte e due, ma potrebbe essere molto complicato implementare un sistema in cui dobbiamo eseguire due azioni contemporaneamente. Una soluzione più semplice da implementare è diversificare il comportamento delle varie parti: se almeno un filosofo è mancino riuscirà a prendere tutte e due le bacchette.

11. Scheduling

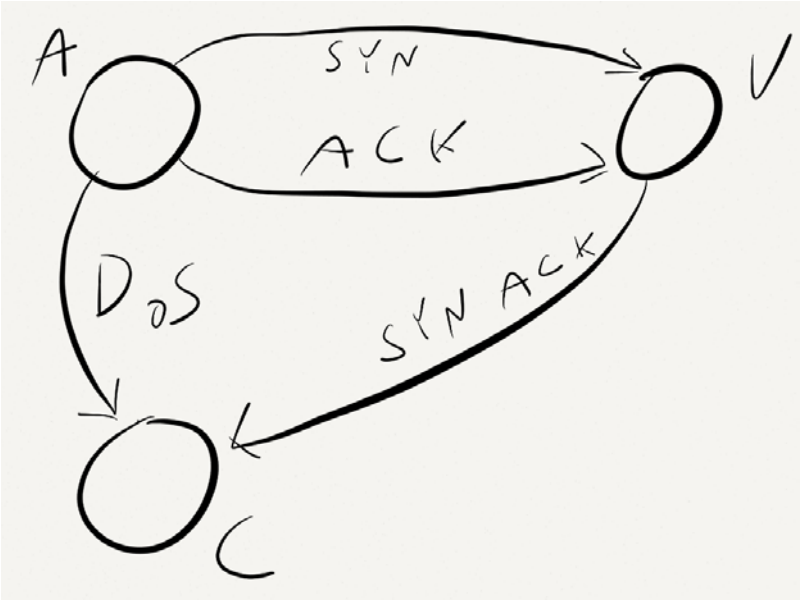
Ci sono diverse politiche con cui si può gesitre lo scheduling. In un sistema dotato di interfaccia grafica ad esempio, potrò dare la priorità all'interfaccia per fare in modo che l'interazione con l'utente non sia mai compromessa, e possa quindi essere l'utente a decidere quali applicazioni terminare, eseguire o mettere in attesa. Se ad esempio ho un'applicazione bloccata, ma l'interfaccia utente è sempre attiva perché lo scheduler gli da priorità maggiore, l'utente sarà in grado di terminare l'applicazione bloccata, ma se lo scheduler ha dato priorità all'applicazione bloccata in un loop e messo l'interfaccia utente in attesa, all'utente non resta che staccare la spina.
I sistemi operativi moderni, basati sul concetto di time sharing, danno la priorità ai processi che hanno maggiore interazione con l'utente, penalizzando quelli di solo calcolo. Un processo che ha interazioni con l'utente si riconoscere dai file descriptor che ha aperto: se ha aperto un terminale vuol dire che può interagire con l'utente. Si favoriscono quindi i processi che usano più tempo in attesa di input/output piuttosto che quelli che fanno un uso intenso del processore.
Uno dei principi di time sharing è quindi quello di favorire i processi che interagiscono con l'utente. Un altro è quello di favorire i processi più brevi. Se ho da eseguire tanti processi brevi ed uno lungo, se eseguo prima quello lungo e poi quelli brevi avrò il massimo tempo medio di esecuzione possibile, mentre se eseguo prima quelli brevi avrò il minimo.
Il problema è che non possiamo sapere il tempo di esecuzione di un thread prima di eseguirlo, lo saprò solo quando il thread termina. La soluzione prevede la combinazione di due tecniche diverse: **round-robin** e **priorità dinamica**.
Il round-robin prevede di prendere un'unità molto piccola di tempo, chiamata **time slice**, passata la quale scatta un timeout e viene mandato in esecuzione un altro processo, sempre per la stessa

time slice. Utilizzando solo il round-robin non si avrebbe una grande ottimizzazione, perché allunga il tempo medio di esecuzione di tutti i processi. Per questo viene abbinato all'uso di priorità dinamica: un valore che può essere assegnato dal sistema per default o direttamente dall'utente. Ogni volta che un processo va in esecuzione, la sua priorità viene diminuita di 1 per fare in modo che al giro successivo venga eseguito qualcun altro' con priorità maggiore. In UNIX il sistema di priorità si chiama NICE. Questa sistema di priorità serve inoltre a distinguere quei processi che si interrompono prima della fine della loro time slice per restare in attesa di I/O: quando un processo si interrompe prima di aver esaurito la time slice, il suo livello di priorità non viene dimunuito. Un processo in attesa non pesa sugli altri perché non è schedulabile: non corre quindi il rischio di essere mandato in esecuzione inutilmente, per scoprire che è ancora in attesa. Con questo sistema quindi, i processi che si interrompono più frequentemente per fare I/O (quindi, per interagire con l'utente) manterranno una priorità più alta mentre quelli che svolgono più calcoli ed esauriscono sempre la time slice vedranno la loro priorità ridursi.

12. Sandboxing e sicurezza

Tramite la virtualizzazione si può aumentare il livello di sicurezza offerto dal sistema operativo. Un sistema operativo moderno è infatti un software estremamente complesso, e quindi conterrà sicuramente dei bug che possono essere sfruttati da un attaccante. Un hypervisor posto al di sotto del sistema operativo aumenta la sicurezza del sistema, perché assume il controllo delle risorse e permette di controllare l'integrità del sistema.
Uno degli utilizzi principali dei malware è quello di creare una botnet: una rete di computer infetti che può essere utilizzata per vari scopi dall'attaccante, tipicamente per compiere attacchi DoS verso altri bersagli. Una condizione necessaria perché il computer rimanga infetto sarà nascondere per bene il software malevolo al bersaglio, e ciò può essere ottenuto modificando le applicazioni che permetterebbero di verificare la presenza del malware. In un sistema UNIX ad esempio converrà modificare il comando ls perché non visualizzi i file del malware.
La sicurezza non può essere creata dal nulla, ma si basa sulla sicurezza delle parti che uso per comporre il mio sistema: se voglio assemblare un sistema sicuro, bisogna preoccuparsi dell'integrità di ogni programma, non solo quelli installati da me ma anche di quelli già presenti nell'hardware della macchina: nessuno mi garantisce che il microcodice contenuto nei chip non contenga parti malevole (alcuni anni fa il governo americano si preoccupò di mantenere negli USA la produzione dei microprocessori). Ragionando in questo modo, raggiungere la sicurezza è teoricamente impossibile, perché anche gli strumenti che uso per verificare l'integrità potrebbero essere modificati per non rilevare alcune infezioni. Di base, la sicurezza dipende dalla fiducia che ripongo nelle componenti base del mio sistema.
Un possibile modo di verificare la presenza di rootkit è di eseguire il sistema in una macchina virtuale, che ne osserva il funzionamento e ne rileva le anomalie. Questo permette di accorgersi di un'infezione prima che produca danni, perché i danni prodotti sono confinati nella macchina virtuale. Il sandboxing consente di mettere a disposizione del software ritenuto non affidabile non il sistema vero, ma un sistema virtualizzato (sandbox, o honey pot) isolato dall'esterno. In questo modo l'hypervisor può osservare il funzionamento del sistema e rilevare l'infezione appena si manifesta. Se invece, trascorso un certo tempo, non vengono rilevate infezioni, posso classificare l'applicazione come sicura ed installarla nel sistema reale.

Sandbox e honey pot non sono completamente sinonimi: l'idea di sandbox è quella di confinare l'applicazione in un ambiente virtuale, l'idea di honey pot è di far credere all'applicazione di essere eseguita nella macchina reale nonostante sia confinata in un'ambiente virtuale, ed è utile ad esempio per verificare se l'applicazione in test è un malware ad orologeria, perché all'interno dell'honey pot posso mostrargli una data diversa. L'honey pot quindi è un modo per spingere i malware a manifestarsi, e ciò può essere fatto ad esempio mostrando vulnerabilità che in realtà non ci sono, e che il malware tenterà di sfruttare.
Tipicamente una sandbox è usata per controllare e limitare le applicazioni su una macchina, l'honey pot per testare servizi di rete a rischio: in modo simile ad un parafulmine, può simulare un servizio molto vulnerabile per fare in modo che gli attaccanti si concentrino su quello. Non mi permette però di identificare con certezza l'attaccante, perché potrebbe aver fatto IP spoofing: sul protocollo UDP è possibile infatti mettere negli header un indirizzo sorgente diverso da quello reale, il pacchetto arriva lo stesso. Esistono tecniche per evitare ciò, ma non sono implementate su tutti i nodi della rete. Anche le tecniche di NAT consentono di mascherare l'indirizzo IP del mittente reale. Lo spoofing su servizi TCP è più complicato a causa del three-way-handshake. Ecco un possibile attacco di IP spoofing su TCP, ideato da Kevin Mitnick e non più utilizzabile:
Un attaccante vuole aprire una connessione TCP usando un indirizzo IP diverso dal suo: invia il SYN con l'indirizzo falso, la vittima lo riceve e invia il SYN ACK a questo indirizzo, che non è sotto il controllo dell'attaccante, e quindi ricevendo un SYN ACK inaspettato risponderà RST per terminare la connessione. Inoltre l'attaccante per inviare l'ACK dovrebbe conoscere l'acknowledgement number inviato dalla vittima alla terza macchina. Sembra un attacco impossibile.
Ma se prima l'attaccante ha effettuato un DoS sulla terza macchina, questa non sarà riuscita a inviare il RST in risposta al SYN ACK inaspettato. A questo punto per completare l'handshake l'attaccante deve indovinare l'acknowledgement number corretto: ciò era possibile sfruttando una vecchia vulnerabilità del protocollo TCP, per cui gli ack# non erano perfettamente casuali ma risultavano predicibili.
Sono necessarie due diverse vulnerabilità: quella della terza macchina che cade all'attacco DoS e quella della generazione degli ack#. C'è un difetto in questo attacco: per indovinare l'ack# è necessario contattare prima il bersaglio con il proprio IP, e infatti è stato proprio questo a far arrestare Mitnick.



Schema dell'IP spoofing di Mitnick: manca la parte iniziale con IP in chiaro per poter predire l'acknowledgement number

Buffer overflow: è uno dei principali meccanismi di diffusione dei worm che sfrutta un errore di programmazione abbastanza comune: non fare il controllo della memoria allocata prima di utilizzarla. Supponiamo di voler implementare una funzione che legge l'input da tastiera a livello basso, non con scanf, getline e simili che sono immuni a questo bug. avrò un buffer di una certa dimensione, in cui andrò a mettere i caratteri letti. Se faccio le cose male e non controllo che non arrivino più dati della dimensione del buffer, potrebbe venirmi dato un input più grande del buffer di lettura, e quindi l'input verrebbe scritto in zone di stack esterne al buffer. Se ho la fortuna di uscire dallo stack avrò un segmentation fault, altrimenti l'input mi sovrascriverà parte dello stack. All'interno dello stack sono salvati anche i registri del processore, come l'indirizzo di ritorno della funzione. Se viene sovrascritto con un indirizzo esterno al segmento codice avrò un segmentation fault, altrimenti mi troverò ad eseguire una parte del segmento codice diversa da quella prevista dal programmatore, e una scelta furba per un attaccante sarà quella di far eseguire una shell. È un modo di modificare a runtime il codice di un processo. Può essere sfruttato a fin di bene per distribuire rapidamente, in una rete locale, la patch per correggerlo: si diffonde un worm all'interno della rete che installa la patch in ogni macchina in cui trova il bug.

Reti (prof.ssa Ribaudo)

1. Introduzione alle reti

Nella parte del corso sui sistemi operativi abbiamo visto come avviene la comunicazione tra due processi su macchine distinte tramite i socket, ipotizzando che tra le due macchine ci fosse una connessione diretta. Ma non è mai così: in genere la connessione tra due macchine avviene attraverso una **rete**, e la comunicazione su una rete è regolamentata dai **protocolli**.

Il concetto di rete nasce in America negli anni '60 con ARPANET, rete pensata per rimanere in piedi anche in caso di attacchi nucleari. Doveva essere quindi decentralizzata: la distruzione di un nodo non ne doveva compromettere il funzionamento. Dagli anni 60 agli anni 80 si svolse ricerca per sviluppare i protocolli di rete.

Il Web nasce all'inizio degli anni 90, e cambia radicalmente il modo di utilizzare la rete: prima del web, infatti, per utilizzare la rete bisognava conoscere dettagliatamente il funzionamento dei protocolli e i comandi per utilizzarli, si faceva tutto da shell.

Il web nasce come unione dei concetti di ipertesto, internet e database.

La rete Internet è un'insieme di sottoreti connesse tra loro, regolate da standard pubblici. I protocolli di rete vengono proposti, discussi per lungo tempo e, se vengono approvati, ne viene spiegato il funzionamento in un documento RFC (Request For Comments), pubblicato dalla IETF su ietf.org. (Internet Engineering Task Force)

Network edge: i confini della rete, le singole macchine che vi si connettono. Devono passare attraverso un **access network** fino ad arrivare al **network core**, router interconnessi che instradano le informazioni sulla rete.

Internet è una struttura gerarchica formata da Autonomous Systems. Quelli di livello più alto sono gli **Autonomous Systems Tier-1 ISP**, di copertura nazionale e internazionale. Sono i provider di livello più alto, chiamati anche **internet backbone**, e sono quelli che garantiscono i collegamenti internazionali. Un esempio di Tier-1 a livello europeo è Geant, rete che mette in comunicazione tutti i network universitari europei.

I **Tier-2 ISP** sono più piccoli dei Tier-1, di dimensione regionale o nazionale, e possono connettersi ad altri Tier-2 o ad un Tier-1. Un esempio di Tier-2 è la rete GARR, che connette tutte le università italiane. I punti di interscambio tra reti sono detti Internet exchange, quello italiano più importante è il MIX (Milan Internet eXchange).

I **Tier-3 ISP** sono l'ultimo livello, il più vicino agli end system.

Le diverse componenti di Internet, come:

- host
- router
- canali di comunicazione
- applicazioni
- protocolli
- hardware e software

devono essere messe in grado di comunicare tra loro, nonostante le differenze. Perché tutto funzioni è necessaria un'**organizzazione modulare** della rete: non esiste un unico algoritmo che gestisce tutto il traffico della rete. È una struttura a livelli in cui la comunicazione avviene solo tra pari livello (peer). Esiste uno standard che definisce i livelli di comunicazione, lo **stack di Internet**, suddiviso in **application, transport, network, link** e **physical** (dal più alto al più basso livello di atrazione). Del livello fisico non vedremo niente, il livello link definisce come viaggiano i dati a livelli dei singoli collegamenti. Il livello network definisce come i dati viaggiano sul network core (rete dei router). A livello di trasporto avviene il multiplexing e demultiplexing dei dati, mentre il livello applicazione è quello più vicino all'utente, ma anche al programmatore di applicazioni utente, che non dovrà preoccuparsi dei livelli transport e network poiché generalmente implementati dal sistema operativo.
A livello applicazione viene generato un messaggio da inviare, a cui a livello transport saranno aggiunte alcune informazioni di indirizzamento (ad esempio le porte). A livello network e signal vengono aggiunte ulteriori intestazioni (a livello network ad esempio va aggiunto l'indirizzo IP del destinatario), fino ad arrivare al livello fisico in cui abbiamo solo dei segnali trasmessi sui mezzi di interconnessione (cavi, etere).

Nelle slide c'è una rappresentazione dello stack di Internet con i vari protocolli utilizzati ad ogni livello. Vediamo che a livello network l'unico protocollo utilizzato è l'IP. Ogni protocollo si appoggia su quelli di livello più basso nello stack: ad un browser che deve comunicare sul protocollo HTTP non interessa come i dati viaggeranno sui router o sui cavi, o di come verrano multiplexati: se ne occuperanno i protocolli di livello inferiore.

2. Livello di trasporto

Ogni livello dello stack dei protocolli di rete è stato pehato per offrire dei servizi. I servizi offerti dai protocolli a livello di trasporto sono:

- **multiplexing/demultiplexing:** indispensabile per il corretto smistamento dei dati alle applicazioni, è offerto da tutti i protocolli di trasporto
- **trasferimento dati affidabile:** voglio che i pacchetti che compongono il mio messaggio arrivino tutti, inalterati e nel giusto ordine.
- **controllo di flusso:** serve per permettere di moderare la velocità di chi invia in modo che chi riceve abbia il tempo di processare correttamente i dati (implementato da TCP ma non da UDP)
- **controllo di congestione**: simile al controllo di flusso, ma sulla rete. Gestisce il traffico di dati in modo che non ci siano ingorghi (implementato da TCP ma non da UDP)
- **controllo degli errori** (checksum)

A occhio sembrerebbe che TCP sia un protocollo molto migliore di UDP, ma in realtà tutto dipende dalle necessità della specifica applicazione: infatti TCP ha più funzionalità, ma è anche più complesso e lento: se la mia applicazione non ha bisogno dei controlli supplementari di TCP o se ha bisogno di un'ottima velocità (ad esempio, applicazioni di VoIP o streaming) allora mi converrà utilizzare UDP, molto più veloce poiché in pratica si limita al multiplexing/demultiplexing dei dati.
I protocolli di trasporto forniscono la comunicazione logica tra processi applicativi di host differenti. In fase di invio i messaggi del livello applicativo e li suddivide in **segmenti,** incapsulati con un header (che servirà al ricevente per il demultiplexing), da passare al livello di rete. In fase di ricezione prende i segmenti e li riassembla per poi passarli al livello applicativo.

Porte
Secondo gli standard definiti da IANA, le porte dalla 0 alla 1023 sono dette **well-known** e in un sistema UNIX sono necessari privilegi di root per fare binding su una di queste porte (quindi, un qualunque server che usi una porta well-known va eseguito come root). Le porte dalla 1024 alla 49151 (**registered**) sono assegnate da IANA su richiesta per servizi specifici, e generalmente possono essere utilizzate con privilegi standard. Le porte dalla 49152 alla 65535 (**dynamic** o **private**) non possono essere assegnate, vengono utilizzate in modo temporaneo dalle applicazioni.

2.1 Protocollo UDP (trasporto senza connessione, RFC 768 del 28 agosto 1980)

UDP (acronimo di User Datagram Protocol) è uno dei protocolli più minimali in assoluto: fornisce quasi l'accesso diretto al protocollo IP. Il suo servizio di consegna best effort è paragonabile al servizio postale: i segmenti possono andare persi o arrivare fuori sequenza. È un **protocollo senza connessione**: non c'è nessun canale tra mittente e destinatario e ogni segmento UDP è gestito indipendentemente dagli altri. Permette una comunicazione di tipo **broadcast**: uno che invia e molti che ricevono.

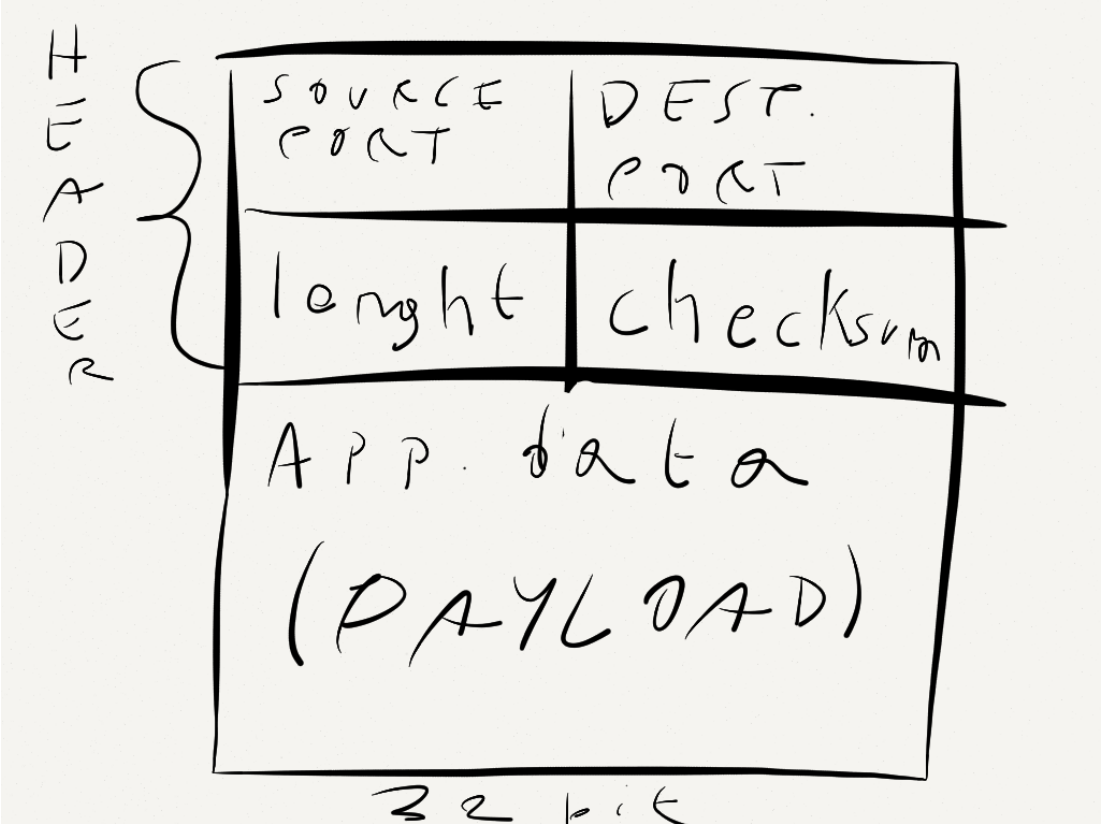
Pregi di UDP:

- **velocità:** non viene creata nessuna connessione
- **semplicità**: non si mantengono informazioni sullo stato della connessione
- **brevità**: header di segmento molto corti (intorno agli 8 byte) perchè non contengono informazioni di controllo
- possibilità di "sparare dati a raffica" vista l'assenza di controlla di flusso e congestione

Per le sue caratteristiche, UDP è spesso utilizzato per applicazioni di streaming audio/video, poiché tollera piccole perdite di dati e permette di spedire a qualsiasi frequenza. Viene impiegato anche dai protocolli DNS, SNMP, DHCP e per comunicazioni di tipo broadcast[1] o multicast[2].
Se necessitò di maggiore affidabilità di quella offerta dal protocollo UDP, dovrò implementarla a livello applicativo (ma non è conveniente).

2.1.1 Segmento UDP

Un segmento UDP è composto dall'header e dal payload (dati effettivi da trasmettere, senza header).Un header UDP cotiene le seguenti informazioni: porta di origine e di destinazione, lungezza e checksum. La dimensione massima di un segmento è di 65536 byte, di cui 65507 utilizzabili dal payload: questo perché ai 65536 iniziali vanno tolti 8 byte di header UDP e 20 byte di header IP. Il checksum (controllo di integrità dei dati) è ripetuto ad ogni livello dello stack ad esclusione di quello applicativo.



Struttura di un segmento UDP.

[1] **broadcast:** un mittente trasmette a tutti i nodi di una rete

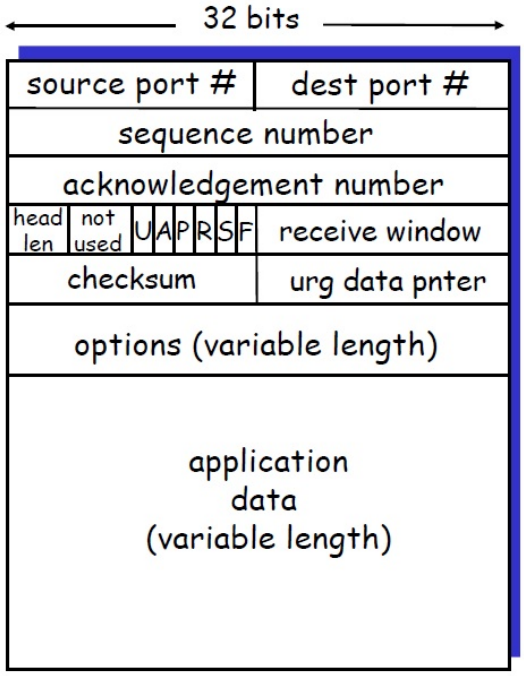
[2] **multicast:** un mittente trasmette ad un sottoinsieme di nodi di una rete.

Il checksum UDP è un controllo eseguito per verificare se ci sono stati errori di trasmissione. Il mittente tratta il contenuto del segmento come una sequenza di parole a 16 bit, ne somma i contenuti e ne fa il complemento a 1: il risultato viene posto nel campi checksum dell'header. Il ricevente somma tutte le parole a 16 bit del payload insieme al checksum: se il risultato è composto da soli 1 non c'è nessun errore, altrimenti si e il segmento verrà ignorato dal ricevente. Il sistema funziona perché sommando un numero al suo complemento a 1 dovrò ottenere solo 1: se non è così vuol dire che o il checksum o il payload sono stati alterati dalla trasmissione.
Questo checksum viene applicato anche su alcuni campi dell'header IP, in parziale violazione del principio per cui ogni livello dello stack deve essere indipendente.
Quando invio una richiesta a un server su una porta su cui non ci sono processi in ascolto, il server mi risponderà con un messaggio ICMP di porta non raggiungibile. Questo permette un attacco DoS chiamato UDP flood attack in cui l'attaccante invia un gran numero di pacchetti UDP su porte casuali, e l'attaccante risponderà con una serie di messaggi ICMP di port unreachable

2.2 Protocollo TCP (RFC 793 del settembre 1981)

TCP (Transmission Control Protocol) è un protocollo punto-punto: un solo mittente e un solo destinatario. Vede i dati non come singoli segmenti ma come un flusso continuo di messaggi, con affidabilità garantita da un meccanismo di **acknowledgment**: ogni volta che invio un messaggio mi aspetto che il ricevente mi dia conferma di averlo ricevuto. Può essere visto come il rapporto di consegna delle e-mail, applicato però ad ogni singolo segmento. L'acknowledgment è integrato da un meccanismo di **timeout**: se, passato un certo tempi dalla trasmissione, non ho ancora ricevuto l'acknowledgment del ricevente vuol dire che il messaggio non è arrivato. È una comunicazione di tipo **full duplex**: flusso di dati bidirezionale.

Segmento TCP



La struttura di un segmento TCP è molto più complessa di un segmento UDP. Ha alcuni campi in comune (source e destination port, checksum), integrati da un **sequence number** e un **acknowledgment number**, un campo **receive window** contenente il numero di byte che il destinatario desidera accettare, alcuni flag per impostare o chiudere la connessione (RST, SYN, FIN), il flag che indica se il segmento è un messaggio di acknowledgment (ACK), più i flag URG e PSH e un campo per dati urgenti (raramente utilizzati).

2.2.1 Connessione TCP

Prima di poter scambiare informazioni, è necessario aprire una connessione tra client e server. Perché questo avvenga bisogna che si accordino su alcuni parametri TCP: numero di sequenza, dimensione dei buffer di ricezione e invio, informazioni per il controllo di flusso. Questa fase è detta **three ways handshake**.
Il primo passo è comunicare al server il numero di sequenza iniziale: invio un messaggio con campo SYN impistato a 1 e seq=x. Se il server è disponibile alla connessione mi risponde con un messaggio SYN ACK: mi da il suo numero di sequenza iniziale y (indipendente da quello del client) e ack=x+1 (dove x è il numero di sequenza iniziale del client). Il terzo messaggio, inviato dal client, è un ACK con seq=x+1 e ack=y+1 . In questo ultimo messaggio di controllo volendo si potrebbero già inviare dati. *//Tutta questa parte si capisce meglio nelle slides*.
Se un client cerca di aprire una connessione TCP su una porta chiusa l'host restituisce un segmento TCP con flag RST impostato a 1.
Anche sul TCP è possibile un attacco DoS simile all'UDP flood: il **SYN flood.** Poiché il server appena riceve un segmento SYN alloca e inizializza buffer e variabili di connessione, io posso inviare un gran numero di pacchetti SYN senza mai completare la terza fase dell'handshake: in questo modo costringerò al server ad allocare un gran numero di risorse inutili, che saranno in seguito deallocate per timeout con conseguente spreco di tempo di CPU, e quindi far bloccare il server. La soluzione per proteggersi da questo attacco è di non allocare le risorse fino al terzo passo dell'handshake:appena ricevuto il SYN rispondo con un SYN ACK con un numero di sequenza iniziale ISN costruito ad hoc, ma non alloco le risorse. Questo numero ISN è generato in modo da codificare tutte le informazioni sulla connessione che si cerca di instaurare. Per instaurare una nuova connessione TCP si usa la primitiva POSIX **`int connect(int socketfd....)`** che inizia il three ways handshake, inviando un pacchetto SYN.
Dai SYN flood ci si può difendere con i SYN cookies: il server non alloca le variabili alla ricezione della richiesta SYN, ma nel SYN ACK di risposta include un ISN (numero di sequenza iniziale) opportunamente calcolato. Se l'ISN incluso nell'ACK del client è coerente, autorizza la connessione.
TCP usa una chiusura simmetrica della connessione: chi vuole chiudere invia un segmento FIN (flag F attivo) al partner, che invierà un ACK seguito da un FIN. L'altro risponde con un ACK, e se entro l'intervallo di timeout non arrivano altri segmenti la connessione è chiusa.
TCP vede i dati come un flusso ordinato di byte: il numero ISN incluso nell'header dei segmenti indica il numero del primo byte del segmento nel flusso di dati. Il numero di acknowledge rappresenta il numero di sequenza del prossimo byte atteso dall'altro lato della comunicazione.
La specifica del TCP non dice come gestire segmenti che arrivano fuori sequenza: possono quindi essere ignorati oppure salvati in un buffer in attesa dei segmenti mancanti.

API per l'utilizzo dei socket:
`socket()` crea un socket
`bind()` associa il processo ad una porta
`send()` invia messaggi (prende come parametro un puntatore al buffer di trasmissione)

`recv()` riceve messaggi (prende come parametro un puntatore al buffer di ricezione)
`send_to(), recv_from()` send e receive per la modalità datagram

Il protocollo TCP mi garantisce il recapito dei messaggi (tramite connessione, acknowledgment, timeout), il protocollo UDP no.

2.2.2 Timeout

Il meccanismo di timeout serve a capire che il messaggio non è stato ricevuto: in caso di ricezione il mittente dovrebbe ricevere un'interruzione, e se dopo un intervallo predefinito di tempo non la riceve saprà che il messaggio è stato perso. In TCP è implementato direttamente nel protocollo, in UDP no ma volendo può essere implementato a livello applicativo. Stabilire il corretto intervallo di timeout è importante, perché con un intervallo troppo breve rischio di considerare persi pacchetti che sono solo arrivati in ritardo, con un intervallo troppo lungo spreco tempo utile. Ma come si decide l'intervallo di timeout? Un possibile metodo può essere quello di calcolare una media dei tempi di trasferimento, e usare quella come riferimento per impostare il timeout. Ma se ad essere perso fosse uno dei primi pacchetti trasmessi, quando ancora non ho dati sufficienti per avere una media, non saprei come fare.

Misura del tempo
I computer misurano il tempo basandosi sul clock. L'architettura x86 prevede un contatore che viene incrementato ad ogni ciclo, e posso usarlo per misurare il tempo. Sarà anche una misura molto precisa, essendo la frequenza di clock nell'ordine dei miliardi di cicli al secondo. Per accedere a questo registro dovrò usare delle istruzioni, che come tutte le istruzioni richiederanno cicli di clock per essere eseguite, e devo sapere esattamente quanti sono per poterli sottrarre al numero contenuto nel contatore e avere così una misura precisa.
Questa misura del tempo presenta anche altri problemi: molto spesso le macchine recenti hanno frequenza variabile, per risparmiare energia. Ma con una frequenza variabile non posso sapere quanto tempo è passato contando i cicli di clock.
Allora posso usare un clock diverso da quello della CPU, un clock che non deve sincronizzare nessun processore ma serve esclusivamente a misurare il tempo. Avrà una frequenza molto inferiore al clock della CPU (sarà nell'ordine dei microsecondi), e rimane sempre acceso, anche a macchina spenta. Posso anche implementare un timer, in cui imposterò un valore che sarà scalato di 1 ad ogni colpo di clock dell'orologio, e mi invierà un interrupt quando arriva a zero. Il timer viene usato dal sistema operativo per misurare il tempo senza dover controllare continuamente il clock: se imposto il timer a un secondo, potrò aggiornare il tempo software ogni secondo perché mi arriverà l'interrupt del timer. Avrò così una misura del tempo approssimata, ma non rallenterò la CPU per leggere continuamente il valore del clock, che dovrà essere interrogato solo la prima volta, all'avvio della macchina, per inizializzare il contatore software.
Il contatore software misura il tempo in secondi passati da Epoch, l'istante zero che tutti i sistemi UNIX usano come standard.

2.2.3 Controllo di flusso e controllo di congestione

TCP, al contrario di UDP, modifica la velocità di trasmissione sulla base di quello che percepisce delle condizioni della rete, per evitare congestioni di rete. Il controllo di congestione viene gestito dagli end system, anche se si riferisce alle condizioni della rete. Potrebbe essere più efficiente se gestito direttamente dai router.

Perché ciò avvenga è necessario avere un corretto intervallo di timeout, deciso basandosi su una stima del RTT, fatta con le formule che abbiamo visto in laboratorio. TCP garantisce un **trasferimento dati affidabile**, anche se avviene su un canale non affidabile come l'IP: questo perché a livello di rete i pacchetti possono andare persi, ma TCP se ne accorge e li fa ritrasmettere con i già visti meccanismi di timeout e acknowledgement. A livello TCP il mittente è un processo che:

- riceve i dati dal processo applicativo sovrastante e li invia
- resta in attesa di un ACK del ricevente
- controlla che non sia passato l'intervallo di timeout

e svolge queste tre operazioni in un loop infinito. Nelle slide lo pseudocodice di una versione semplificata del mittente TCP.

Controllo di flusso

Il controllo di flusso è necessario per non sovraccaricare i buffer di ricezione del ricevente. Ciò avviene grazie al campo Receive Window degli header di segmento TCP: qui verrà incluso la dimensione attuale del buffer di ricezione, così il mittente potrà regolare la velocità di trasmissione di conseguenza. Sei buffer è pieno il campo WIN sarà uguale a zero, e la trasmissione verrà bloccata. Per farla riprendere verrà inviato un ACK ripetuto, cioè con stesso ackowledgement number del precedente ma con WIN diverso.

Controllo di congestione

Il controllo di congestione viene fatto a livello locale e asincrono dal mittente TCP, in base alla sua conoscenza dello stato della rete. La congestione di rete a livello di router infatti può comportare perdita di dati. Ci si può accorgere di una congestione di rete in caso di troppe ritrasmissioni: questo può infatti voler dire che ci sono problemi sulla rete.
Sarebbe molto più efficiente se fatto a livello di rete invece che a livello di trasporto, ma sarebbe estremamente costoso perché comporterebbe la sostituzione di tutti i router.
Il sender TCP controlla la congestione di rete con una variabile Congestion Window, che viene aumentata in caso di ricezione di ACK (=trasmissione andata a buon fine) e diminuita in caso di timeout raggiunto o ACK duplicati (=trasmissione fallita). Le diverse implementazioni del TCP stabiliscono come calcolare gli aumenti e le riduzioni della cwnd. Esistono infatti diverse versioni del TCP, e ogni sistema operativo ne ha una diversa.

Implementazioni Tahoe del controllo di congestione TCP

L'implementazione Tahoe utilizza meccanismi di **slow start** e di **adaptive increase**. Lo slow start, partendo da un tasso di invio iniziale di 1 MSS per RTT, incrementa di 1 MSS per ogni ACK ricevuto. Ha crescita esponenziale (non è slow per niente). Quando perdiamo un segmento capiamo di aver raggiunto un valore di cwnd troppo grande, quindi riparte da 1 e continua fino ad aver raggiunto la metà del valore a cui è avvenuta la perdita (**adaptive increase**), trasformando la crescita di cwnd da esponenziale a lineare.

Nell'implementazione Reno viene invece usato il **fast retransmit**: dimezzamento della cwnd dopo 3 ACK duplicati, e prosegue con crescita lineare. È il **TCP sawtooth** (a sega dentata) per la forma che ha sul grafico cwnd/RTT.

3. Livello applicativo

In questo livello le cose si fanno più semplici, perché l'informazione non è vista come byte ma come testo. Se a livello di trasporto parlavamo di segmenti, a livello applicativo parliamo di messaggi, la cui dimensione non è definita a priori come per i segmenti di TCP e UDP (le pagine web e i file scaricati possono avere qualsiasi dimensione). Le informazioni di questo livello verranno spacchettate in segmenti di uguali dimensioni dal protocollo di trasporto. A livello applicativo stanno il Web (protocollo HTTP), varie applicazioni (Skype) e anche il protocollo DNS (Domain Name System), che essendo un protocollo orientato a messaggi di piccole dimensioni che necessitano di risposte immediate, si appoggia al protocollo UDP per il trasporto. Ma il DNS ha anche bisogno dell'affidabilità che UDP non offre, quindi la implementa a livello applicativo.

Storia dei protocolli applicativi
Il primo protocollo applicativo fu Telnet, usato per il login su macchine spremote ma insicuro poiché trasmetteva tutto in chiaro, soppiantato da SSH. A seguire ci fu il protocollo FTP, per il trasferimento di file, il protocollo per l'email e quello per i newsgroup (NNTP). All'inizio degli anni 90 arriva il Web, a fine anni 90 il p2p. Oggi la maggior parte del traffico è sul web per il trasferimento di video, e il traffico wireless sta per superare quello su reti cablate,

3.1 Domain Name System (RFC 883 del novembre 1983)

DNS è il protocollo che si occupa del mapping dei nomi logici delle macchine con i loro indirizzi IP (risoluzione DNS). Prima dell'introduzione di questo protocollo esisteva un file hosts.txt, aggiornato manualmente, che conteneva tutte le associazioni nome/indirizzo di tutte le macchine connesse ad ARPANET. Ma con la crescita della rete un sistema del genere sarebbe impossibile, sia per problemi di dimensioni sia perché un sistema così centralizzato ridurrebbe notevolmente la tolleranza ai guasti dell'intera rete. Inoltre sarebbe uno solo in tutto il mondo, ci sarebbero quindi posti più vicini e posti più lontani, e le prestazioni della rete non sarebbero uniformi.
Il server DNS utilizza la porta 53, il client è la parte che invia la richiesta di risoluzione di un nome.

3.1.1 Nomi di dominio

I nomi di dominio hanno una struttura gerarchica, e vengono gestiti da ICANN. I TLD (Top Level Domain) sono l'ultima parte di un nome e possono essere generici (.com, .net, .org) o nazionali (.it, .fr, .us).
I nomi di dominio sono suddividi in zone, ognuna con un server DNS responsabile della zona. Non esiste quindi un unico server che conosce il mapping di tutti i nomi (sarebbe come il file hosts.txt di ARPANET).

La richiesta passa attraverso:

1. local name server
2. authoritative name server
3. TLD name server
4. root name server

I root servers sono solo 13, di cui 10 negli USA, uno a Londra, uno a Stoccolma e uno a Tokyo. Hanno però diverse repliche in tutto il mondo
Quando un root name server riceve una richiesta, se conosce il mapping lo restituisce, altrimenti interroga il TLD name serve di competenza, che a sua volta interrogherà l'authoritative name server di competenza.
Il root name server può anche non farsi carico direttamente della richiesta, ma passare al richiedente il nome del TLD name sever di competenza a cui chiedere. Il modello in cui il server si fa carico della richiesta è detto **schema ricorsivo**, quello che passa l'indirizzo è lo **schema iterativo**.
I root name server preferiscono usare lo schema iterativo poiché è meno lavoro da fare per loro. I client possono provare a richiedere una query ricorsiva, ma il root name server può rifiutare.
Usando lo schema ricorsivo si alleggerisce il carico dei root name server, che hanno moltissime richieste, e si lascia la maggior parte del lavoro ai server locali, meno impegnati.

3.1.2 Caching DNS

I name server utilizzano una cache per conservare i mapping ricevuti, in modo da poter risolvere un nome rapidamente se gli viene chiesto più volte. Quando un name server di livello inferiore fornisce come risposta un mapping che aveva in cache, segnalerà al client che lui non è unserver autoritarivo per quel dominio e allega l'indirizzo del server autoritativo. Se è importante l'efficienza mi accontenterò delle risposte di cache, se è importante la precisione andrò a contattare di persona il nameserver competente. Questa funzionalità consente di effettuare attacchi detti **DNS cache poisoning** in cui vengono inseriti mapping volutamente errati per dirottare le richieste.
I dati in cache sono validi per un tempo stabilito, il **TTL (Time To Live)**, in genere di uno o due giorni. Per dati che sappiamo che cambiano raramente si può impostare un TTL più lungo.
I sistemi operativi hanno un file HOSTS che permette di conservare mapping che sappiamo che non cambiano (ma anche per impedire che alcuni server vengano contattati, associando i loro nomi ad un indirizzo IP errato).
Molti sistemi operativi hanno un meccanismo statico per la risoluzione dei nomi logici, il **file HOSTS**, che contiene alcuni mapping che quindi non andranno richiesti tramite DNS, ma saranno presi direttamente da questo file (/etc/hosts su Linux, C:\Windows\system32\drivers\etc\HOSTS su Windows)

Nslookup

Il comando nslookup, oltre alle informazioni su indirizzo IP e porte, ci può anche comunicare un **canonical name**, il nome che la macchina che stiamo contattando assume all'interno della sua rete locale: questo perché una macchina può assumere diversi nomi a seconda del servizio che offre (**alias**): ad esempio il server che ospita www.disi.unige.it ha canonical name c3po.disi.unige.it, ma per il servizio web utilizza l'alias www.

Possiamo anche ottenere più di un indirizzo IP per uno stesso nome logico: questo per alleggerire il traffico su siti molto visitati (impossibile pensare che a tutte le richieste per google.com risponda una sola macchina, l'nslookup eseguito in locale su Ubuntu (risponde da 127.0.0.1) restituisce cinque indirizzi IP per www.google.com.

Google offre un servizio di DNS pubblico (da 8.8.4.4 a 8.8.8.8), può essere utile per superare le limitazioni imposte ai DNS italiani (thepiratebay.org, kat.ph)

Formato dei messaggi DNS

Un messaggio DNS ha un'intestazione di 12 byte contenente 16 bit per identificare la richiesta, dei flags in cui è possibile specificare diverse opzioni (risposta autoritativa, metodo iterativo o ricorsivo...), e una serie di record per domande, risposte, informazioni aggiuntive e referral per server autoritativi (se la risposta è un referral, non sarà nel campo risposte ma nel campo referral autoritativi)

Le informazioni sul server cercato sono contenute nel r**esource record,** così strutturato: **name TTL class type type-specific-data**

- **name:** nome del dominio da risolvere
- **TTL**: intero a 32 bit che determina la durata (in secondi) della presenza in cache della risorsa
- **class:** generalmente IN (internet)
- type: informazioni sul tipo della risorsa (indirizzo IPv4 o IPv6, canonical name...)

Risoluzione inversa
Il DNS consente anche la risoluzione inversa, ovvero dato un indirizzo IP ci restituisce il suo dominio. Per farlo inoltranuna richiesta di tipo PTR sfruttando il dominio **in-addr.arpa**, che è parte della gerarchia dei domini: (root)=>arpa=>in-addr=>tutti i numeri che possono comporre un IP (nelle slide si capisce meglio).

3.2 Protocollo HTTP (RFC 1945 del 1992)

Si sta lavorando alla versione 2.0, ancora in fase sperimentale (non standardizzata).

È un protocollo applicativo basato su richiesta e risposta, sfrutta il TCP come protocollo di trasporto e trasmette sulla porta 80. Usa il TCP perché non vogliamo perdite di dati e perché la dimensione del messaggio non è stabilita a priori.

Il browser richiede la pagina partendo da un URL (Universal Resource Locator) che ci indica il server su cui si trova la pagina e la sua posizione nel filesystem del server. A seguito della richiesta, il protocollo DNS ottiene l'indirizzo IP del server.

Quando installiamo un web server, i file pubblicati sul web saranno nella directory /var/www

Quando una richiesta HTTP va a buon fine viene restituita una risorsa, che può esssere una pagina HTML, codice JavaScript, immagini o video ecc.

Quando digitiamo un URL nel browser viene formulata una richiesta HTTP, composta da header e body. Nell'header c'è una prima riga che specifica il tipo di richiesta che stiamo facendo: comprende metodo, URL e versione del protocollo, separati da uno spazio. Se richiediamo una versione del protocollo non supportata dal server, ci risponderà con quella precedente.

I metodi di richiesta più usati sono GET, POST e PUT.

Nelle righe dell'header vengono inclusi: host (poiché la request line include solo il percorso interno al server, senza specificarne il nome di rete), user agent, lingua in uso e altre informazioni.

Le header lines sono separate dal body da due caratteri carriage return e line feed. Il body contiene le informazioni inviate nelle richieste POST e PUT, usate quando carichiamo qualcosa. Non sempre i dati vengono inviati nel body però: per piccoli trasferimenti i dati da inviare possono essere inclusi nella request line (ad esempio quando facciamo una ricerca le keyword appaiono nella barra degli indirizzi: sono quindi state inviate nella request line).

Alla nostra richiesta il server ci invierà una risposta, strutturata in modo simile. La prima riga è la **status line**, contenente la versione del protocollo con cui ci è stata inviata la risposta, un codice di status (come il 404) e una stringa che descrive meglio l'esito della richiesta (OK, Not Found ecc.). Nelle header lines ci sono informazioni sulla risorsa e sul server, come data e ora di invio, nome del web server (Apache, Google Web Server...), ultima modifica, dimensione della risorsa, tipo di risorsa. Il body contiene la risorsa richiesta (pagina HTML, immagine ecc.)

Le informazioni su data e ora di modifica vengono date solo se la risorsa richiesta è una pagina statica. In caso di pagine dinamiche no perché non esistono sul filesystem, sono state create appositamente per noi in seguito alla nostra richiesta. Le date sono in formato standard GMT.

Appena finita la lettura dell'header, il browser inizia il rendering della pagina.

Un server web può anche essere interrogato da terminale con il comando telnet: dopo aver stabilito la connessione con **telnet nome.server.com 80** possiamo inviare a mano le richieste HTTP (bisogna essere veloci però, altrimenti scatta il timeout e il server ci chiude la connessione)

3.2.1 Prestazioni di HTTP

La principale misura delle prestazioni di HTTP è il **PLT (Page Load Time)** che dipende sia dalle caratteristiche della rete (TCP, round trip time, banda disponibile) sia della risorsa da trasferire. La versione 1.0 dell'HTTP apriva una nuova connessione TCP per ogni risorsa trasferita. Poco

performante per le pagine ricche di risorse, poiché si perde tempo a stabilire un sacco di connessioni. Le prestazioni sono state migliorate con

- **tecniche di riduzione del contenuto (minifing):** programmi che rendono il codice ultra compatto eliminando spazi, a capo, ed ogni carattere non indispensabile
- Browser che supportano fino a 6 connessioni parallele per dominio (dal 2008)
- **Connessioni persistenti e pipelining**: modifiche al protocollo che permetrono di sfruttare la banda al meglio. In particolare le connessioni persistenti sono una soluzione al problema detto prima degli sprechi di tempo per stabilire la connessione: una connessione persistente viene usata per più risorse e non viene terminata in caso di errori. Il pipelining invece serve ad inviare sequenzialmente le richieste, si può inviare una nuova richiesta senza dover prima aspettare la risposta a quella precedente.
- **Caching e proxy:** tecniche per evitare di trasmettere più volte lo stesso contenuto
- **Content delivery network (CDN):** l'idea che sta alla base delle CDN è di portare i contenuti il più vicino possibile agli utenti. Consistono in mirror (repliche) del sito in questione, messi a disposizione tramite leggere modifiche al DNS, che restituiranno una diversa risoulzione dell'indirizzo a seconda di dove si trova il client che ne ha fatto richiesta, in modo da dargli l'indirizzo del mirror più vicino. Un esempio di CDN sono le risorse di Facebook che ci appaiono con indirizzo **fbstatic-*.akamaihd.net** Il sito www.webpagetest.org ci fa vedere cosa succede quando apriamo una pagina.

Caching e proxy

Un ottimo modo per risparmiare tempo in trasferimenti inutili è il caching: ogni volta che visitiamo una pagina **statica** ne salviamo una copia locale, e per le richieste successive useremo la copia salvata invece di scaricarla nuovamente. Questo ovviamente se la pagina sul server non è cambiata, e questo posso verificarlo senza fare richieste al server: l'header Expired, infatti, include la data di scadenza della pagina, e se non è ancora passata posso usare la mia copia locale. Se quando ho scaricato la pagina il server non mi ha fornito la riga Expired nell'header, posso comunque servirmi della cache senza riscaricare la pagina: invio al server una richiesta di GET condizionale in cui dico di inviarmi la pagina solo se è stata modificata dall'ultima volta che l'ho scaricata. Se la pagina non è stata modificata, il server mi invierà una risposta 304 Not Modified, se è stata modificata mi invierà un 200 OK con la pagina richiesta. Se la copia posseduta in cache dal client è valida, la risorsa è disponibile in un RTT: infatti viene inviata la richiesta If-Modified-Since al server, che invierà la risorsa solo se è più recente della nostra copia, e se non lo è risponderà con un header 304 senza risorsa allegata.

Web proxy

Il proxy è un intermediario tra il client e il server. È utile a gestire il caching in una rete locale: se tutti i client di una determinata rete passano attraverso un proxy, questo terrà una copia cache delle pagine visitate da tutti gli utenti della rete, aumentando quindi le possibilità di hit.

Ci sono alcuni header di **Cache-Control** che possono definire se e come la pagina va memorizzata in cache.

3.2.2 Accesso con autenticazione

Un server web può decidere di non restituire alcune pagine quando vengono richieste, poiché riservate, ma di richiedere invece le credenziali per assicurarsi che il client sia autorizzato a visualizzare la pagina. Questo può essere fatto tramite speciali header HTTP (codice 401 Authorization Required), che faranno visualizzare al client una finestra di dialogo in cui inserire le credenziali. Non ha a che fare con i siti che richiedono login in una pagina (Facebook ecc.), questo è un diverso tipo di autenticazione fatta direttamente a livello di protocollo. L'amministratore di sistema del server può creare un file .htaccess, che ha l'effetto di proteggere con questo sistema la directory in cui si trova. Se l'header Authorization ha richiesto una connessione Basic username e password vengono inviati in chiaro (codificati in base 64), altrimenti vengono criptate. Le informazioni di autenticazione vanno inviate ad ogni richiesta, anche dopo che l'utente ha fatto il login.

3.2.3 Cookies

Introdotti da Netscape a metà anni '90, i cookies servono a tenere traccia dell'utente in modo che si possano avere informazioni aggiuntive non gestite dal protocollo (stato del login, carrello degli acquisti). Standardizzati dall'RFC 2965, i cookies sono file di testo contenenti nome, data di scadenza, dominio, contenuto e un attributo **secure** che specifica se il cookie può essere trasferito solo su connessioni sicure. Vengono rilasciati dai siti quando vengono visitati, attraverso l'header **Set-Cookie** incluso nella risposta HTTP.

Sono molto usati per riconoscere gli utenti a fini pubblicitari. Per motivi di privacy, il browser invia un determinato cookie solo al dominio che li ha rilasciati. Non sono sicuri per l'autenticazione. Ne vengono memorizzati al massimo 300 per browser, 20 per dominio e hanno dimensione massimo di 4 KB.

Un problema per la privacy sono i **third-party cookies**: cookies di inserzionisti che riceviamo anche se non abbiamo visitato la pagina in questione, ma basta aver visitato una pagina che aveva un'inserzione pubblicitaria di quell'azienda. Questo permette alle grandi aziende pubblicitarie di avere un'enorme collezione di informazioni, e quindi di poter offrire pubblicità personalizzate agli utenti.

3.3 Posta elettronica

Esistono diversi standard proprietari per la posta elettronica (i.e. MS Exchange), ma quello usati da tutti è la posta Internet. L'affidabilità è "best effort": il messaggio potrebbe anche non arrivare (ad esempio se la casella del destinatario è piena) ma il sistema sa gestire l'errore e avvisa il mittente del mancato recapito.

Il formato dei mesaggi è testo ASCII standard a 7 bit (ASCII senza caratteri speciali): questo perché è nata negli anni '80 (niente multimedialità) negli USA (niente caratteri speciali). L'architettura del protocollo è client/server, e come HTTP può essere usato via Telnet (se scrivi veloce, perché scattano i timeout).
Nelle slide un'immagine dell'ecosistema e-mail.
La posta elettronica è un sistema asincrono: l'invio e la ricezione di un messaggio sono scollegate tra loro.
Il client invia la mail al suo server di posta, che lo inoltra al server di posta del destinatario, che lo inoltrerà a sua volta al destinatario. Le informazioni sul destinatario vengono trovate a partire dall'indirizzo e-mail: da quello conoscerò il server ricevente, che ha le informazioni del destinatario.

3.3.1 Protocollo SMTP (RFC 821/822, aggiornati a 2821/2822)

Usa TCP come protocollo di trasporto. Usa la porta 25 per conessioni non protette, la porta 465 per connessioni protette (SSL). L'interazione client/server è basata su domande e risposte (più complesse dell'HTTP). L'invio avviene in tre fasi:

- Connection setup (greeting)
- Mail transfer (SMTP transaction)
- Connection closing

Dopo il connections setup, il messaggio viene costruito passo passo inviando al server in sequenza tutte le informazioni necessarie: MAIL TO, RCPT TO ecc, e il server ci risponderà con messaggi di ackowledgment come 250 2.1.0 Ok. Quando abbiamo finito con le intestazioni e vogliamo iniziare il messaggio vero e proprio invierò DATA. La fine dei dati è segnalata dal punto e a capo. Finito l'invio dei dati il server metterà il messaggio in coda (restituendoci anche l'ID del messaggio nella coda). A questo punto potrò mantenere la connessione se voglio inviare altri messaggi, oppure se ho finito segnalarlo al server con QUIT, che chiuderà la connessione.
Errori che si possono verificare nell'interazione SMTP sono:

- ricevente e destinatario entrambi di dominio diverso dal server
- destinatario inesistente

Sicurezza SMTP
I server SMTP un tempo erano sensibili ad attacchi chiamati **third party mail relay**: si chiede al server di inoltrare un messaggio ad un destinatario non nella sua rete, da parte di un mittente non nella sua rete. Questo perché erano aperti: consentivano a chiunque di inviare posta e non solo ad utenti conosciuti.
Per avere più sicurezza è stato introdotto l'**Extended SMTP** (**ESMTP**), che dopo la richiesta iniziale EHLO (invece dell'HELO di SMTP liscio) richiede autenticazione tramite comando AUTH. Supporta anche la comunicazione criptata tramite il comando STARTTLS (impossibile da telnet).

Struttura dei messaggi
Come in tutti i protocolli, i dati da inviare (payload) sono incapsulati in una struttura dati contenente l'header, che per le email è chiamata **envelope.** Contiene indirizzo del destinatario, infirizzo del mittente, nome del mittente, oggetto dell'e-mail e altre intestazioni. Le inforazioni contenute negli header possono essere diverse da quelle dell'envelope.

MIME (Multipurpose Internet Mail Extension)
MIME è uno standard che, tramite header aggiuntivi, estende le potenzialità dei messaggi e-mail. Permette di definire una codifica diversa dall'ASCII a 7 bit, come ASCII 8 bit, quoted printable e base64. Permette inoltre di includere allegati di vario tipo.

I dettagli sulla sintassi dei vari tipi di header e delle interazioni SMTP sono tutti nelle slides.

3.3.2 Accesso alla posta elettronica

La lezione scorsa abbiamo visto come si inviano i messaggi di posta elettronica, oggi vedremo come si scaricano. I metodi più noti di accesso all'email sono POP e IMAP. Vedremo come funziona il POP3, molto più semplice.

3.3.2.1 POP3 (RFC 1939)

È un protocollo a caratteri asincrono (come SMTP), che usa TCP come protocollo di trasporto. Utilizza la porta 110 per connessioni non cifrate, la 995 per connessioni criptate SSL. La connessione avviene in tre fasi: **authorization**, **transaction** e **update**.
L'autenticazione è fondamentale, altrimenti tutti potrebbero leggere la posta di tutti. La transazione è la fase di download/modifica dei messaggi. L'ultima fase, update, è quella in cui vengono apportate le modifiche.
Dopo l'autenticazione, con il comando LIST vedo la lista dei messaggi con la loro dimensione. Non vedo ne mittente, ne subject ne altro, solo numero e dimensione. Per scaricare un messaggio si usa RETR (numero messaggio). Per eliminare un messaggio si usa DELE, ma il messaggio non viene eliminato subito: verrà eliminato solo quando invio il comando QUIT. Per annullare una cancellazione sbagliata devo dare il comando RSET prima di uscire. Per avere la lista dei comandi supportati dal server, si invia CAPA.

3.3.2.2 IMAP4

Il protocollo IMAP usa sempre il protocollo di trasporto TCP (porta 143), e le principali differenze con pop3 sono il supporto all'accesso concorrente (più client accedono contemporaneamente allo stesso account), la gestione direttamente sul server di messaggi e stato dei messaggi, la gestione in cartelle e la possibilità di scaricare solo una parte di un messaggio MIME.

3.3.2.3 Webmail

Protocollo per la gestione della posta via HTTP, introdotta per prima da Microsoft Hotmail negli anni '90.

3.4 Protocollo NTP

NTP è il protocollo che permette di sincronizzare i clock delle macchine connesse alla rete. È un protocollo applicativo definito dall'RFC 5905 (per NTPv4) e 1305 (per NTPv3). La versione 4 di NTP utilizza il protocollo IPv6, la versione 3 IPv4. La 3 quindi, nonostante sia stata aggiornata è ancora molto utilizzata perché la maggior parte delle macchine utilizza ancora IPv4.

La versione 4 inoltre rappresenta il tempo su 64 bit invece dei 32 della 3: quest'ultima avrà un problema di overflow nel 2036 (e non nel 2038 come UNIX, poiché questa inizia a contare il tempo dal 1900 invece che dal 1970).
Abbiamo già visto che ogni computer ha un clock riservato al conteggio del tempo. È impossibile però che tutte le macchine collegate alla rete abbiano il clock perfettamente sincronizzato dall'inizio: serve quindi un protocollo che sincronizzi tutto.
Serve innanzitutto un'ora indipendente dal fuso orario: un'ora universale chiamata UTC, che sarà usata sulla rete, e dalla quale potrò risalire alla mia ora locale.
C'è anche un problema di precisione e di accumulo dell'errore: se un orologio perde un secondo al giorno dopo un mese sarà indietro di un minuto, dopo un anno di sei minuti. Dopo ripetute sincronizzazioni con un ideale orologio perfetto, riuscirò a capire se il mio orologio è troppo veloce o troppo lento e potò agire di conseguenza.
È proprio di questo che si occupa NTP: sincronizzare il clock locale della macchina e correggerne gli errori di precisione, usando come ora di riferimento quella fornita da alcuni orologi atomici collegati alla rete (lo strumento di misurazione del tempo più preciso esistente). Esiste una gerarchia di server NTP decisa in base alla precisione dell'orologio utilizzato. La gerarchia è stabilita da un numero detto **strato**, in cui lo strato 0 sono gli orologi atomici, che vengono collegati ad un server attraverso reti che non utilizzano il protocollo IP, poiché ho bisogno di maggiore precisione. Questo server sincronizzerà il suo orologio con l'orologio atomico, e avrà strato 1 nella gerarchia di NTP. Ci possono essere fino a 15 livelli nel protocollo NTP: il livello 16 è quello di un sistema mai sincronizzato con NTP. Nel dipartimento di Fisica c'è un server di livello 2.
Finora abbiamo detto che il livello 0 è un orologio atomico: in realtà ci sono altre fonti primarie valide ma più economiche come ad esempio i ricevitori GPS: i satelliti del GPS hanno infatti a bordo un orologio atomico. Avremo una misura estremamente precisa se teniamo conto anche del ritardo dovuto alla trasmissione del messaggio dal satellite.
NTP è un protocollo a domanda/risposta basato su UDP, quindi nella sincronizzazione dovrò anche tenere conto del tempi di ritardo dovuto al passaggio della risposta sulla rete. Per farlo con la massima precisione possibile, è necessario avere:

- un tempo t1 di invio della richiesta dal client
- un tempo t2 di ricezione del messaggio dal server
- un tempo t3 di invio del messaggio dal server
- un tempo t4 di arrivo della risposta al client

Avendo questi dati, il client può calcolare il RTT con la formula **RTT=[(t4-t1)-(t3-t2)]/2** Se il mio clock locale è sincrono col server interrogato varrà la condizione **t4=t3+D**. Il protocollo prevede che si contattino almeno 3 server a livelli diversi per poter calcolare l'ora.
Si stabilisce un intervallo di tempo abbastanza breve, nell'ordine del minuto, e ogni volta scaduto queso intervallo il client NTP si sincronizza inviando la richiesta ad un numero di server stabilito.
Abbiamo detto che la connessione avviene tramite UDP: ci possono quindi essere anche dovuti all'inaffidabilità del protocollo di trasporto. La quantità di risposte corrette ricevute e l'entità dei ritardi permettono di stabilire localmente una gerarchia, cui in cima ci saranno i server che mi danno il maggior numero di risposte con il minor ritardo. La variabilità dei ritardi è significativa e può essere indice di sovraccarico (del server o della rete), tenderò quindi a preferire i server con una bassa variabilità del ritardo, anche se magari il ritardo è più grande: questo perché con un ritardo costante la correzione sarà più affidabile.

Controllando se dopo un po' di tempo dalla sincronizzazione il mio clock è avanti o indietro, potrò sapere se va troppo veloce o troppo lento e regolarne la velocità di conseguenza. Una volta raggiunta una situazione stabile, il numero di richieste di sincronizzazione NTP viene diminuito.
Su Linux le sincronizzazioni NTP sono gestite dal daemon **ntpd**. Posso interrogare il suo stato con il comando **ntpdc**.

Sicurezza
Esiste la possibilità di firmare digitalmente i messaggi NTP per essere sicuri di stare comunicando realmente con un server NTP e non con un impostore. Tuttavia l'apposizione della firma digitale influisce negativamente sui tempi, quindi le versioni sicure di NTP sono molto poco utilizzate, viste le scarse possibilità di un attacco di questo tipo si è preferita l'efficienza alla sicurezza.

4. Livello di rete

È il livello che mette in collegamento le varie sottoreti. È comune a tutti: reti Wi-Fi, Ethernet e di altro tipo comunicano tutte con lo stesso protocollo di rete.
Le funzioni chiave del livello di rete sono il **routing** e il **forwarding**: il routing determina il percorso dei pacchetti dall'origine alla destinazione, il forwarding trasferisce i pacchetti dal link in entrata di un router al giusto link in uscita. A questo livello si indirizza solo con l'indirizzo IP: le porte non contano, se ne occuperà il protocollo di trasporto.
L'operazione di lookup è quella che, guardando l'indirizzo di destinazione di un pacchetto, controlla nella tabella di forwarding su quale link deve essere instradato il pacchetto destinato a quell'indirizzo
Quando è stato progettata l'infrastruttura di rete, si è dovuto scegliere se usare un approcio connection oriented o no: con un approccio orientato alla connessione è necessario stabilire a priori il percorso attraverso i router di un pacchetto (come in una rete telefonica), con un approccio datagram (connectionless) no. Si è optato per un approccio datagram, più dinamico e meno sensibile ai guasti.
Il protocollo di rete non ha consegna garantita, né assicura che i messaggi vengano recapitati nel giusto ordine, e non garantisce neanche un'ampiezza di banda minima. Ha però un meccanismo di checksum per assicurare l'integrità dei messaggi. Abbiamo visto che il checksum c'è anche a livello di trasporto: questo perché quando i singoli protocolli sono stati progettati non si sapeva quali altri controlli ci sarebbero stati da parte degli altri protocolli dello stack. Attualmente il più efficiente controllo di integrità è a livello datalink (Ethernet). Le garanzie di consegna e di tempi non garantite dal protocollo IP sono invece offerte da protocolli realtime per usi critici.

Router
Il router è il dispositivo che si occupa di instradare i pacchetti. Riceve i dati dal livello fisico, li processa a livello datalink, ne determina la porta di uscita (forwarding) e li invia. Se il tasso di ricezione è superiore a quello di invio si formeranno delle code di uscita, che se diventano troppo lunghe possono causare perdite di dati per saturazione dei buffer.

Address lookup
È l'operazione di controllo nella tabella di forwarding del corretto link di uscita per un dato pacchetto, basandosi sull'indirizzamento IP. È diventato un collo di bottiglia delle comunicazioni di rete: mentre le altre funzionalità si sono evolute, il lookup continua ad impiegare più tempo.

4.1 Classi di indirizzamento IPv4

Un indirizzo IPv4 è un numero di 32 bit, raggruppati in blocchi da 8 bit. È suddiviso in due parti, **subnet** e **host**, che identificano rispettivamente sottorete e macchina. La parte di sottorete è la più importante, quella utilizzata dai router intermedi per instradare il pacchetto, e solo quando si arriva al router della sottorete di destinazione si andrà a vedere la parte che indica la macchina di destinazione.
Quando configuriamo manualmente una connessione di rete, ci servono:

- il nostro indirizzo IP
- la maschera di sottorete
- l'indirizzo del DNS
- l'indirizzo del gateway (router di uscita della nostra sottorete)

Queste informazioni vengono configurate automaticamente tramite protocollo DHCP.
Il protocollo IPv4 consente di avere 2^{32} indirizzi diversi, che adesso sono finiti. L'esaurimento degli IPv4 è stato molto rallentato dall'introduzione di DHCP, indirizzi privati e indirizzi senza classe. Le classi stabiliscono il numero di bit riservati per parte di rete e parte di host e sono suddivisi in:

- **Classe A:** reti fino a 2^{24} macchine
- **Classe B:** reti fino a 2^{16} macchine
- **Classe C:** reti fino a 2^{8} macchine

L'esponente del 2 nelle macchine indirizzabili corrisponde al numero di bit riservato al campo host. Questa gestione comporta un grande spreco di spazio, perché un organizzazione che compra un blocco di indirizzi di una certa classe non è detto che poi usi tutti i 2^{x} indirizzi che gli sono stati riservati, soprattutto se si tratta di blocchi di classe A. Questi indirizzi non saranno comunque disponibili all'assegnazione per altri soggetti in quanto riservati a chi ha comprato il blocco. Inoltre questa schema rendeva anche molto inefficiente l'address lookup, perché dovendo aggiungere una nuova entry alla tabella di forwarding per ogni nuovo indirizzo creato, c'erano un sacco di entry inutili dovute a indirizzi assegnati ma non utilizzati.
La maschera di rete serve a capire in che tipo di rete ci troviamo: è come un indirizzo composto solo da blocchi di 8 bit di soli 1 (255) o di soli 0: facendo l'AND di un indirizzo IP con la sua maschera di rete ci rimarrà solo la sua parte subnet.

Indirizzamento IPv4 senza classi
L'indirizzamento senza classi CIDR permette maggiore flessibilità sui prefissi di rete, che possono essere di lunghezza arbitraria. Questo è possibile specificando ogni volta il numero di bit della parte di subnet e il numero di bit della parte di host. In questo schema la subnet mask non è necessariamente composta solo da 255. Avendo i prefissi lunghezza arbitraria, l'address lookup si complica.

4.2 Indirizzamento IPv6

Si è reso necessario a causa dell'esaurimento degli indirizzi IPv4. È un numero a 128 bit che permette di avere 655571 miliardi di miliardi di indirizzi per ogni metro quadrato di superficie terrestre, scongiurando quindi il problema dell'esaurimento degli indirizzi anche sul lungo termine.
Le macchine attuali gestiscono entrambi i protocolli IPv4 e IPv6, poiché siamo ancora in fase di transizione. Si usa la soluzione **dual stack**, che permette di passare agevolmente da un protocollo di indirizzamento all'altro nel caso in cui, durante il percorso di un pacchetto, si incontri un router abilitato solo all'IPv4. Un'altra tecnica è quella del **tunneling**: se nel percorso incontriamo nodi che non conoscono IPv6, il datagramma IPv6 diventa payload di un datagramma IPv4 permettendo al pacchetto di transitare.

Assegnazione degli indirizzi
I blocchi di indirizzi vengono gestiti da IANA, che delega l'assegnazione a cinque register (uno per continente). Oggi è possibile richiedere solo IPv6, visto che gli IPv4 sono esauriti

4.3 Struttura degli header IP

L'intestazione di un datagramma IPv4 contiene:

- versione del protocollo IP utilizzata
- lunghezza dell'header
- lunghezza del datagramma (abbandonato in IPv6)
- tipo di servizio (non più utilizzato)
- identificatore e offset per la frammentazione (situazione in cui un datagramma è troppo grande per essere inviato come singolo pacchetto a livello datalink, e quindi viene frammentato)
- time to live: numero massimo di passaggi da un router all'altro
- upper layer: protocollo di trasporto utilizzato (6=TCP, 17=UDP, 1=ICMP)
- checksum sull'header
- indirizzi IP sorgente e destinazione
- altre opzioni

4.3.1 Frammentazione e negoziazione MTU

Ogni canale di comunicazione ha un suo MTU (Maximum Transfert Unit): questo dipende dal livello datalink e non dal livello IP, quindi può capitare che un pacchetto IP sia più grande del MTU del protocollo datalink su cui deve transitare. I router possono quindi frammentare i datagrammi IP se sono troppo grandi per essere gestiti come un singolo pacchetto: il riassemblaggio viene poi fatto dall'end system. Per frammentare, il router suddivide il payload in più datagrammi IP, ognuno con la sua intestazione. I flag More Fragment e Don't Fragment ci dicono se il pacchetto un frammento o no: in particolare More Fragment è impostato a zero nell'ultimo frammento.
L'identifier è un numero a 16 bit che identifica univocamente il frammento.
In caso di perdita di frammenti, il router non se ne occupa: la situazione sarà gestita dal protocollo di trasporto, che chiederà la ritrasmissione se ha un meccanismo di timeout. È possibile un attacco DoS (Jolt2) che consiste nell'inviare frammenti a caso.

Negoziazione MTU
La tecnica della frammentazione è inefficiente perché obbliga l'end system a ricostruire il pacchetto. Oggi si preferisce negoziare un MTU minimo su un dato percorso per evitare la frammentazione: questo viene richiesto impostando a 1 il flag Don't Fragment, e in questo caso verrà cercato l'MTU più basso nel percorso che il pacchetto deve fare. Questo richiede uno scambio di messaggi tra i vari nodi del percorso, che avvengono su protocollo ICMP.

4.4 Protocollo DHCP
Il DHCP è un protocollo che permette di ottenere automaticamente un indirizzo IP dinamico. Utiizza il protocollo di trasporto UDP, perché trasmette messaggi molto piccoli che possono andare persi (se l'indirizzo non arriva, la richiesta viene reiterata dopo un timeout). Il server è in ascolto sulla porta 67, il client usa la porta 68.

4.4.1 DORA (Discover, Offer, Request, Ack)
Quando un nuovo client entra in una rete locale, invia un messaggio broadcast (all'indirizzo 255.255.255.255:67) di "**DHCP discover**", che verrà ricevuto da tutte le macchine collegate alla rete locale. In questo momento il client non ha ancora un indirizzo IP, quindi la risposta del server (**DHCP offer**) viene inviata al MAC address della scheda di rete del nuovo client. Non è anch'esso un messaggio broadcast, viene ricevuto solo dal client interessato.
Ma nella rete potrebbero esserci più server DHCP, quindi ad un DHCP discover potrebbero rispondere più server, offrendo naturalmente IP diversi. Il client dovrà quindi inviare un altro messaggio broadcast (**DHCP request**) in cui dice quale offerta ha accettato, in modo che tutta la rete sappia il suo IP. Il server che ha assegnato l'IP invia quindi il **DHCP ack** con alcuni parametri di connessione (server DNS ecc.). L'ack viene inviato ancora usando il MAC address e non l'IP, perché in questa fase le tabelle ARP di corrispondenza indirizzo IP-indirizzo MAC non sono ancora state aggiornate, il pacchetto non potrebbe essere instradato correttamente.

4.5 NAT (Network Address Translation)
Un router NAT è un dispositivo che appare all'esterno non come router ma come singolo dispositivo con un unico indirizzo IP, nonstante abbia dietro una rete. Serve per collegare una rete locale che utilizza indirizzi IP privati con l'esterno: gli IP privati infatti non esistono al di fuori della loro rete locale. Questo sistema migliora la sicurezza della rete, sia perché nasconde i dispositivi che ne fanno parte al mondo esterno sia perché i sistemi di sicurezza saranno messi tutti sul NAT: è come avere un portinaio. Facilita inoltre la gestione della rete perché posso cambiare gli indirizzi IP privati come voglio senza che questo influisca sulle comunicazioni con l'esterno.
Quando inviamo un pacchetto verso l'esterno, il router NAT sostituisce nel datagramma l'indirizzo IP sorgente (privato) con l'indirizzo IP pubblico del router e la porta specificata dal client con una sua porta (ephimeral). In questo modo sa che i pacchetti ricevuti su quella porta sono destinati a quel client della LAN. È un mapping porta-indirizzo IP, e le porte sono numeri a 16 bit, quindi un singolo NAT può indirizzare 65000 connessioni con un solo IP pubblico.
L'utilizzo del NAT è contestato dai puristi della rete perché usa le porte per indirizzare gli host, mischia i livelli dello stack, interferisce nella comunicazione end-to-end e perché la vera soluzione all'esaurimento degli IPv4 è l'adozione di IPv6.

4.6 Protocollo ICMP

Sappiamo che IP è un protocollo inaffidabile: non ha modo di accorgersi se un pacchetto arriva o no. È inoltre soggetto a errori dell'utente, se ad esempio invio un pacchetto su una porta chiusa. Il protocollo ICMP serve a gestire i messaggi informativi e di errore tra le varie componenti della rete. I messaggi ICMP sono payload di un messaggi IP, con campo upper layer=1, e contengono informazioni sul pacchetto che ha causato l'errore.
I messaggi ICMP possono essere di errore o informativi: questi ultimi permettono di fare test sul funzionamento della rete come ping e traceroute. A differenza del programma che abbiamo sviluppato in laboratorio, le vere utility ping lavorano a livello di rete, sul protocollo ICMP.
L'utility traceroute invece serve ad ottenere il nome di tutti i router incontrati nel percorso di un pacchetto, e lo fa inviando una serie di datagrammi IP su porte casuali con TTL crescenti: la porta casuale scelta sarà quasi sicuramente chiusa, quindi l'host contattato risponderà con un messaggio ICMP di porta non disponibile.

4.7 Algoritmi di routing

Gli algoritmi di routing si occupano di creare le tabelle di forwarding che dicono al router su quali canali vanno instradati i datagrammi.
Internet è un grafo estremamente complesso: perché funzioni a dovere quindi servono algoritmi estremamente efficienti. È gestibile perché è organizzata gerarchicamente: se fosse piatta sarebbe impossibile trovare i cammini da un nodo all'altro. In oiù ci sono degli **hub**, nodi speciali che accorciano i cammini. Internet quindi è un grafo enorme ma con un basso diametro (diametro=cammino minimo più grande all'interno di una rete).

Modelli di consegna

Finora abbiamo visto solo il modello di consegna **unicast**: un solo mittente che comunica con un solo ricevente. Ma esiste anche il modello **broadcast**: un solo mittente invia un messaggio a tutti i nodi della sua rete: lo abbiamo visto nella fase discover del protocollo DHCP. L'indirizzo per i messaggi broadcast è 255.255.255.255, con indirizzo MAC FF:FF:FF:FF:FF:FF. Con il **multicast** invece si contattano solo alcuni nodi della rete. Il modello **anycast** invece si usa quando abbiamo più nodi equivalenti e vogliamo che sia sempre contattato il più vicino: è usato dalle CDN.

Proprietà degli algoritmi

Un buon algoritmo di routing deve avere le seguenti proprietà:

- **correttezza**: se un cammino tra due nodi esiste, l'algoritmo deve trovarlo
- **efficienza**: trovare sempre i cammini più efficienti
- **fairness**: tutte le richieste devono essere eseguite, senza preferenze
- **fast convergence**: recupero veloce in seguito ai cambiamenti della rete
- **scalabilità**: se la rete aumenta di dimensioni l'algoritmo continua a funzionare correttamente

Un algoritmo di routing può essere **globale** o **decentralizzato**: un algoritmo globale ricevedin input tutti i nodi e i costi dei link tra i nodi e calcola il cammino a costo minimo, un algoritmo decentralizzato invece produce un vettore di stima dei costi verso gli altri nodi della rete in modo distribuito e iterativo. Possono essere ulteriormente classificati in **statici** e **dinamici**: un algoritmo statico cambia i cammini molto raramente (di solito è l'amministratore che li modifica

manualmente), un algoritmo dinamico determina di volta in volta i cammini in base al volume del traffico e allo stato della rete (guasti, ingorghi ecc).
Un buon algoritmo deve scegliere sempre i cammini migliori, che sono definiti da:

- bassa latenza: si evitano percorsi inutili e cicli
- banda massima: si evitano i link più lenti
- economia: si evitano i link costosi
- numero minimo di salti da un host all'altro

Algoritmi link-state
Un algoritmo link state richiede che la topologia della rete e il costo di ogni link siano noti a tutti i nodi. Un esempio di algoritmo link-state è l'**algoritmo di Dijkstra** che, preso un nodo di partenza, crea l'albero di copertura, cioè i cammini a costo minimo da quel nodo a tutti gli altri nodi della rete, e con questo costruisce localmente la tabella di forwarding. Per fare questo è necessaria una fase iniziale di propagazione delle informazioni sulla topologia della rete: non posso calcolare localmente l'albero di copertura altrimenti. Dovranno quindi essere inviate in broadcast le informazioni sui costi dei link. E' un algoritmo iterativo: dopo k iterazioni conosce i cammini per i nodi a distanza k dall'origine. I collegamenti di cui non si conosce il costo vengono considerati di costo infinito. ***//spiegato molto meglio nelle slides***

Algoritmo distance vector
Con un algoritmo distance vector ogni router si costruisce una propria tabella di instradamento basandosi esclusivamente sulle informazioni raccolte dai suoi vicini. Si basa sulla formula di Bellman-Ford per il calcolo dei cammini minimi. Costruisce un vettore delle distanze contenente le stime dei costi dei percorsi a costo minimo dal nodo origine a tutti i nodi della rete, e lo scambia con i suoi vicini in modo da poter imparare nuovi percorsi. Quando viene ricevuto un nuovo distance vector

Instradamento gerarchico
Gli algoritmi che abbiamo visto (nelle slide) per quanto efficienti non potrebbero funzionare su centinaia di milioni di nodi. Per questo gli algoritmi vengono eseguiti all'interno di ogni AS (autonomous system). Ci sono quindi **Interior Gateway Protocols** per il routing all'interno di un AS ed **Exterior Gateway Protocols** per la comunicazione tra AS diversi. In particolare gli IGP possono essere diversi in ogni AS, ma il protocollo per la comunicazione tra AS deve essere lo stesso per tutti.

Policy
Un ISP può decidere di offrire un servizio di transit o di peering per i suoi clienti. Il transit consiste nel far passare il traffico del cliente da e verso la rete, il peering avviene tra ISP e consiste nel consentire lo scambio gratuito di dati tra due reti, ma senza uscire all'esterno. Se due reti A e B sono in peering tra loro, potranno scambiarsi gratuitamente dati ma non potranno inoltrare dati verso l'esterno attraverso l'altra rete. Queste non sono solo impostazioni tecniche, ma derivano da accordi commerciali. I router dovranno comunque essere in grado di rispettare le policy.

4.7.1 RIP (RFC 1058, 2043)

È un protocollo distance vector che utilizza il protocollo di trasporto UDP, sulla porta 520 (521 per la versione v6). Utilizza il numero di hop[3] come metrica di costo, ne consente al massimo 15. I router adiacenti si scambiano aggiornamenti (detti **RIP advertisement**) ogni 30 secondi, che contengono un elenco di sottoreti raggiungibili all'interno dell'AS (al massimo 25, probabilmente per non eccedere la dimensione massima dei pacchetti UDP). In caso di errori nelle tabelle di routing, RIP potrebbe erroneamente instradare un pacchetto in un loop, che lo farebbe tornare al mittente. I router possono anche richiedere informazioni sui cammini ai router vicini tramite **RIP request messages**.
Se dopo 180 secondi un router non riceve informazioni da un suo vicino, lo considererà spento o guasto e modificherà la tabella di routing per eliminare il router guasto. Questa modifica verrà poi propagata a tutta la rete tramite RIP advertisement.
I problemi principali del protocollo RIP sono la lentezza dei router nel calcolo delle tabelle (slow convergence) e la possibilità di finire in un loop (count to infinity). Per questo RIP è utilizzato solo nelle reti di piccole dimensioni, per quelle grandi si usa OSPF.

4.7.2 OSPF

È un protocollo link state che usa la tecnica del **flooding** per inviare le sue tabelle agli altri nodi e l'algoritmo di Dijkstra per calcolare i cammini minimi. Le sue specifiche sono pubbliche (la O sta per Open).
Ogni volta che un link cambia, l'informazione viene inviata a tutti gli altri router della rete con messaggi trasportati direttamente da IP, com upper layer impostato a 89.
A differenza di RIP, OSPF non specifica come calcolare il costo dei link: sarà stabilito dall'amministratore di rete. Permette di **autenticare** i messaggi OSPF con password o hash MD5, e supporta il **multipath**: se ci sono più percorsi disponibili posso utilizzarli tutti, per aumentare il througtput.

4.7.3 BGP

Lo standard per la comunicazione tra AS è il protocollo BGP, che utilizza il protocollo TCP sulla porta 179 con connessioni semipermanenti. I messaggi scambiati, detti BGP route, contengono prefisso IP, path vector e next hop, e viaggiano in direzione opposta al traffico. Un path vector descrive un elenco di AS che permettono di raggiungere il prefisso IP speficiato (evitando loop). I router BGP si trovano ai confini degli AS e propagano gli annunci BGP a tutti i router interni. La validità di un percorso è determinata in base alle informazioni di raggiungibilità e alle policy tra AS.

4.8 Protocollo IPv6

Nasce a causa dell'esaurimento degli indirizzi IPv4, ma porta anche divers migliorie come la semplificazione dell'header per velocizzare elaborazione ed inoltro ed agevolare la qualità di servizio: gli header IPv6 infatti hanno flag per il traffico critico, da smistare prima (aeroporti ecc.). Sono inoltre stati rimossi alcuni campi inutilizzati per vaere una dimensione fissa dell'header di 40 byte, in ogni situazione. Il payload è di 64 KB.

[3] **Hop**: singolo passaggio da un punto all'altro di un cammino

Adesso siamo in una fase di transizione in cui i protocolli IPv6 e IPv4 convivono: per evitare incompatibilità si sfrutta il **tunneling**: quando un pacchetto IPv6 si trova a dover transitare per dei router compatibili solo con IPv4, il pacchetto IPv6 diventa payload di un pacchetto IPv4 che passerà per i router non aggiornati.

Header IPv6

- **version**: 6
- **Diff. Serv**.: priorità dei datagrammi del flusso
- **Flow label**: identifica i pacchetti che appartengoni ad un flusso
- **Next header**: identifica il protocollo di trasporto
- **Hop limit:** equivalente al Time To Live dell'IPv4, il nome è stato cambiato perché era ambiguo: questo campo non indica infatti un tempo, ma il numero massimo di passaggi da un router a un altro (hop) consentiti. Campo di 8 bit
- **Source e destination address**: numeri di 128 bit costituiti da 8 gruppi di 4 cifre esadecimali

Altre novità
Per motivi di efficienza dei router, non è prevista frammentazione durante il cammino. Se un router riceve un pacchetto troppo grande per i suoi link di uscita, lo scarta e restituisce al mittente un messaggio ICMP di Packet too big. È stato rimosso anche il campo checksum, visto che tanto è implementato anche sugli altri livelli dello stack di Internet. È stato eliminato anche il campo Option, ma può comunque essere indirizzato nel campi Next header.
Un router può non ricevere informazioni per lungo tempo da un vicino se questo è andato giù, ma anche se le tabelle di routing non vengono cambiate (non ci sono nuove informazioni da trasmettere). Per questo esiste il messaggio di KEEPALIVE, per evitare che un router venga considerato guasto quando non ha bisogno di trasmettere nuove tabelle di routing per tanto tempo.

5. Livello datalink

Il livello link si occupa della comunicazione sul singolo canale, da un punto all'altro (che sia un host o un router a questo livello non importa). Dipende dal mezzo su cui avviene il collegamento: cablato o wireless. È implementato in parte a livello software dal driver della periferica di rete e in parte dalla scheda di rete stessa. A questo livello i pacchetti sono chiamati **frame**, suddivisi in **header**, **payload** e **trailer**.
Il livello link riceve dal livello fisico una sequenza di bit: un flusso continuo di 0 e 1 in cui bisogna riconoscere dove iniziano e finiscono i vari frame. Per questo fin dagli anni '70 sono state fatte varie proposte: una di queste è il **byte count** in cui il primo byte di ogni frame indica la sua lunghezza. Questo mi impone una limitazione della dimensione dei frame di 255 byte (un byte rappresenta al massimo il numero 256, da cui devo togliere 1 perché il primo byte è quello della lunghezza). Inoltre, un byte di lunghezza corrotto compromette la lettura di tutto il flusso.
È stato proposto allora il **byte stuffing**: utilizzare caratteri speciali come flag che indicano inizio e fine del frame. Rende necessario l'utilizzo di caratteri di escape nel caso in cui questi caratteri

speciali si trovino anche nel payload, per fare in modo che vengano interpretati come dati del payload e non come flag.
Con il **bit stuffing** invece si usano sei bit 1 consecutivi come flag di inizio e fine frame, e nel payload metterò uno zero ogni 5 bit, che verrà rimosso dal ricevente: in questo modo se trovo sei 1 consecutivi sono sicuro che sia un flag di inizio o fine frame.
La trasmissione però è molto sensibile agli errori: i sengali infatti sono un'onda che viene interpretata come 1 o 0 a seconda della sua intensità: idealmente dovrebbe essere un'onda quadra, in pratica il rumore increspa questa linea, a volte fino a renderne difficoltoso o errato il riconoscimento (immagine nelle slides). Gli errori vanno quindi gestiti, ma in modo molto diverso da uanto avveniva ai livelli superiori: non dobbiamo affrontare errori di routing per cui un pacchetto mi arriva prima di un altro, ma bit errati.
Ci sono codifiche di **error detection** e di **error correction**, entrambe aggiungono bit di controllo per riconoscere o correggere gli errori: il mittente costruirà attraverso una funzione una codeword di bit di controllo, a partire dal frame da trasmettere. Il mittente dovrà ricalcolarla usando la stessa funzione, e se la codeword calcolata è uguale a quella ricevuta non ci sono stati errori, altrimenti si. Ethernet utilizza il sistema di controllo **CRC-32 (Ciclyc Redundancy Check)**.

5.1 Protocollo ARP

Il protocollo ARP (Address Resource Protocol) serve a mappare gli indirizzi IP con gli indirizzi MAC degli adattatori (schede di rete). Gli indirizzi fisici sono composti di 48 bit, rappresentati in esadecimale. L'indirizzo di broadcast è FF:FF:FF:FF:FF:FF. Il protocollo ARP è l'analogo del DNS a livello link: se invio un pacchetto ad un indirizzo della mia stessa sottorete devo incapsularlo in un frame contenente il MAC del destinatario, ma se devo inviarlo ad un'altra sottorete devo scoprire il MAC del gateway di quella rete attraverso ARP.
In una rete locale ogni nodo ha un modulo ARP: quando un nodo vuole sapere l'indirizzo MAC di un altro nodo (di cui conosce l'IP) invia un messaggio broadcast nella rete contenente l'indirizzo IP del destinatario (**ARP request**). Ogni macchina appena ricevuta l'ARP request confronterà il proprio indirizzo IP con quello specificato nella request, e solo la macchina che ha l'indirizzo IP richiesto risponderà con un'**ARP reply** contenente il suo MAC. Adesso il richiedente avrà salvata nella sua tabella ARP una corrispondenza IP-MAC, e non avrà più bisogno di inviare richieste ARP per raggiungerlo. Con il comando arp visualizziamo la tabella di corrispondenze IP-MAC memorizzate sulla nostra macchina, o definire una nuova corrispondenza manualmente con l'opzione -s.
Esisteva anche una risoluzione inversa MAC-IP, tramite protocollo RARP (Reverse ARP), ma i nuovi kernel non lo supportano più.
È possibile un attacco di **ARP poisoning** per intercettare il traffico destinato ad altri: consiste nell'inviare ARP reply errate contenenti il proprio MAC invece di quello del destinatario.

5.2 Ethernet

L'ethernet classica, diversa da quella attuale, si impose come standard negli anni '80/'90 soprattutto grazie al suo basso costo rispetto ad altri standard. I dati sono suddivisi in frame, a loro volta suddivisi in campi:

- **preamble:** 8 byte per attivare gli adattatori dei riceventi avvisandoli di accendersi perché stanno arrivando dati
- **destination e source address:** due campi distinti contenenti gli indirizzi MAC del mittente e del destinatario (6 byte)

- **type:** protocollo di destinazione (solitamente IP, ma può essere anche ARP o altri)
- **payload**
- **CRC**

È un protocollo connectionless e inaffidabile: non esistono meccanismi di ACK o di timeout. È un protocollo ad accesso multiplo: i nodi comunicano su un canale di broadcast condiviso, uno stesso cavo su cui viaggiano tutti i segnali. Ogni macchina catturerà solo quelli destinati a se, e i messaggi broadcast. Essendo il canale condiviso, se due nodi trasmettono contemporaneamente avviene una collisione e il segnale si corrompe, e per evitarli viene usato il protocollo **CSMA (Carrier Sense Multiple Access).** Ogni nodo ascolta il canale prima di trasmettere: se è libero procede, altrimenti ritarda la trasmissione. Potrebbero comunque avvenire collisioni se due nodi ascoltano il canale libero contemporanamente, e quindi iniziano la trasmissione contemporaneamente. Per questo il protocollo è stato aggiornato al **CSMA/CD (Collision Detection)**, che rileva se è in corso una collisione e nel caso invia un segnale di disturbo (jam) per annullare la trasmissione, e riprova dopo un certo ritardo. Ma se il ritardo fosse uguale per entrambi i nodi saremmo punto e a capo: per questo il ritardo è calcolato da un algoritmo di **exponential backoff**: dopo la collisione si sceglie tra k={0,1} e ritrasmette dopo k volte il tempo necessario a trasmettere 512 bit. Questo alla prima collisione: alla seconda avremo k={0,1,2,3}, e così ad ogni collisione, raddoppiando ogni volta le possibilità di scegliere il ritardo. Se avvengono più di 10 collisioni ci si ferma ad un massimo di 1024.

5.2.1 Modern Ethernet

L'ethernet moderna non utilizza più un canale di broadcast unico ma si serve di switch per indirizzare più canali tra un nodi e l'altro. Ha dei buffer di invio e di ricezione (critici per la perdita di frame, quando sono pieni). Ogni switch ha la sua **tabella di switch** che contiene le associazioni indirizzo MAC-canale fisico. Sono costruite dallo switch in modo automatico osservando il traffico: ogni volta che qualcuno si collega alla rete lo switch lo aggiunge alla tabella. I nodi non vedono più il traffico di tutti, ma solo quello a loro destinato e il traffico broadcast: per questo un attacco efficace punta a prendere il controllo dello switch. Una versione stupida dello switch è l'hub, che si limita a ripetere tutto il traffico che riceve su tutti i suoi canali.

5.3 Protocolli wireless

Su reti wireless la comunicazione è più complicata: un nodo può o inviare o ricevere ma non due cose contemporaneamente, quindi un meccanismo per evitare le collisioni come il CSMA/CD non è applicabile. In più la comunicazione è tutta forzatamente broadcast: il segnale si propaga nell'etere e tutte le macchine nel raggio di azione possono riceverlo. Una situazione a rischio è quella dei dispositivi nascosti: due nodi A e B non si vedono tra loro, ma vedono entrambi un nodo C: se provano a comunicarvi entrambi contemporaneamente si verifica una collisione. Nodi esposti copia da slide.

MACA (Multiple Access Collision Avoidance)

È un protocollo che usa un breve handshake per la connessione per evitare collisioni. Il mittente invia una RTS (Request To Send) con la lunghezza del frame da spedire, il mittente risponde con un CTS (Clear To Send) in cui ripete la lunghezza del frame, e a questo punto tutti i nodi che hanno ricevuto il CTS non trasmettono. La collisione è ancora possibile, non più sui dati ma sui messaggi dei controlli: è rapidamente verificabile però dato che sono molto brevi.