# Sys&Net Lecture notes

Leonardo Biason

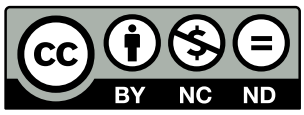
September - December 2023

# Contents

# Chapter 0
## About these notes

## 0.1 License






The decision of copyrighting this work was taken since these notes come from **university classes**, which are protected, in turn, by the **Italian Copyright Law** and the **University's Policy** (thus Sapienza Policy). By copyrighting these works I'm **not claiming as mine** the materials that are used, but rather the creative input and the work of assembling everything into one file.

All the materials used will be listed here below, as well as the names of the professors (and their contact emails) that held the courses.

The notes are freely readable and can be shared, but **can't be modified**. If you find an error, then feel free to contact me via the socials listed in my website. If you want to share them, remember to **credit me** and remember to **not** obscure the **footer** of these notes.

Those notes were made during my first year of university at Sapienza. These notes **do not** replace any professor, they can be an help though when having to remember some particular details. If you are considering of using *only* these notes to study, then **don't do it**. Buy a book, borrow one from a library, whatever you prefer: these notes won't be enough.

I hope that this introductory chapter was helpful. Please reach out to me if you ever feel like. You can find my contacts on my website. Good luck!

Leonardo Biason

## 0.2 Bibliography and references

[1] Arpaci-Dusseau, Remzi H., Andrea C. Arpaci-Dusseau. (2023). *Operating Systems: Three Easy Pieces, 1.10*. Arpaci-Dusseau Books

[2] Jim Kurose and Keith Ross. (2020). *Computer Networking: a Top Down Approach*. Pearson



| Professor's name | Contact | Teaching Course |
|---|---|---|
| Gabriele Tolomei | gabriele.tolomei@uniroma1.it | Systems & Networking<br>Unit 1<br>2023-2024 |
| Novella Bartolini | novella.bartolini@uniroma1.it | Systems & Networking<br>Unit 2<br>2023-2024 |



Inside those notes, there are multiple boxes, and each of them has a different meaning. Based on the color of their stripe on the left, boxes can be separated into 4 macro categories

Theory boxes

Practice boxes

Remarks boxes

Curiosities boxes

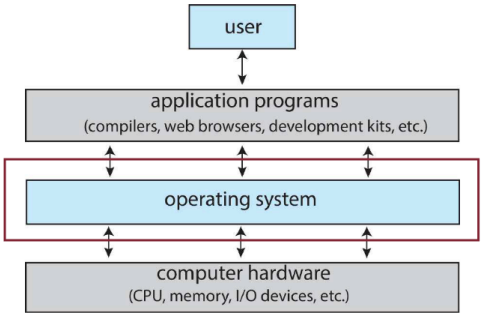
# Chapter 1
## Operating Systems (OS)

Each computer is based on some hardware, and the way to control such hardware is usually through an **operating system**. But what is an operating system? There is no universally accepted definition, but we can try to define a general definition:

**Definition:** Operating System (OS)

An **operating system** (**OS**) is the implementation of a virtual machine that is "somehow" easier to program than the bare hardware

An operating system should not particularly worry about the hardware where it's running: it should target a certain type of machine, without making a different version for each possible machine. An OS must be programmed with **abstraction** in mind.




An OS should be able to manage multiple users and multiple processes of any kind. Abstraction is necessary also for that: the OS should not limit the applications that run on the machine.

Also, what features an OS may or may not have depends on the needs: it depends on the system designs. In general, every OS has two parts:

- the **kernel**: it's the core of the OS, which always runs on the background and is responsible for all the interactions between the applications and the hardware;
- the **system applications**: every other application which is not part of the OS, but that can be installed in a second moment.

The OS handles many tasks: it is a **resource manager**, since it is responsible for handling and sharing the hardware resources such as the memory and the CPU, with the goal of achieving **efficiency**; it is also responsible for virtualizing any physical resources (an example is the virtual memory, the `VRAM`); it also is responsible for the **HW/SW interface**, usually through a set of common services (also called **API**s), but also to allow the users or the applications to interact with the hardware without a direct interaction.

# 1.1 Computer Organization

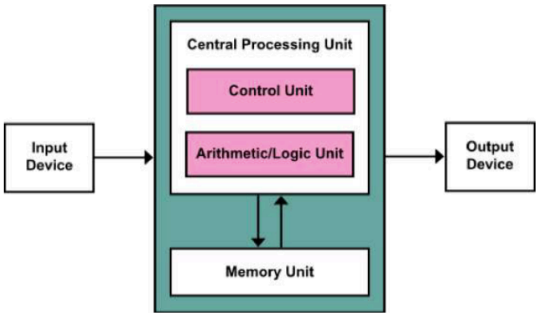
All the components of a computer (such as the CPU, the memory or the HDDs) are connected between each other via **system busses**. Let's have a more specific look at the various components:

- the **CPU**: is the unit in the computer responsible for the **execution** of the **various operations**. As of today, multi-core CPUs are the standard in each computer, but for the sake of simplicity, in this course we will always consider a single-core CPU;
- the **main memory**, which is responsible for the storing of data and instruction that will (or were) used by a CPU, and they are shared by all the **I/O** devices;
- the **I/O devices**, such as USB devices or video devices, which are each associated with a specific controller.

Conceptually, the same architectural model is used in most of the computing devices used in the world. The most used architecture is the **John von Neumann architecture**, which is based on a **stored-program** concept, as opposed to the fixed-program ideology. The idea of Neumann was to have a general purpose machine, which could perform any task as long as there was a program capable of being injected into the machine and executed.

The Neumann architecture employs 4 main pieces:

- a **CPU** with a **Control Unit** and an **Arithmetic Logic Unit** (**ALU**);
- a **memory** with the data and the instructions used by the CPU;
- some **input** and **output devices**.




## 1.1.1 Central Processing Unit

A **CPU** executes cyclically 3 steps: **Fetch**, **Decode** and **Execute**.

In the **Fetch** step, the CPU retrieves an instruction from a specific point into the memory; the point is memory is stored as an address into the **Program Counter** (**PC**).

In the **Decode** step, the instruction gets interpreted, such that the CPU can understand the requirements for the operation.

In the **Execute** step, the instruction gets executed.

The instructions are stored in **machine language**, which defines a set of elementary instructions that the CPU can execute directly. Some examples are the `add` or the `branch` instructions. More in depth, the machine language is represented by binary numbers, which is the most low-level that we can get. Instructions are stored in **words**, which are

groups of bits. Nowadays, words range from 32 to 64 bits.

The collection of instructions defined by the machine language is called **instruction set**. Each machine language instruction is made of two parts:

- an **opcode** (**op**erator **code**);
- zero or multiple **operators**, which define for instance some internal registers or some memory addresses.

There are multiple types of machine languages, such as **ARM**, **x86**, **MIPS**, **RISC**, etc... On this course, we'll focus on **x86**.

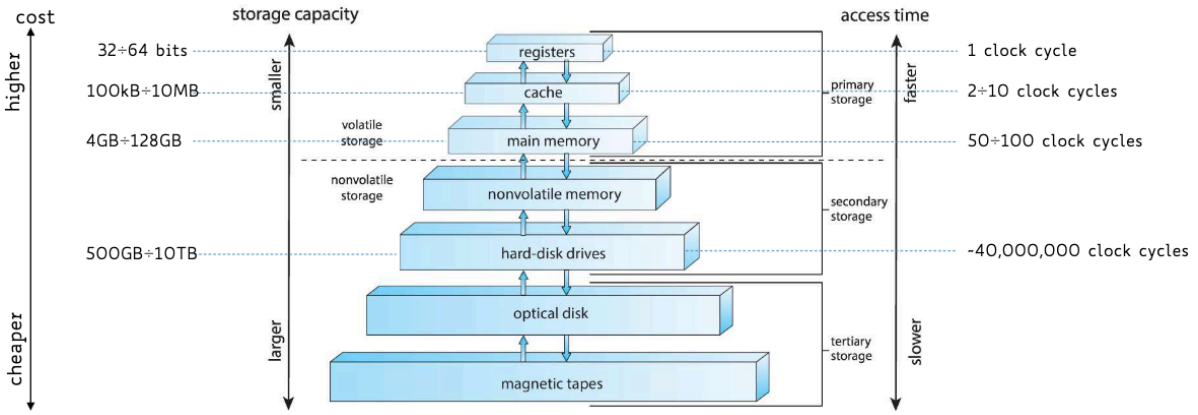
Registers are a sort of on-chip storage for words, and they usually match the CPU word size. There are multipurpose registers that can be used, such as `eax`, `ebx`, `ecx`, etc..., while other registers are special-purpose registers, such as:

- `esp`, which contains the **stack pointer** for the top address of the stack;
- `ebp`, which contains the **stack base pointer**, which points to the current stack frame;
- `eip`, which contains the **instruction pointer**, which holds the program counter.

Systems, as we said earlier, can be split into two major groups: **single-processor systems** and **multi-processors systems**. In a single-processor system, there is only one main CPU which executes the programs, while the other CPUs execute other tasks. In **multi-processor systems**, multiple CPUs are used to increase throughput (which doesn't increase linearly). Such systems have a higher resiliency to failures though.

## 1.1.2 Memory

When talking about memory, there is a hierarchy. Usually when talking about memory we refer to the **RAM**, but there are multiple types.




On top of the hierarchy, we have the **registers**: they are the fastest memory available, and they take only **one clock cycle** to retrieve the data, but they are really expensive. The main memory (**RAM**) takes around 50/100 clock cycles, and they cost less than registers. **Hardisks** take way more clock cycles, but they cost less and come in bigger sizes than, for instance, the RAM.

**Registers** and **caches** are usually **managed by the architecture** (even if in some cases caches may be managed by the OS), while **all the other** memories are managed **by the OS**.

Memory is usually represented and organized as a sequence of **cells**, where each cell represents a **group of 8 bits** or a multiple of it. Each cell has its own **address**, and in order to read or write to the memory, an address is needed.

### 1.1.3 Computer Busses

Busses connect all the components of a computer. Initially, there was one single bus which would connect all the components. Such bus was called **system bus**, which combined the functions of multiple busses:

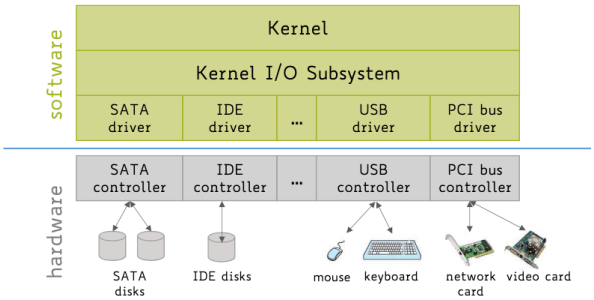
- the **data bus**, which carries information;
- the **address bus**, which determined where the information should be sent to;
- the **control bus**, which indicated the operation that should be performed.

Later on multiple busses were added for multiple purposes, generically to manage the CPU-to-memory and the I/O traffic. Some examples are the `PCIe` bus, the `SATA` bus, the `USB` device, ect...

### 1.1.4 I/O Devices

Each I/O device is made of two parts: the **device** itself and its **controller**, which is a chip or set of chips. There are multiple categories of I/O devices, such as the storage or user-interface devices. When the computer has to deal with a specific device, the OS has to use a specific **device driver**.

For each controller there is a specific driver, which allows the kernel to communicate with the device. The specific part of the kernel which communicates with the I/O is generically called **kernel I/O subsystem**.




Every device controller has some dedicated registers, which allow for the communication between the computer and the device itself:

- **Status registers**: they provide some status information to the CPU, such as if the device is ready, if it's idling, if it's used, etc...
- **Configuration/Control registers**: it stores the configurations for a specific device, which are used by the CPU to properly understand how to use and control the device;
- **Data registers**: they are used to send/store data from/to the I/O device.

But how does the CPU **communicate** with the devices? If the address bus and the data bus are shared, how can the single parts understand when they are called and when not? Let's start from scratch: when the **CPU asks** for a specific piece of **data** in the memory, it sends the **address through the address bus** and a `READ` signal on the **control bus**; the memory will eventually send the **data through** the **data bus**. If the CPU wants to retrieve/store data on/from an **I/O device**, a **special flag signal** (called `M/#IO`) is **raised**, which is important to ensure that the data will come from the I/O device and not from the memory.

CPU can talk to a device controller in two ways:

- through a **Port-mapped I/O**, the CPU references to the controller's registers by using a **separate I/O address space**, which is not in the main memory. Back in the old days, this method was used because there was less memory, and it was important to not waste it. For this method, the **I/O device controller's register** gets **mapped** to a specific port at **boot time**. The CPU uses the `IN/OUT` instructions to interact with the I/O device. When such instructions are used, the `M/#IO` signal is **asserted**, such that the computer knows that the instruction is not for the memory, but rather for the I/O device;

- through a **Memory-mapped I/O** instead, the controller's registers are **mapped** to the **same address space** used **for** the **main memory**. The problem with such method is that main memory gets occupied. Nowadays it's not a problem, since RAM is usually much more spacious than before, and that's why back in the past a port-mapped I/O system was used. With such method, **no** `IN/OUT` **instructions** are **needed**, since the I/O devices are, for the CPU, equal to some memory addresses. The addresses of the I/O device controller's registers are mapped at boot time as well. Whenever the CPU uses the I/O device's addresses, it also asserts the `M/#IO` signal.

How do we perform operation on the I/O devices though? There are multiple ways of doing it:

- **Polling**: the CPU constantly checks for the status of the I/O task. This is not convenient, since it creates continuous noise inside the busses;

- **Interrupt-driven**: the CPU receives an **interrupt signal** whenever the controller has **done** or **finished** the **I/O task**. Initially, the CPU starts the process, and while the I/O device works on the task, it executes other instructions. On each clock cycle, the CPU hears for any interrupt signal from the computer, and if there is one, it transfers the control of the currently executed task to the **interrupt handler**, so that the CPU can handle the previously started I/O process. Once the I/O process gets executed, the CPU returns to the interrupted task;

- **Programmed I/O**: the CPU does the actual work of moving the data from the memory to the device or vice versa;

- **Direct Memory Access** (**DMA**): the CPU delegates the task to a dedicated DMA controller. This way, we overcome the limitation of the Programmed I/O way. It's often useful to use the DMA when transferring large quantities of data, for instance from a disk to the memory. Such method works in this way: the device driver is asked to move some data from the device itself to the memory; the device driver

will tell that $n$ bytes must be transferred. The DMA (which is a specific chip) will move byte by byte from the device to the main memory through the CPU memory bus, and will decrease $n$ for each byte transferred. Once it finishes transferring (so when $n = 0$), then it will send an interrupt signal to the CPU.

The first two ways (**polling** and **interrupt-driven**) describe the way the task is managed, while the last two (**programmed I/O** and **Direct Memory Access**) describe who does the task. Usually, tasks are handled in an **Interrupt-driven** and **Direct Memory Access** way.

Why do we emphasize so much on I/O? Because in a computer multiple programs run at once and get stored in the main memory, and maybe in between we have to do some I/O tasks, and some computations must be suspended. Modern computers can keep all the programs in the main memory, doing what we call **multi-programming**.

## 1.2 Multi-programming Systems

In the 60s, as computers got more popular, instead of having a technician that loaded each program into the computer (as it previously happened with the punch card programs), the OS "became" the technician, in some sense. The OS would keep several jobs loaded into the main memory, and the CPU would jump from job to job. The OS was responsible of **scheduling the jobs**, **protecting** the **main memory** from the various accesses and **schedule** (and **perform**) the **I/O operations**. There was a problem though: I/O operations were considered as **blocking operations**, which would block the CPU and leave it idle while the I/O operation was carried out.

There is a solution to the blocking problem: the OS can make a schedule of all the jobs, and then, whenever a process needs to make an I/O operation, the CPU would know what process it should work on while the I/O operation is being worked out.

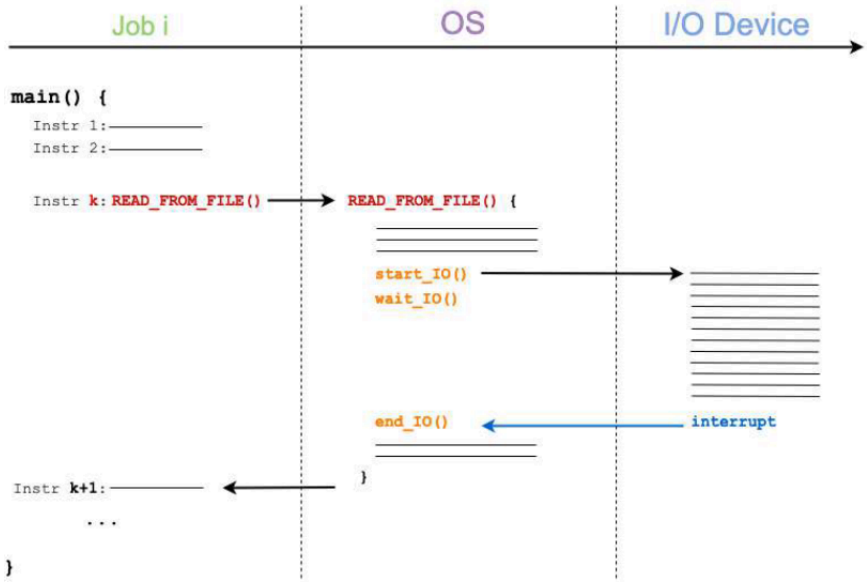



**Figure 1.1:** An example of blocking I/O operation

Let us consider the following example: we have a job $i$ which needs at a certain point to read something from a file. This calls an I/O operation, which would pass the operation

to the OS, which would then call the I/O device to perform the I/O operation and, once the I/O operation is completed, the result of the operation is returned to the OS, which gets returned in turn to the CPU so that it can finish job $i$. But in that case, while the I/O operation gets performed, the CPU is in idle.

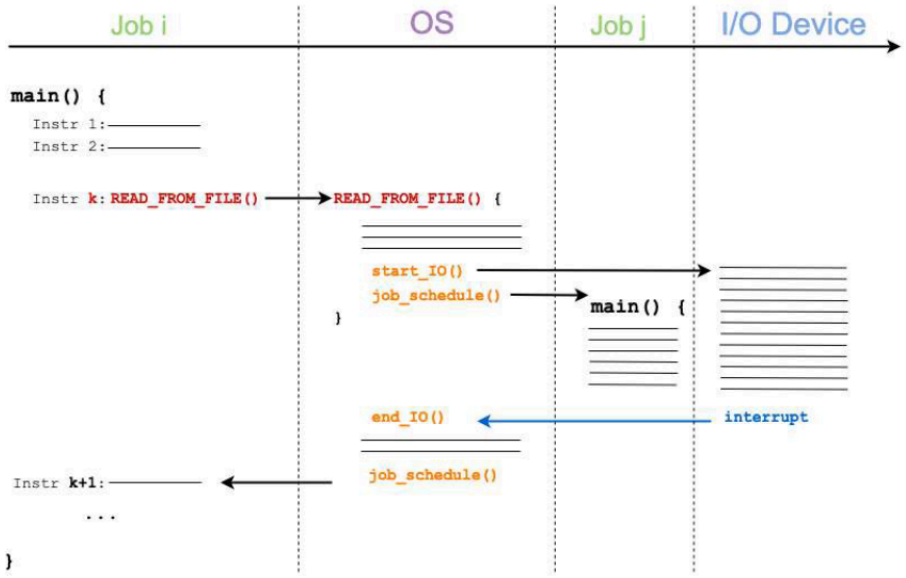



**Figure 1.2:** An example of non-blocking I/O operation with a second job $j$

A solution can be, as we said earlier, to call the CPU for starting another job $j$ in the meanwhile, so that the CPU won't stay idle. This, even if keeps the CPU always occupied, uses a lot of resources, because we always have to save the state of the job asking for an I/O operation, and it would take time to restore the job to its previous state.

It could happen for instance that a job would occupy the CPU for an enormous amount of time. Before, any job could use the CPU for any time it would want, but that would just block the whole computer on one single assignment. We call such job a **CPU bound job**.

A solution was found to this problem: in order to let the CPU complete as much tasks as possible, an artificial timer would be set, such that after each amount of time an **interrupt** signal would be sent to the CPU, such that it would pass from a job $i$ to a job $j$, and this would repeat in a continuous cycle. This way, there was the illusion of having a sort of parallelism, which is called **pseudo-parallelism**. It pretty much works as in the following image:

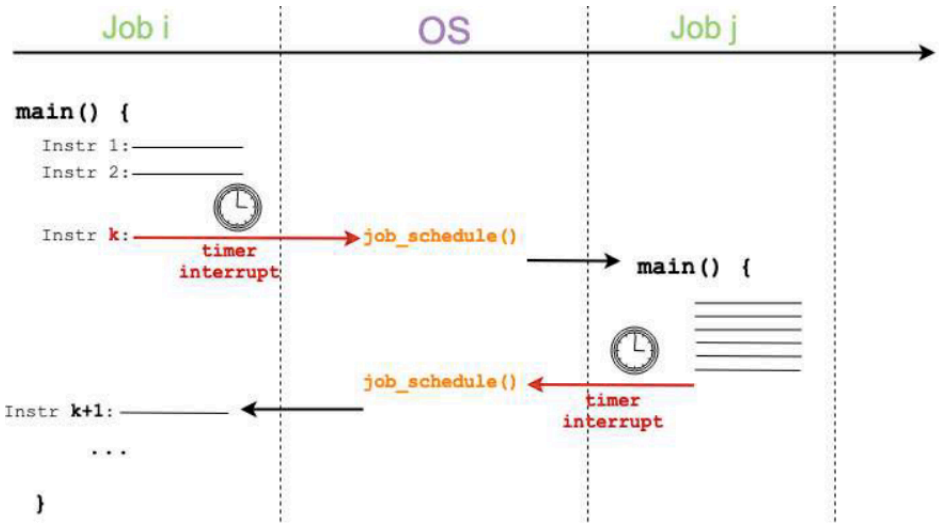



**Figure 1.3:** An example of pseudo-parallelism, where a timer sends an `interrupt` signal every $t$ time

## 1.3 OS Functionalities and Services

While designing an OS, we also must think about its services and functionalities. Some functionalities are basic, and are enabled by some architectural features. The capability of an OS is dictated by the architecture of the computer though: it may be helpful or it may be a disadvantage, but the relation between the two is really deep.

Some basic functionalities are for instance the kernel and a set of protected instructions. There is also a set of sensitive instructions, such as the `MOV`, the `HLT` (Halt instruction, it halts and shuts down the system) and the `INT X` (Interrupt instruction, generates an interrupt `X` signal) instructions. The `HTL` and `INT X` instructions can **not** be run from the user, but only from the OS.

The idea, while designing an OS, is that the OS should be the only one allowed to use sensitive instructions.

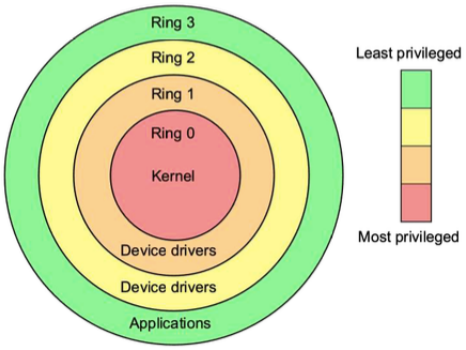
While executing CPU instructions, the computer might be in 2 states:

- **kernel mode**: it's the unrestricted mode, where the OS can perform any instruction without restrictions;
- **user mode**: it's a more restricted mode, where the user is not able to interact directly with the **I/O devices**, to **manipulate** the **content of the main memory**, **switch to kernel mode**, and other sensitive features...

Such state is stored in the hardware as a bit, that could be either 0 or 1. System calls (**SysCalls**), which in `RISC-V` were invoked via the `ecall` instruction, have the function to delegate some code to the OS, and "switch" from the user to the kernel mode.

The hardware should support **at least** the kernel and user mode, but during time this distinction between kernel and user mode became more fine-grained. It's common now to have **4 protection rings**: each ring is a different layer, and the lower the ring is, the more unrestricted access it has. We can still save this distinction separately in the hardware with the use of 2 bits.

The architecture of the hardware should also allow the OS to protect each program to the others: there shouldn't be resources invasions, one program should not go in the memory reserved for another program. This means that there has to be some protection.




The simplest way is to have 2 dedicated registers:

- **base**: it contains the starting valid memory address;
- **limit**: it contains the last valid memory address.

Upon program startup, the OS loads into these registers the values of the starting and last memory address, and whenever the program wants to access the main memory, there is a check done by the CPU so that the programs don't access the main memory where they are not allowed.

System calls are used from user programs that allow them to perform some special actions which would usually be performed only by the OS in kernel mode. System calls basically make a request to the OS so that it may perform some actions on behalf of the program. They are the only way to cross the boundary of user and kernel mode, without instantiating a **trap**.

**Definition:** Trap

A **trap** is any event that causes the switch to the OS kernel mode

There are 3 types of **trap**:

- **system call**: also called **software-trap**, it's a **synchronous**, **software initiated** request for an OS service (for example `ecall` on `RISC-V`, or `SYSCALL` on `x86`);
- **exception**: also called **fault**, exceptions are **synchronous**, **software initiated** responses to an exceptional event (such as a division like %, or a segmentation fault);
- **interrupts**: it's an **asynchronous**, **hardware initiated** calls. Some examples might be for instance that a network packet arrived, the completion of a timer, etc...

## 1.4 System Calls

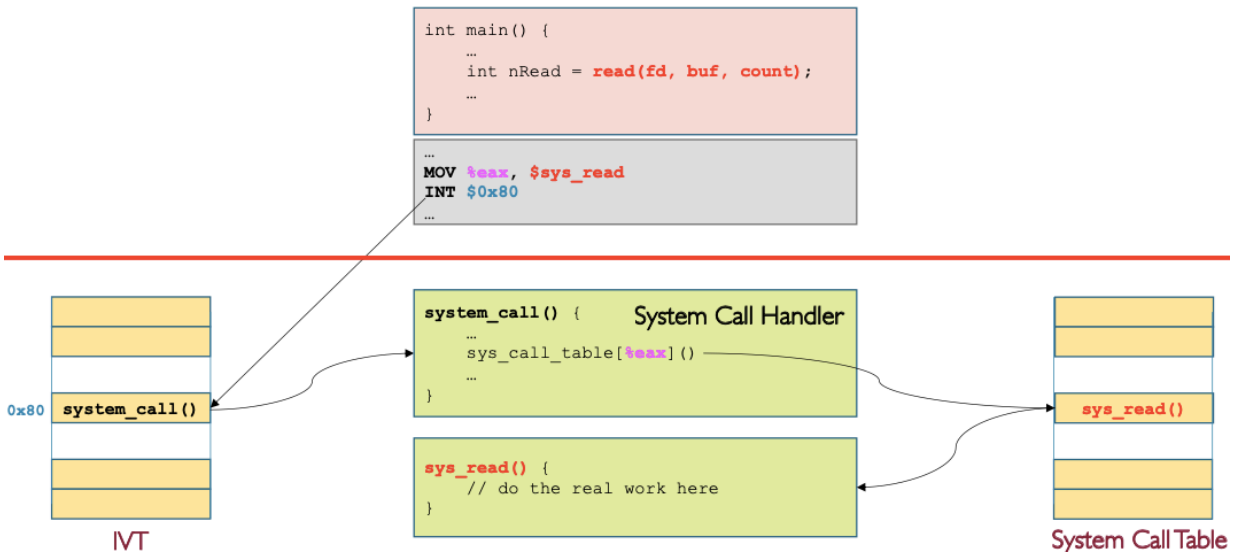
We talked about system calls in these pages, and we said that they are the only way for a program to access to some restricted instructions. More in general, system calls are **OS procedures** that execute **privileged instructions**, and are a programming interface to the services provided by the OS. System calls are typically written in an high-level language such as `C` or `C++`, and are mostly accessed by programs via a high **A**pplication **P**rogramming **I**nterface (**API**). An example is the GNU C Library, used by Linux systems, Unix systems and macOs.

There are 5 categories of system calls: **process control**, **file management**, **device management**, **information maintenance** and **communications**.

### 1.4.1 Anatomy of a System Call

A common system call that is frequently used by the OS is the `read` call. Let's see an example of how it works:

1) an application would call for the `read()` function because it wants to access to a file;
2) `C` would then compile the program and transform the call into machine code, where via an ID, would invoke the specific system call needed for reading;
3) the OS, in the kernel, has a data structure called **I**nterrupt **V**ector **T**able (**IVT**) which lists all the possible traps that might be executed. Depending on the value of the interrupt signal, the system call varies;
4) once the corresponding system call has been found, it gets handled by the **System Call handler**, which then redirects the system call to the right handler (so for instance, a read system call would need some particular handler, while an exception would need another handler);
5) all the handler are listed on a table called **S**ystem **C**all **T**able (**SCT**). Once the system call reaches its own handler, it then gets executed. After ending, it returns its result to the program.




We said earlier that every time that a trap is called, the CPU switches from user mode to kernel mode. The system call handler has the task of **saving** the **state** prior to the trap of the user programs on some dedicated registers, **finding** the **correct routine** of the trap and, when the task is completed, **return** to **user mode**.

### 1.4.2 Timer and Atomic instructions

Each system has a **timer**, which allows the CPU to schedule its jobs. It has the form of an incremental value, or more specifically, of a clock. On every $x$ time quantity, the timer emits an interrupt signal, so that a CPU won't be monopolised by one single job.

We don't want though sometimes to interrupt a process (for instance while updating a variable, or while reading something from the memory), maybe because it's important or may risk to corrupt the data: the OS should be able to organise and synchronise the various activities. Hardware should ensure that certain processes are executed **atomically**. That may happen either by disabling interrupts and re-enabling them after the task has been done, or by having some special instructions being natively executed automatically

## 1.5 Virtual Memory

Virtual memory is an abstraction of the physical main memory: it gives to the programs the illusion that the computer has unlimited resources, and that the memory is actually way more spacious that what it actually is. Such technique allows program to be run also if they are not entirely loaded in the main memory, even if they are actually loaded in the virtual memory.

This technique is implemented by the hardware with the **M**emory **M**anagement **U**nit (**MMU**), which is responsible for translating a virtual address into a physical one, and by the OS, which is responsible of the management of such memory. It uses a cache called **T**ranslation **L**ook-Aside **B**uffer (**TLB**) for quicker lookups regarding recent mappings. It's important for the OS to be aware of which pages are loaded in the memory and which ones are not.

How do virtual addresses work though? On a 64-bit system, the CPU is capable of addressing $2^{64}$ bytes (thus 16 exbibytes (EiB)): this means that the virtual address space has the range $[0, 2^{64} - 1]$, which is way more than the available space in the physical memory. The virtual address space is organized in **pages**, of usually 4 KiB each, which are stored and loaded on the disk, rather than in the main memory.

## 1.6 OS Design

Designing an OS is quite a challenge: the internal structure is vast, depending on what we want the OS to do. Designing an easy to use OS is quite different and does not always coincide with an easy to program OS. To which we should point, it's up to the programmer.

It's important to keep **policies** (what will be done) and **mechanisms** (how to do it) separate, since it improves the system's **flexibility**, so that the addition, the modification and the removal of policies can be easily done, **reusability**, so the possibility to use certain mechanisms multiple times, and **stability**, so that while adding a new policy the system doesn't destabilize.

Back in the days, OSs were implemented with low level machine language, which was highly **efficient**, since it targeted a specific machine and took full advantage of it, but also a problem, because it wasn't easy to use it on other machines and to port it (**reduced**

**portability** in general). Nowadays OSs are made of multiple languages, such as `ASM`, `C`, `C++`, `Perl`, `Python`, `Rust`, etc...

OSs, in order to keep modularity, should be divided into multiple **subsystems**, where each subsystem handles certain functionalities. Depending on the OS, the structure varies.

## 1.6.1 MS-DOS and Unix

**MS-DOS** was made by the Microsoft corporation, and it's the predecessor of the modern **Windows**. The OS was very compact: there were **no modular subsystems**, and most importantly, **no separation** between **user** and **kernel mode**. The good side in this structure is that it's easy to implement, while the main problem was that there were multiple security issues.

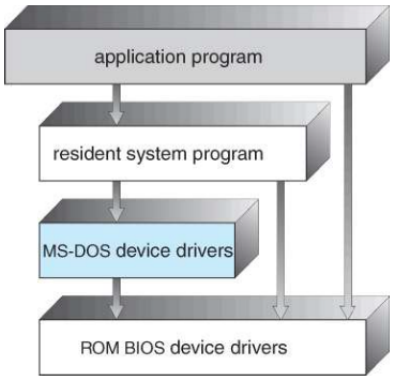



**Figure 1.4:** MS-DOS OS structure

**Unix** instead is different. Started in the 1960s, it has a traditional monolithic kernel. Essentially, it was a huge process, **running all at once**, with all the subsystems active. Most of the modern OSs use a variant of this OS. The good side of it is that it's really **efficient**, and also **easy to implement**, although it was rigid, and **not that much secure**, because of the continuous communications between the subsystems.

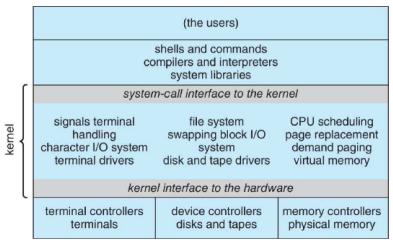



**Figure 1.5:** Unix OS structure

## 1.6.2 Monolithic OSs and Microkernel

OSs are usually divided into multiple $N$ **layers**, where each layer uses the functionalities of the layer underneath $(N-1)$ to give new functionalities to the upper layer $(N+1)$. The good side of a layered structure is that it's **modular**, **portable** and **easy to debug**, while the downsides are that there are **extra copies** of some functionalities and there is a weak communication between the layers.

The concept of **microkernel** structure is opposed to the monolithic structure: the kernel is made of only some basic functionalities, where everything else is executed in user mode. This way, the separation of policy (user mode) and mechanism (microkernel) is ensured. This method ensures **security**, **reliability** and **extendibility**, while **message passing** is slowed by the multiple components.

Many modern OSs use **L**oadable **K**ernel **M**odules (**LKM**), which have an object-oriented approach: each module's core is separate, and communicates with the other cores through known interfaces. Each module is loadable and needed within the kernel. It's very similar to the layer structure, but it more flexible, since it allows for major customization.

Each system chooses different architectures, depending on the needs. While designing

an OS, one wants to try to get the best out of both the monolithic and microkernel approaches. For instance, Linux and Solaris use a monolithic OS with the addition of LKMs (this combination is often called modular monolithic), Windows NT uses a mostly monolithic OS with the addition of some microkernel features for different subsystems, while macOs uses a combination of monolithic OS (coming from Unix), microkernel (from Mach) and LKMs.

# Chapter 2
## Process Management

Prior to now, there wasn't a specific distinction between **process** or **program**. Let's see the distinction between the two:

**Definition:** Program

A **program** is an executable file, which resides on the persistent memory.

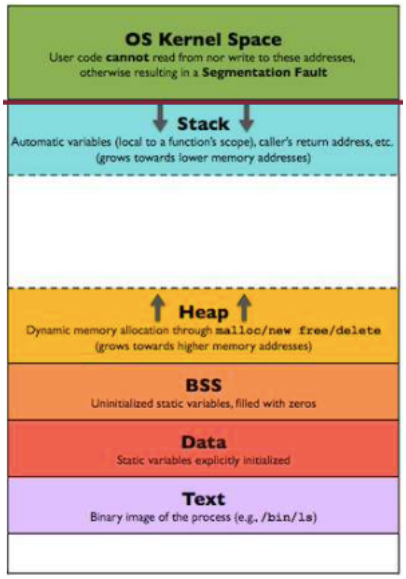
An example of program could be an executable such as Firefox, or for instance all the command line tools such as `ls`, `cd`, `chown`, and so on... Such programs are usually stored on the disk on a location like `bin/<programname>`.

**Definition:** Process

A **process** is the OS abstraction of a running program.

A process is **dynamic**, while a program is **static** (it's code and data only). Several process may concur to run the same program, but each process has its own state. An example of process might be when there are multiple shells running simultaneously the `ls` command. Processes execute sequentially, one at a time.

The OS manages how much virtual address space is given to each process. On the right there is an image representing what an Unix 32-bit system looks like. Usually, Unix systems reserve in the memory the last addresses (in this case it would be from address `0xC0000000` to address `0xFFFFFFFF`, which amounts to around 1GB). The virtual address space is just an abstraction of the physical memory address space, and each machine generates its own valid virtual addresses (the virtual memory space is machine-dependant): for instance, on a 32-bit system, we would have a range of possible addresses in the range $[0, 2^{32} - 1]$ (depending on the OS some space might be reserved to the OS kernel).




Let's see more in detail what each section of the memory does:

- **text**: it contains executable instructions;
- **data**: it contains global and static variables, which are initialized at boot time;

- **BSS**: it contains variables uninitialized or initialized to 0;
- **stack**: it contains temporary variables that are needed by the called functions, and it's structured in a LIFO order;
- **heap**: it is used to contain variables whose size is not determined at compile time (for instance the `String` in Rust is stored in the heap, since it's dynamic, while `&str` is stored in the stack, since it's static).

## 2.1 Stack

The stack is a place in memory which allows to store the temporary variables made by the functions. It has a **LIFO** structure (**L**ast **I**n, **F**irst **O**ut) and allows for 2 operations: **push** (to insert elements into the stack) and **pop** (to delete elements from the stack). A dedicated register (usually `esp` on `x86` and `sp` on `RISC-V`) stores the address of the top of the stack (we use %esp to indicate the content of `esp`, or %sp if we want to indicate the content of `sp`). The register will always store the top of the stack relative to the process that is running. Conventionally, the stack memory grows from top to bottom.

Each **function** uses a portion of the stack; such portion is called **stack frame**. At every point in time, we will have multiple stack frames existing simultaneously, yet only one at a time will be active. Each stack frame has 3 parts: **function parameters** and the **return address**, which are instantiated by the **caller** of the function; **back-pointer** to the **previous stack frame**, which is instantiated by the **callee**; **local variables**, also instantiated by the **callee**.

Whenever we add an item to the stack, it grows towards down, and %esp decrements by 4 bits (in 32-bit systems); once %esp decremented, the item gets copied into the stack to the specified address and the `call` instruction (which calls the function, it's similar to the `jal` instruction in `RISC-V`) implicitly pushes the return address of the stack. There is a problem though: the stack may grow if for instance the function needs to store multiple variables, and if the stack pointer doesn't have a fixed location, then retrieving the data from the stack may be hard.

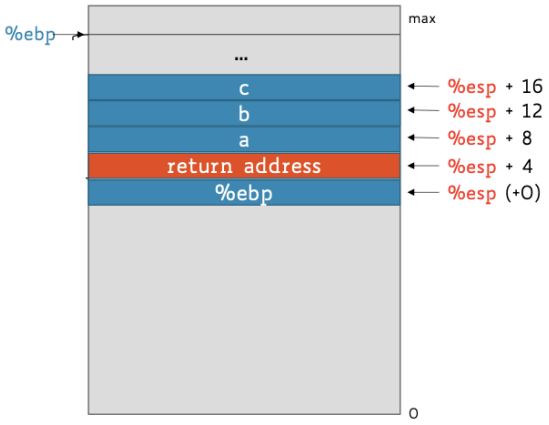



**Figure 2.1:** A picture of a complete stack, with %esp and %ebp

A possible solution is to use another register, `ebp`, to mark the beginning of the stack frame. `ebp` will stay fixed, while instead `esp` is free to change and enlarge how much it wants. After %ebp is fixed, it gets saved in the stack frame, so that the function can access it whenever it needs it.

The stack base is saved in this way: first, the **callee** saves the stack base pointer %ebp into the stack **after** the **return address** of the function: at that moment, the place where `esp` points is where %ebp will be stored. Usually, %ebp points to the base of the

stack. After having stored where the base of the stack lies, we then set `ebp` to the address of where the old `%ebp` lies. This way, we can use `ebp` to access the function's parameters by making `%ebp grow` and we can use `%esp` to access to the local variables by making the value decrease. When we don't need anymore that stack frame, we just repeat these steps backwards.

## 2.2 Process Execution States

A process has 5 possible states:

1) **New**: the OS just started the process;
2) **Ready**: the process is ready to be processed, and it's scheduled to go into the CPU;
3) **Running**: the process is currently being executed by the CPU;
4) **Waiting**: the process is suspended because it's waiting for a resource to be available or an event to happen/occur;
5) **Terminated**: the process is completed and the OS can destroy it.

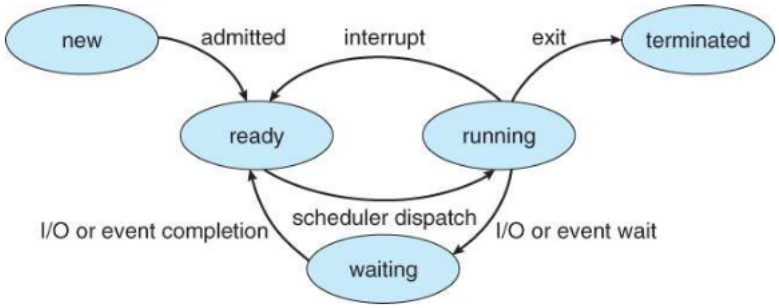



**Figure 2.2:** The state transition diagram of all the possible processes' states

The various processes go from one state to the other depending on the actions taken by the OS: for instance, if there is a process in the **running** state and an `interrupt` signals gets propagated (such as the `timer`), then the process goes in the **ready** state, and waits for the OS to schedule it again.

A lot of system calls are blocking, and while a blocking system calls is executed, the user can't do anything. The OS, when a system call is called, puts the current running process in a **waiting** state, and schedules a different process to make sure that the CPU is never idle. When the system call gets finally computed, the previously waiting process goes back to the **running** state. The system calls don't block the whole system, **they just block** the **running process**.

Each process state consists of the following information: the **code** of the program; the **static data** of the program; the **program counter** of the program; the **stack** and **heap** addresses of the program; the **resources used**; the **execution state**.

The progress of each process is stored by the OS in the kernel memory in a table called **P**rocess **C**ontrol **B**lock (**PCB**), and it allocates a new table for each process. Each process has a unique identifier, which is used to differentiate each table. Each table has at least the following attributes:

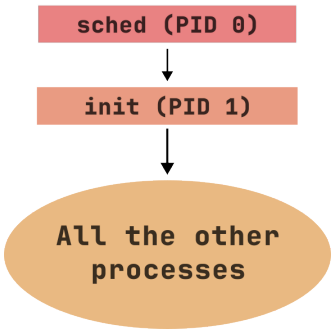
- the **process state**;
- the **process number** or **process identified** (**PID**);
- the **program counter**, the **stack pointer** and the related **general purpose registers**;
- the CPU scheduling information, such as the priority of the process;
- the memory related information, such as the page tables related to such process;
- the accounting information

## 2.3 Process Creation

When the OS gets started, some processes are started automatically. But for instance, when the OS is ready, then the user might want to use some applications and start thus some processes. Since creating a process is handled by the OS, then **in order to create a process** we **must use** a **system call**.

When a process gets created, such process may want to start some other processes. The creator process is called **parent**, while all the processes started by the parent are called **children**. A parent process can either wait for a child to complete (if parent and child are not related and do not require each other to complete) or run in parallel with the child. The parent process can share resources and privileges to its children. Each process has an identifier, called **p**rocess **id**entified (**PID**), and a **p**arent **p**rocess **ID** (**PPID**), which identifies the parent process of a given process.

In Unix systems, the process scheduler is named **sched**, and has `PID 0`. At startup, the **init** process is started, and has `PID 1`. The role of **init** is to start all the system daemons and user logins, and is the ultimate parent of all processes. Other processes are created through the use of the `fork()` system call.




Upon the creation of a process, we have two ways for creating it:

- the **UNIX way**: we duplicate the parent process, and then it modifies the process' code with the wanted code. This way, we use two system calls: one to duplicate the process (which is `fork()`) and one to edit and execute the code (called `exec()`). When the process gets duplicated, the child process shares **program** and **data** memory with the parent process, but it will have its own `PCB`, `PID`, `PC` and registers. After the program gets loaded into the new process by using the `exec()` system call, then the process receives a new **code** and **data segment**;
- the **Windows way**: it uses the `spawn()` system call and creates from scratch a process, and then the OS proceeds to load the new code.

In UNIX, after creating a child, there are two options for the parent after creating a child:

1) to **wait** for the child to terminate before proceeding: the parent uses the `wait()` system call, and waits for the child to finish before issuing a new prompt;

2) to run concurrently with the child, continuing to process without waiting. This happens for instance when in the UNIX shell we use the `&` symbol after the command: the process will run concurrently in the background.

When a parent process runs the `fork()` system call, then such call creates a copy of the parent process which will then become the child process, and a specific `PID` gets created for the child process. After the system call gets executed, both the parent and the child process continue from that same line where the `fork()` system call was called, and this happens because after the system call the parent and the child still share the same memory for data and program. The system call will return a different code depending on who executes it:

- if the call is made by the **parent**, then the `fork()` system call returns the **child's PID**;
- if the call is made by the **child**, then the `fork()` system call returns `0`;
- if the call wasn't able to produce a child process, it will return `-1`.

Here follows an example of a Rust code that would make a child process and then, depending on the result of the `fork()` system call, it will print the `PID` and `PPID` of the parent and the child process.

```rust
fn main() {
    unsafe {
        println!(" →  PID of this process: {}", process::id());
        println!(" → PPID of this process: {}\n", getppid());

        let new_proc: Result<ForkResult, Errno> = fork();

        match new_proc.expect( msg: "Fork Failed: Unable to create child process!") {
            Child ⇒ {
                println!(" · Value of the fork() system call: {:?}", &new_proc);
                println!(" →  PID of the child process: {}\n → PPID of the child process: {}\n",
                    getpid(), getppid()
                )
            },
            Parent { child :Pid } ⇒ {
                wait().expect( msg: "Error: couldn't wait");
                println!(" · Value of the fork() system call: {:?}", &new_proc);
                println!(
                    " →  PID of the parent process: {}\n → PPID of the parent process: {}\n",
                    getpid(), getppid()
                );
            }
        }
    }
}
```

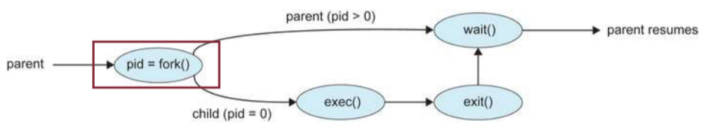



**Figure 2.3:** Code of a `fork()` system call and its visual representation

Let's analyze what happens in this code snippet:

1) first, we retrieve the `PID` and `PPID` of the parent process;

2) then, we execute the `fork()` system call, and we store its result inside the `new_proc` variable;

3) we then match the result of the `fork()` system call: Rust encodes directly the integer code of the `fork()` system call into a data type. If the data type is:

    - an `error`, then it means that `fork()` returned `-1` and the forking process failed;
    - `Child`, then it means that the result of `fork()` was `0`, which means that such part of the code is getting executed by the **child** process. It thus proceeds to print the value of `fork()`, the `PID` and the `PPID` of the newly created child process;

- `Parent`, then it means that the **parent** is executing that part of the code, and thus the result of `fork()` is the `PID` of the child. The parent then waits for the child process to complete and, when the child process executed its code, the parent process proceeds to execute its own code. It then prints the value of `fork()`, the `PID` and the `PPID` of the parent process.

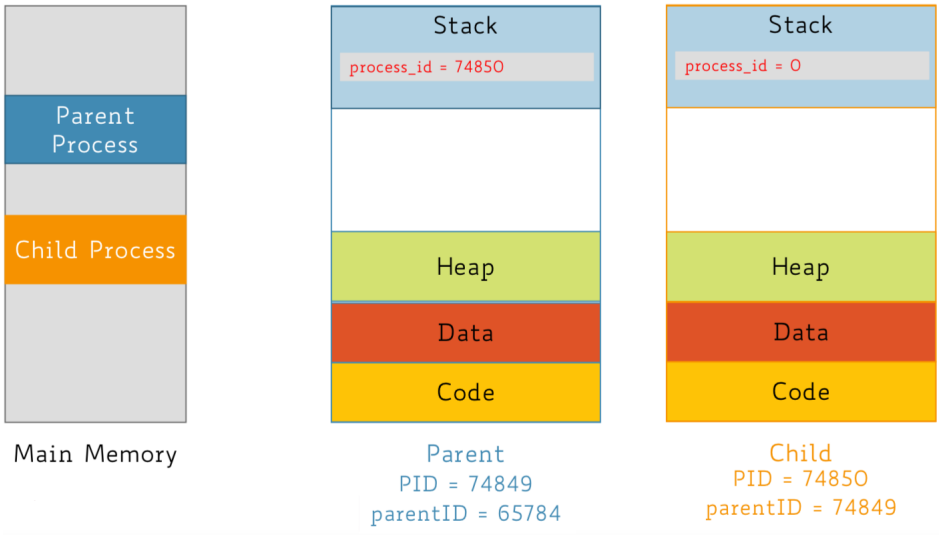
Now, if we run this program, we'll obtain the following results:
the first printed results are the `PID` and `PPID` of the parent process, and then the fork happens: we create a child process and we let the parent wait. The child executes then its code: it prints the value of the `fork()` result relative to itself, then it prints its `PID` and its `PPID`, which we notice being equal to the `PID` of the parent printed at the beginning.

```
./processesRS
→  PID of this process: 25199
→ PPID of this process: 12988

· Value of the fork() system call: Ok(Child)
→  PID of the child process: 25200
→ PPID of the child process: 25199

· Value of the fork() system call: Ok(Parent { child: Pid(25200) })
→  PID of the parent process: 25199
→ PPID of the parent process: 12988
```

**Figure 2.4:** The executed code in **figure 2.3** and its outputs

After the child, the parent runs: we notice that the `PID` and the `PPID` are the same of the ones printed on top. But who's the parent of the parent process? It's actually the shell, since it invoked the `processesRS` program, and thus being the parent's parent.

Here below is shown a figure that highlights how a process is created and stored in the memory: first, there is the parent process. When `fork()` gets called, then the child process is copied and stored inside the memory. Then on the stack we have the result of `fork()`, stored as `process_id`: on the parent is the `PID` of the child, while on the child is `0`, meaning that the fork was successful.




But what happens if we let the parent execute first? What will happen? As we can see from the image on the right, the parent executes and then the terminal closes the prompt, waiting for the next one. After it though, the child executes. This happens because the two processes ran concurrently,

```
./forkWithSleep
→  PID of this process: 36080
→ PPID of this process: 33199

· Value of the fork() system call: 36081
→  PID of the parent process: 36080
→ PPID of the parent process: 33199

· Value of the fork() system call: 0
```

not one after the other, and also because the parent process didn't wait for the child process.

Now, the `fork()` system call is not enough alone to start a new process, in fact, it just creates an identical process to the one that executes the system call. This is also the reason why the child process in the previous example was identical to the parent process: they had the same code, and they executed from the same point onwards. If we want to load the code of a new process, we have to use the `exec()` system call, which edits the code of the process and runs it.

Let us examine the code on the left:

1) first, we take the `PID` of the program and we print it. After it, we take an input from the terminal: such input will be the program that will be opened (it can also be seen as the process that will be started);

2) we then use the `fork()` system call: this creates an identical process to the one that is executing, and then there is a branch: if the child is the one running, then we load the new program with the `execlp()` system call, otherwise, if the parent is running, it proceeds to wait for the child. The system call for executing another program has the following form:

```
execlp(const char *file, const char *arg, ...)
```

   It takes as parameters the path to the executable and some arguments for the program. When we run this system call, it tries to locate the program. If the program can't be found, it then prints the statement below, otherwise it loads the program and executes it. After the execution of the program, the child process won't print the final statement, but why does it happen? Because it has a different code now, and it doesn't follow anymore the code of the parent;

3) in the meanwhile, the parent is running too, but it stops at the `waitpid()` system call. Such system call makes the parent wait until a child process specified by the `PID` given as parameter finishes executing. The structure of the system call is the following:

```
waitpid(pid_t pid, int *status, int options)
```

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

using namespace std;

int main() {

    int current_pid = getpid();
    cout << "Current process ID is: " << current_pid << endl;

    string progStr;
    cout << "Type the name of the executable program you want to run: ";
    // read the name of the program we want to start
    getline(cin, progStr);
    const char *prog = progStr.c_str();

    int pid = fork();

    if (pid == 0) { // child
        execlp(prog, prog, 0); // load the program
        // if prog can actually be started, we will never get to the
        // following statement, as the child process will be replaced by prog!
        printf("Can't load the program %s\n", prog);
    }
    else { // parent
        //sleep(1); // give some time to the child process to starting up
        waitpid(pid, 0, 0); // wait for child process to terminate
        printf("In the parent process: child process %s is finished!\n", prog);
    }
    return 0;
}
```

**Figure 2.6:** Code for using `fork()` and `exec()`

where `pid` is the `PID` for which the program has to wait and `status` is where the system call should save the information relative to the waiting (if different from `NULL`). After waiting for the child to finish, it then prints the final statement.

## 2.4 Process Scheduling

When scheduling processes, there are two objectives: one is to **keep** the **CPU always active**, so that no resources are wasted; the second is to **deliver acceptable response times** for all the programs. In order to achieve there two objectives, the CPU is able to swap between processes, even if this action goes slightly against the objectives: if we want to keep the CPU always active, then while swapping processes we waste some time, making the CPU "sleep" while changing the process data.

The OS maintains all the PCBs of all the processes in multiple **state queues**: one for each state. There are also different queues for each I/O device. Whenever the OS changes the status of some process, then the PCB of such process gets unlinked from a queue and moved to the appropriate one. Each state queue is not managed in the same way, it depends from OS to OS and from the hardware as well.

Some queues have limitations: for instance, if we have only one CPU with only one core, then we can have only one process running at a time. This means that the `Running` queue can have at most one process. The other queues are basically unbounded, they depend on the size of the memory, there is no limit to how many processes we can have on a machine.

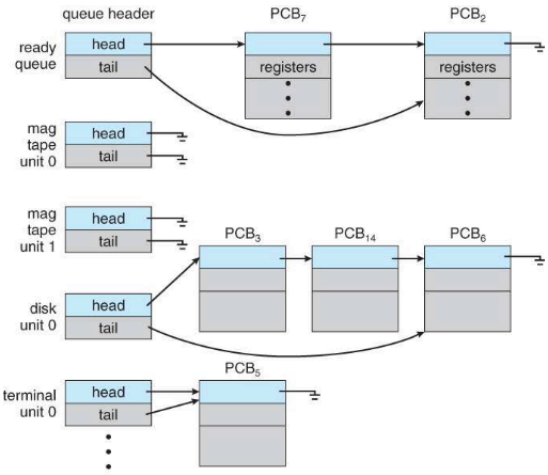



**Figure 2.7:** An example of Process State Queues

There are two types of **schedulers**:

- **long-time schedulers**: it runs infrequently and is typical of a batch system or a very heavily loaded system;
- **short-term schedulers**: it runs very frequently (usually around every 100 milliseconds) and must quickly swap one process from the CPU to the queue and vice versa;
- **medium-term schedulers**: when the system load becomes high, a medium-term scheduler is dispatched to swap the longer processes with the smaller ones, so that the system gets cleared easily.

An efficient scheduling system may select CPU-bound processes and I/O bound processes together, so that all the computer's parts are used simultaneously.

When a process gets swapped from the queue to the CPU or vice versa, we call such

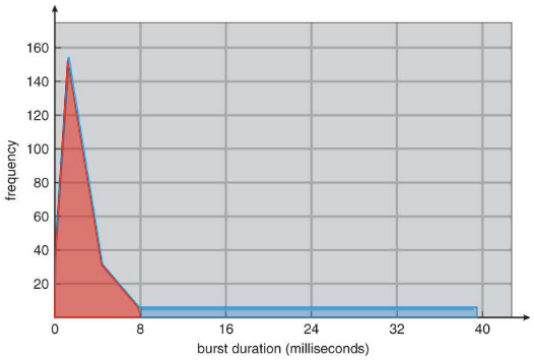
procedure **context switch**. This procedure is an highly costly operation because we have to save the state of all the registers into the PCB (such as the program counter, the stack pointers, and so on...) and load the states of the new process. A context switch happens usually when there is a **trap**: may it be a system call, an `interrupt` signal, an exception... Whenever a trap gets caught by the CPU, then it has to save the state of the current process, switch into kernel mode to handle the trap and then load back the data of the previously interrupted process.

Usually I/O processes get switched due to I/O requests, while CPU-bound processes could, theoretically, never issue an I/O request. In order to not block the CPU on working on only CPU-bound processes, a timer is set, which triggers a context switch. This timer is called **time quantum** or **slice**. We call **time slice** the maximum time between two context switches, and ensures that a context switch happens at least every, say for instance, 50ms. While theoretically a context switch may never happen, it practically happens much more frequently than we may expect. The time slice can be easily implemented in the hardware through a timer.

## 2.5 CPU Scheduling

We know that the CPU continuously switches from process execution to I/O operations, with the scope of being always busy and avoid moments where there would be no workload. While a process asks for an I/O operation, another process gets executed, in order to maximize the throughput. All processes reside in different queue tables, and each process resides in one and only one queue; this way, the OS can easily manage when to switch processes.

Processes alternate from two states in a continuous cycle: they go from **CPU burst** to **I/O burst** and vice versa. A **CPU burst** is a state where the CPU performs calculations, while **I/O burst** is a state where the system waits for the transfer of data. But not all processes have the same duration of CPU and I/O bursts. A particular study underlined how processes have an highly skewed distribution when it comes to how much they remain in the CPU burst state: most of the processes have short CPU bursts (usually around the 8ms), while only very few processes have long CPU bursts.




We differentiate between two types of scheduling: **long-term** and **short-term**. For **long-term scheduling** we mean the way the OS decides which programs have to be loaded in the main memory, thus the level of **multiprogramming**, while for **short-term scheduling** we mean the way the OS decides which process from the queue of the **ready** processes should execute first. The short-term scheduler is also called **CPU scheduler**

Whenever the CPU becomes idle, the scheduler picks another process from the `ready` queue in order to execute it. The queue does not necessarily have a **F**irst **I**n **F**irst **O**ut (**FIFO**) structure: in fact, not every process may be suitable to be executed when another

process gets moved in the `waiting` queue (we saw it from the study that we explained earlier). But when does the CPU scheduler act? There are 4 cases where the CPU scheduler acts:

1) when a process switches from the `running` to the `waiting` state (because of an I/O operation or a system call);
2) when a process switches from the `running` to the `ready` state (because of an `interrupt` signal);
3) when a process switches from the `waiting` to the `ready` state (after an I/O operation gets completed or after returning from a `wait()` system call);
4) when a process is created or is terminated.

There are two types of CPU scheduling: **non-preemptive** and **preemptive scheduling**:

- **non-preemptive scheduling** takes place when there is no choice (so either in the $1^{st}$ or $4^{th}$ condition). In this case, a process will continuously run until either it finishes of it voluntarily blocks itself;
- **preemptive scheduling** happens when there is the choice of either **continuing** with the current process or **select** a **new one**.

There are some **corruption** issues with preemptive scheduling though: some problems may occur when the kernel is busy **implementing** a **system call** (so for instance while updating some critical kernel data structure), but also, if two processes share data, then one process might be interrupted while **updating** the **shared data structure**.

There are some countermeasures that allow us to avoid such corruption issues: for instance, one is to make the **process wait** until the system call has either completed or blocked before allowing for preemption. This is a problem though for real-time systems, since real-time response wouldn't be guaranteed anymore. Another possible way to avoid such problem is to **disable interrupts** before entering critical code sections, and then reenable them immediately afterwards. This has to be done in very rare situations though, and only on very short pieces of code that have to finish quickly.

There is a module that gives to the currently running process the control of the CPU, and it's called **dispatcher**. This module is responsible for **switching context**, for **switching to user mode** and to **jump in the location of the newly loaded program**. The dispatcher acts on each **context switch**, and it consumes some time called **dispatch latency**.

There are some useful definitions regarding timing that we have to know:

- **arrival time**: is the time at which the process **arrived** in the `ready` queue;
- **completion time**: is the time at which the process **completes** its **execution**;
- **burst time**: is the time required by a process for **CPU execution**, and it doesn't include the I/O burst;
- **tournaround time**: is the time **difference** between **completion** and **arrival** time;

- **waiting time** (into the `ready` queue): is the time **difference** between **tournaround time** and **burst time**.

Remember that **I/O time is not considered** here in these definitions.

### 2.5.1 Scheduling criteria

There are multiple criteria that get considered when trying to select the best or most optimal scheduling algorithm, such as:

- the **CPU utilization**: so the percentage of time for which the CPU is always busy. Ideally, the CPU would be busy 100% of the time, but on a real system the optimal CPU usage should range from 40% to 90%. We should maximize it where possible;
- the **throughput**: the number of processes completed in a certain unit time. It should be maximized as well;
- the **tournaround time**: so the time needed for a process to complete, and should be minimized where possible;
- the **waiting time**: the time spent by the processes in the waiting queue, which is **different** from the waiting state: processes in the waiting state are not under the control of the CPU scheduler. It should be minimized where possible;
- the **response time**: so the time from when a prompt gets issued to when the process starts responding. We want to minimize it as well.

Ideally, a good algorithm would respect all of the previous metrics, but it's generally impossible, and some trade-offs must be made. We can though create some policies and let the algorithm stick to them as much as we can. While describing such policies, it's important to remember that some assumptions are being applied: first, there is **one process per user**; secondly, **each process** is **independent** from each other; lastly, there is **one thread of execution for each process** (we'll talk about threads later on).

Some policies that we might apply are for instance the **minimization** of the **average response time** and provide an output to the user as quickly as possible, the **minimization** of the **maximum response time** and lower the worst-case bound, or the **minimization** of the **variance** of **response time** in order to provide a consistent system. These three past policies are common in **interactive systems**.

Some other policies are the **maximization** of the **throughput** (which would minimize the OS context switches and would use efficiently all the system resources) and **minimize** the **waiting time** (we would give the same CPU time to all the processes, with a possible increase of the average response time. These two policies are common in **batch systems**.

Before continuing, let's quickly make a note on some terminology, specifically on the meaning of **job/task** and **thread**.

**Definition:** Job/Task and Threads

A **job** or **task** is a general unit of CPU execution, while a **thread** is an actual unit of CPU execution

## 2.6 Scheduling Algorithms

Some of the most used scheduling algorithms are the following:

- **First Come First Serve** (**FCFS**);
- **Round Robin** (**RR**);
- **Shortest-Job-First** (**SJF**);
- **Priority Scheduling**;
- **Multilevel Queue** (**MQ**);
- **Multilevel Feedback-Queue** (**MFQ**).

### 2.6.1 First Comes First Serve (FCFS) and Round Robin (RR)

The **F**irst **C**ome **F**irst **S**erve algorithm is really simple: there is a queue structured with a **FIFO** structure, and the scheduler executes the jobs in arrival order. The scheduler takes over only in the case of an I/O operation, but any process could use the CPU for an indefinite amount of time. Such algorithm is **non-preemptive**. It's a very simple algorithm, but the waiting time might differ depending on the length of the processes (if lower-CPU burst processes sit in the back of the queue they'll need more time to get executed). Plus, the **convoy effect** may happen, so there would be a poor overlap CPU and I/O depending processes, since CPU-bound processes block the I/O-bound jobs.

The **R**ound **R**obin algorithm is similar to the **FCFS** algorithm, except that CPU bursts are delimited by some time limits called time quantum (or time slice). Each process has a **timer** when it receives the control of the CPU: if the job finishes before the time quantum, then it **gets swapped** just like the **FCFS** algorithm, but if the process can't get completed before the time quantum, then it gets **sent back** in the **last position** of the queue. Many time-sharing systems use this algorithm combined with timer interrupts. This is a **preemptive** algorithm.

For the Round Robin algorithm, the ready queue is treated as a **circular queue**, meaning that when all processes had a turn on the CPU the scheduler allows the first process of the first turn to go back into the CPU (if it didn't finish before the time quantum, that is). Round Robin allows to **split the CPU evenly** between all the processes, although the **average waiting time** is **higher** than with the other algorithms. The Round Robin algorithm heavily depends on the length of the time quantum: too large time quantum degenerates into the **FCFS** algorithm, while too small time quantum implies more context switches, which take some time and resources and will complete more slowly the various processes. A trade-off can be made: we can make the overhead for the context switching should be relatively smaller than the time slice.

We can try to compare **FCFS** to **RR**, and we can see an example here: suppose that we have have 5 processes $A$, $B$, $C$, $D$ and $E$, and each one needs 100 time units of CPU burst. Let us

| | | turnaround time | | waiting time | |
|---|---|---|---|---|---|
| Job | CPU burst | FCFS | RR | FCFS | RR |
| A | 100 | | | | |
| B | 100 | | | | |
| C | 100 | | | | |
| D | 100 | | | | |
| E | 100 | | | | |
| Avg. | | | | | |

suppose that we have a time quantum of 1 unit of time, that the context switch needs 0 units of time and that the arrival time needs 0 units of time (for all jobs). Now, we have the following situation:

1) with **FCFS** we start executing process $A$, then $B$, $C$, $D$ and finally $E$. Each process will take 100 time units of CPU burst and after 500 time units will complete all the processes. **RR** starts executing all the process also in the same order as FCFS, but on each time unit it will switch to the next process. Also RR after 500 time units will end executing all the processes, but each process terminates one time unit before the other, differently from FCFS where each process' termination is separated by 100 time units;

2) let's then analyze the waiting time for each process: process $A$ with FCFS remains in the waiting queue for 0 time units, but with $RR$ it stays there for 396 time units; process $B$ also has a lower waiting time with FCFS than with RR. If we check now the average times for all of the two cases we see that FCFS outruns RR in every case. Although the average is indeed lower with FCFS, RR has a much bigger impact if we consider the variance: all the processes end nearly at the same time, with just 1 time unit of difference for each process, while with FCFS we have to wait 100 time units between each process.

### 2.6.2 Shortest Job First (SJF) and Priority Scheduling

The **S**hortest **J**ob **F**irst plans the jobs according to their duration. In general, processes are scheduled based on the least expected amount of work to do until its next I/O operation or termination, where for "amount of work" we mean the CPU burst. Such algorithm is probably the most optimal when it comes to minimizing the average waiting time, and it works with both preemptive and non-preemptive schedulers.

The preemptive version of SJF is called **S**hortest **R**emaining **T**ime **F**irst (**SRTF**). The difference between SJF and SRTF is that SJF will make a **process complete** up **until** the **end of the process** itself, while with SRTF **preemption** occurs **whenever** a **process joins** the `ready` queue and its **predicted CPU burst** is **shorter** than the one of the executing process.

The downside of this algorithm is that it's almost **impossible** to know the next CPU

burst time of a job; moreover, long running CPU-bound jobs can starve, since I/O-bound processes have a lower CPU-burst and will implicitly have a greater priority.

How can we estimate the CPU time of a job? We can use previous runs of a process in order to compile an estimation. Another fast, accurate and simple method is the **exponential smoothing**:

**Definition:** Exponential smoothing

Let us consider as $x_n$ the **actual** length of the $n^{th}$ CPU burst and as $s_{n+1}$ as the **predicted** length of the $(n+1)^{th}$ CPU burst. Then, with a parameter $\alpha \in \mathbb{R}$ such that $0 \leq \alpha \leq 1$, we define the exponential smoothing as

$$s_1 = x_0$$

$$s_{t+1} = \alpha x_t + (1-\alpha)s_t$$

This formula is a **weighted average** between a previous observation and a previous prediction. With $\alpha = 0$ we consider $s_{t+1} = s_t$, thus the **observed bursts** are **ignored** and only the constant burst is assumed, while with $\alpha = 1$ we consider $s_{t+1} = x_t$, which means that we consider the **next burst** to be **equal** to the one of the **previous process**. In the case of $\alpha = 1$, the recent history doesn't count. Usually $\alpha$ is set to 0.5.

If we unroll from top to bottom the formula of the exponential smoothing, the following happens: first, we start with the general formula:

$$s_{t+1} = \alpha x_t + (1-\alpha)s_t$$

then, let's replace $s_t$ with the actual formula:

$$s_{t+1} = \alpha x_t + (1-\alpha)(\alpha x_{t-1} + (1-\alpha)s_{t-1}) \implies s_{t+1} = \alpha x_t + \alpha(1-\alpha)x_{t-1} + (1-\alpha)^2 s_{t-1}$$

let's replace it again:

$$s_{t+1} = \alpha x_t + \alpha(1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + (1-\alpha)^3 s_{t-2}$$

We arrive to a general unrolled version:

$$s_{t+1} = \alpha x_t + \alpha(1-\alpha)x_{t-1} + (1-\alpha)^2 s_{t-1} + \ldots + (1-\alpha)^{t-1} s_2$$

Now, let's consider more in depth the formula of $s_2$:

$$s_2 = \alpha x_1 + (1-\alpha)s_1 = \alpha x_1 + (1-\alpha)x_0 \quad \text{since } s_1 = x_0$$

Now, if we go back on the previous general unrolled version, we can substitute what we found for $s_2$ and we can group $\alpha$:

$$\alpha\left(x_t + (1-\alpha)x_{t-1} + (1-\alpha)^2 x_{t-2} + \ldots + (1-\alpha)^{t-1} x_1\right) + (1-\alpha)^t x_0$$

We can see how all the part that is multiplied to $\alpha$ at the beginning is nothing more than the sum of all the past $t$ observations, while $x_0$ is basically the bootstrap. If we assume that $\alpha > 0$, then we basically have a decreasing weight for each prediction going towards

the first predictions. Plus, we can see how the sum of the predictions increases exponentially.

As we said earlier, SJF has a preemptive version: **SRTF**. Such algorithm is very similar to SJF: whenever a new process arrives in the `ready` queue, the scheduler assigns the control of the CPU to the job that has the shortest remaining time. Let us consider for instance the following example:

| Job | Arrival time | CPU burst (time units) |
|---|---|---|
| A | 0 | 8 |
| B | 1 | 4 |
| C | 2 | 9 |
| D | 3 | 5 |


**Figure 2.8:** Example of an execution of the SRTF algorithm

Let's compute the average waiting time:

$$\frac{\overbrace{(17-0-8)}^{\text{Process } A}+\overbrace{(5-1-4)}^{\text{Process } B}+\overbrace{(26-2-9)}^{\text{Process } C}+\overbrace{(10-3-5)}^{\text{Process } D}}{4}=\frac{26}{4}=6{,}5$$

Let's see how SJF scores compared to FCFS and RR: supposing that all the processes join the `ready` queue at the same time, the scheduler will schedule, in order, processes $E$, $D$, $C$, $B$ and $A$. The average turnaround and waiting times with SJF are way lower than the ones of FCFS and RR, and even the single values for both timings are similar to the ones of FCFS and RR.

| | | turnaround time | | | waiting time | | |
|---|---|---|---|---|---|---|---|
| Job | CPU burst | FCFS | RR | SJF | FCFS | RR | SJF |
| A | 50 | 50 | 150 | 150 | 0 | 100 | 100 |
| B | 40 | 90 | 140 | 100 | 50 | 100 | 60 |
| C | 30 | 120 | 120 | 60 | 90 | 90 | 30 |
| D | 20 | 140 | 90 | 30 | 120 | 70 | 10 |
| E | 10 | 150 | 50 | 10 | 140 | 40 | 0 |
| Avg. | | 110 | 110 | 70 | 80 | 80 | 40 |

**Figure 2.9:** Comparison between FCFS, RR and SJF

There is another algorithm which is based on the same fundamental concept on which both SJF and SRTF are based: they work on **priorities**. In the case of SJF and SRTF, the priority is the **CPU burst time**. A more general algorithm exists: it's called **P**riority **S**cheduling (**PS**).

> The idea behind the priority scheduling algorithm is to have a more **general case of SJF**, where each job has a **priority** assigned and the one with **highest priority** gets **executed first**. In practice, priority is expressed via an integer: **the**

**lower the integer** of the priority, the more important a process is (where 0 means highest priority). There are no conventions on what a good priority is.

Priorities can either be assigned **internally** or **externally**:

- **internal** priorities are assigned by the OS using various criteria, such as CPU burst time, ratio between CPU and I/O burst time, system resources, and so on...
- **external** priorities are assigned by the users, depending on their needs.

Again, priority scheduling can be either **preemptive** or **non-preemptive**. There is one issue with priority scheduling though: **indefinite blocking** or **starvation**: if a task gets assigned a lower priority, then it might never get executed because some other jobs will always have a higher priority; such tasks might run either when the workload on the machine gets extremely light or when there is a crash or a shutdown/reboot. A solution to such problem is **aging**: with time passing, the priorities increase, so that older processes get their chance of being executed.

### 2.6.3 Multilevel Queue (MLQ) and Multilevel Feedback Queue (MLFQ)

The idea behind the **M**ulti**L**evel **Q**ueue (**MLQ**) algorithm is to have **different** `ready` **queues** depending on the type of processes. Each queue implements internally whatever scheduling algorithm is more appropriate for the kind of processes that are inside such queue. In order to select which queue should act, there are two ways:

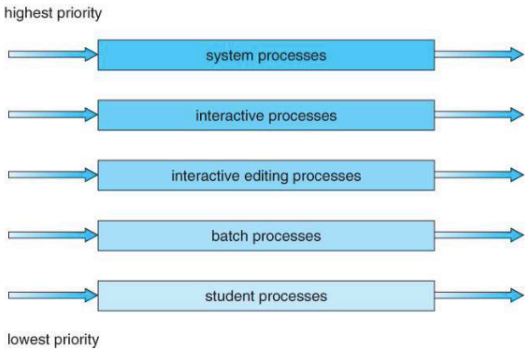



**Figure 2.10:** Example of queue classification

- **strict priority**: a **priority** is assigned to **each queue**, and the high-priority queues have the **precedence**: no process from the lower priority queues can run until all the processes in the high-priority queues are done;
- **round robin**: each queue gets a **time slice** on which a process (or more) from such queue can run on the CPU. The time slices can be **different** from queue to queue, depending on the importance of such queues.

With such process classification, it's **not possible** for a process to **switch** from one queue to another. If queues were scheduled with a round-robin criteria, and if the time slices were to be **different**, then they would **grow**, from the lowest priority queue to the highest priority one, in an **exponential way**.

There is another version of MLQ that allows for processes to switch from one queue to the other, and this version is called **M**ulti**L**evel **F**eedback **Q**ueue (**MLFQ**). Switching from one queue to another might be helpful for instance when the process has different

necessities from when it has to do I/O operations or CPU-bound operations, or simply whenever a process has been waiting for too long (in order to solve aging).

For the MLFQ algorithm, each process starts in the **highest priority queue**. If the process' time slice ends before the completion of the process, then the process' priority **drops** by one unit, and is moved to a lower priority queue (the one immediately below the first queue); if the time slice doesn't expire (because of a context switch due to an I/O request), the priority **increases** by one unit, and the process changes queue. In this way, **CPU-bound** processes **quickly drop** their priority, while **I/O bound** processes will have an **higher** priority.

The MLFQ algorithm is one of the most complex algorithms to implement, but is also the most flexible one. It has many parameters, such as the number of queues, the scheduling algorithm for each queue, the upgrade and demotion methods, etc... It's clear though that, with an **increase of fairness** and favouring of shorter processes also comes an **increase of** the **average waiting time**.

How can fairness be improved? First, a **fraction** of **CPU time** could be given **to each queue**, although this would be fair only if all the processes are evenly distributed among the queues; second, the **priority** of the jobs could be **dynamically adjusted** as they don't get scheduled. This last method would **avoid starvation**, but it could lead to an **increase of the waiting time** if the system gets overloaded (thus in an hypothetical scenario where all the jobs get to the highest queue)

### 2.6.4 Lottery Scheduling

Another scheduling algorithm is the lottery scheduling: the CPU gives to each process a certain number of "lottery tickets" and, on each time slice, a winner is picked before all the jobs **uniformly at random**. The CPU time is then assigned to the various processes according to the original distribution of the tickets. By the law of large numbers, as the number of time slices goes to infinity, then each process will eventually be executed.

Tickets are assigned as follows: short running jobs get more tickets, while long running jobs get few tickets (this prioritization aspect emulates SJF). In order to avoid starvation, each job must have at least one ticket. The algorithm degrades gracefully as the system load changes: adding or deleting a job affects all the other jobs in a proportional way.

We can express this algorithm mathematically: suppose that we give $m_{\text{short}}$ tickets to every short job and $m_{\text{long}}$ ticket to every long job; suppose also that we have $n_{\text{short}}$ short jobs and $n_{\text{long}}$ long jobs. Then we have

$$N = n_{\text{short}} + n_{\text{long}} = \text{total number of jobs}$$

and also

$$M = m_{\text{short}} \cdot n_{\text{short}} + m_{\text{long}} \cdot n_{\text{short}} = \text{total number of tickets}$$

We say that the probability, for each job of one of the two sides, to get control over the CPU is, respectively:

$$\text{CPU}_{\text{short}} = \frac{m_{\text{short}}}{M} \qquad \text{and} \qquad \text{CPU}_{\text{long}} = \frac{m_{\text{long}}}{M}$$

In general, if we have $m_i$ tickets assigned to job $i$ and a total numbers of jobs $N$, then the total number of tickets is equal to

$$M = \sum_{i=1}^{N} m_i$$

while the probability of job $i$ being scheduled is

$$p(i) = \frac{m_i}{M}$$

# Chapter 3
## Threads

So far, we assumed that we had a computer with a single core CPU that can execute only one process at a time. In some programming languages, we may recognise the word **thread**: usually, the program that we write, whenever it executes, is called **main thread**. Nowadays, modern OSs allow for the control of multiple threads all at once, which participates in the illusion of parallel programming.

But what is effectively a thread?

**Definition:** Thread

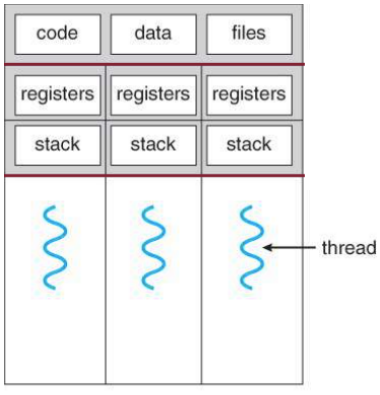
A **thread** is a basic unit of CPU utilization, consisting of a custom **program counter**, **stack** and a **set** of **registers**

Usually, heavy weighted processes don't have multiple threads, but instead count as one main thread. Applications that allow for multithreaded processes have, within one process, multiple threads, which have for each its own program counter, stack and registers but share the same data and the same memory.

Generically, a process defines the address space in the memory, the text, the data and the resources, while a thread defines a **single sequential execution stream** within a process. Each thread is bound to a specific process, even if one process may have multiple threads. In order to communicate between each others, threads **do not need** to use system calls.

We use threads usually when a task (or multiple tasks) that has to be performed is independent from the other tasks: a particular case may be for instance when it's known that a task blocks (or may block) the CPU; with threads we can allow for multiple tasks to not block each other. A couple examples:

- a word processor program, where one thread handles the spelling checker and another thread handles the keyboard input;
- a web server, where multiple threads handle multiple connections.



multithreaded process

Now, we could think that it would be probably easier to implement threads with smaller single-threaded processes, instead of complicating the scenery with multithreading, but that's not true for 2 particular reasons:

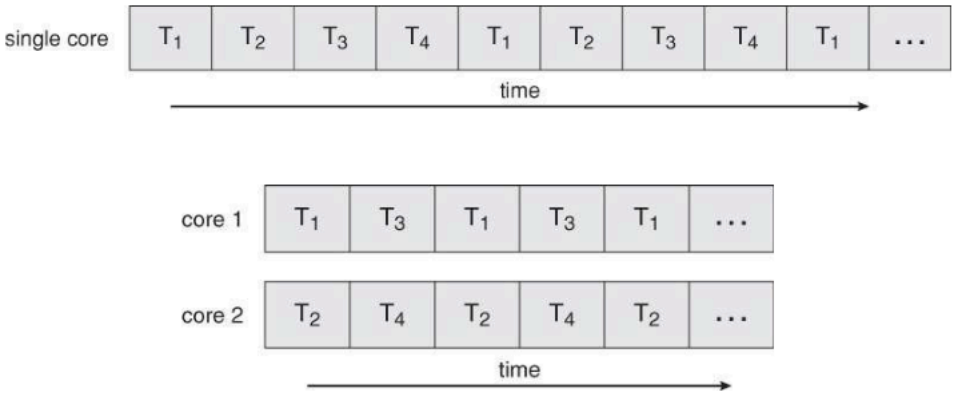
1) the communication between threads is way faster than the one between processes: no system calls are needed;

2) making context switches between threads is also faster than the one between processes;

In general, we can find 4 main benefits:

- **responsiveness**: some threads might be more responsive than other threads, if such threads are occupied in intensive computations;
- **resource sharing**: threads share the same data space, even if such space gets partitioned between each thread;
- **economy**: creating and managing the life cycle of threads is faster and easier than creating and managing processes: even the fact that for creating a process we have to use two system calls makes a huge difference;
- **scalability**: on multi-cores architectures, we can spread threads on various CPU cores, allowing for true parallel programming.

Nowadays all CPUs are built with multiple cores, allowing the OS to spread processes and threads across all the cores and doing true parallel programming. Of course, new algorithms are needed in order to schedule efficiently all the threads across the CPU. Some CPUs have now developed also some software which allows threads to run concurrently on a **single** core, such as Intel Hyper-Threading.




Suppose that we have to implement a single program that takes as input a positive integer $N$ and produces the sum from 1 to $N$. The easiest solution would be to make a `for` loop which adds to a variable $N$ all the number from 1 to $N$. This way we get a CPU bound process. But what happens when $N$ grows? The more it grows, the more the becomes expensive in terms of computation, so how can we improve it?

Let us consider the following setups:

- **1 CPU core, 1 process and 1 thread**: we just have a program that runs in linear time; there is no parallelism and no concurrency;

- **1 CPU core, *M* processes, 1 thread**: the idea is to divide *N* into multiple chunks, compute the sum of each chunk (where each chunk is associated to each process) and then add everything up. But is there any advantage? There is none: we still have one CPU core, and each process has to switch in order to compute each chunk's sum; actually, we take more time because we also do context-switches, and also because we have to sum up all the chunks (and thus having **inter-process communication**);
- **1 CPU core, 1 process and *M* threads**: this situation is similar to the one explained earlier: we don't really get any speedup. We have a speedup in respect to the previous idea;
- ***M* CPU cores, *M* processes and 1 thread**: here we have some sort of speedup, since the chunks execute in parallel and then one of them computes the final sum. Still, we have to use system calls to retrieve the data, so we could go even faster;
- ***M* CPU cores, 1 process, *M* threads**: this is the most efficient solution: each thread runs concurrently on each core, and since they share memory then it's easier to compute the final sum.

Not all tasks are only CPU bound, most of them require some I/O operations; usually, whenever the CPU is being used, I/O operations are being executed and vice versa. Some examples of tasks that would need both CPU and I/O are for instance disk defragmentation or the retrieval of large piece of data. In such cases, multithreading might be helpful, even on a single core CPU: even if some time must be spent in order to split the CPU and I/O intensive tasks, then it's more helpful on the long term to have the two tasks alternate on the CPU: even if the CPU will be stresses, then the I/O bound gap will be reduced or, eventually, eliminated.

## 3.1 Multithreading

How is multithreading implemented in the various OSs? We can classify OSs in 4 classes, such as the following:

| | **Single Address Space**<br>Uniprogramming | **Multiple Address Spaces**<br>Multiprogramming |
|---|---|---|
| **Single Thread** | MS-DOS | Unix |
| **Multiple Threads** | Xerox Pilot | Mach, NT, Solaris |

Right now mostly every system allows for multithreading: back in the past though they weren't supported. Now most modern systems allow for multithreading support, more specifically there are two kinds of support that are given: a specific support for **kernel threads** (so threads executed at the kernel level) and another one for **user threads** (thus threads at the user levels).

**Kernel threads** are managed by the OS kernel itself, so if we need to manage a new kernel thread then a system call is required. **User threads** instead are managed in the user space by a **thread library**, which doesn't need the OS intervention. Some examples might be the `thread` library of Java or the `tokio` library for Rust.

### 3.1.1 Kernel Threads

We consider a **kernel thread** to be the smallest unit of execution that can be scheduled by the OS, which is also responsible for the threads' management. We know that each process has a PCB, but what about threads? They have a **T**hread **C**ontrol **B**lock (**TCB**), which is similar to the PCB. Kernel threads are managed via system calls.

There are some advantages regarding kernel threads: first, the kernel has **full knowledge** regarding all the running threads, which is helpful when the CPU has to organize the threads **scheduling**, since it can decide whether to give to a thread more CPU time rather than to another thread. It's a good architecture for applications that may **frequently block** the CPU, and **switching** between threads is **faster** rather than switching between processes.

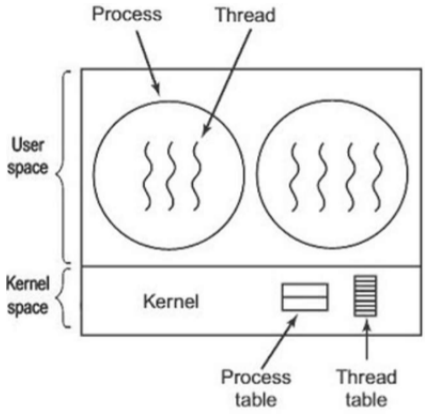



**Figure 3.1:** Representation of a **kernel thread**

There are also some disadvantages: since we split the thread into two parts (the data is in the kernel and the execution is related to the user mode), we **augment** the **complexity** of the program; it's **slow** and **inefficient** when it comes down to **fetch information** from the OS (since we use system calls); **context switching**, even if it's lighter, it still **needs the kernel** to manage it.

### 3.1.2 User Threads

**User threads** are managed entirely by the run-time system, which is a **thread library** which differs between each programming language. The **OS doesn't know** anything **about** the user-level **threads**, and it treats **each** user-level **thread** as a **single-threaded process**. Ideally, the call of a thread should be as fast as a function call. Threads that run only and uniquely on the user level are called **green threads**

Also user-level threads have some advantages: they are really **fast** and **lightweight**, and **scheduling policies** are more **flexible**, since they are managed by the OS at a user-level, and not at a kernel-level. They can be implemented in **any** OS, even the ones that have no support for kernel level threads. There are **no system calls involved**, since everything happens on the user level, and there is **no** actual **context**

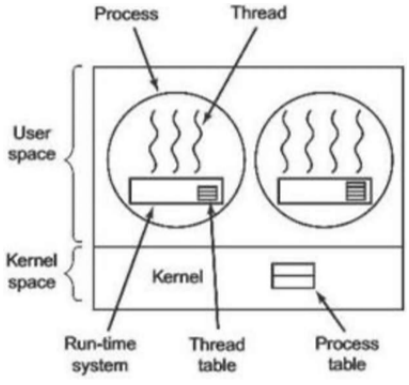


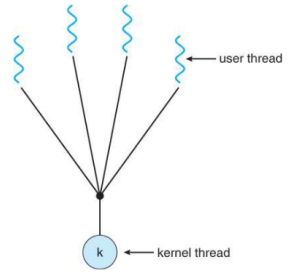
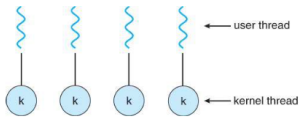
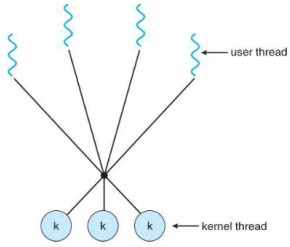
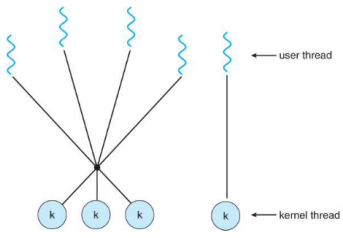

**Figure 3.2:** Representation of a **user thread**

**switch**.

User threads come with also some disadvantages: there is **no true concurrency** of multi-threaded processes, since for the OS they all seem like single-threaded processes, and it follows that the OS may make some **poor scheduling decisions**; there would be a **lack of coordination** between kernel and threads and it would require **non-blocking** system calls, since otherwise all the other threads within a process would need to wait.

### 3.1.3 Hybrid Threads

In modern OSs we rarely have pure kernel-level only threads and user-level only threads: usually a combination of both is used. In order to use both of them, **lightweight processes** are used: they are the **counterpart** of the **kernel threads** in the **user level**. In fact, there is a one-to-one mapping between each lightweight process and each kernel thread. There are various ways that allow us to map user threads to kernel threads:

| Model image | Model description |
|---|---|
|  | **Many-to-one model**<br>Many user threads are all mapped to the same kernel thread and the process allows the threads to run only one at a time. A single kernel thread can operate on a single CPU, multi-user thread processes cannot split the threads on multiple CPUs. If a thread makes a blocking system call, then the whole process is blocked. Such model is said to be **purely user-level** based. |
|  | **One-to-one model**<br>There is a separate kernel thread reserved for each user thread. Such model allows to overcome the limitations of blocking system calls, and the various kernel threads can be split across multiple CPUs. Managing this system is more expensive in terms of computational power, and it may slow down the system. When using this model, there usually is a limit to the number of kernel threads that can be created. Such model is **pure kernel-level**. |
|  | **Many-to-many model**<br>This model multiplexes between the various user threads onto an equal or smaller number of kernel threads. Users have no restrictions regarding the number of threads created, and the kernel levels can be split across multiple CPUs. Moreover, blocking system calls do not block the whole process. |
|  | **Two-level model**<br>Is a variant of the many-to-many model, since it's a mix between that model and the one-to-one model. It's useful because it increases the flexibility of scheduling policies. |

## 3.2 Thread Libraries

We mentioned the **thread libraries** quite few times in the previous pages, but what are they? Thread libraries are an abstraction level, and are APIs that allow to manage and create threads. There are mainly two ways to implement a thread library:

- on the **user space**: the API is implemented purely and only on the user space. In order to communicate with the threads we use **function calls**;
- on the **kernel space**: the API is implemented in the kernel, which has to support multithreading. Communication with the threads is done via the use of **system calls**.

There are 3 main thread libraries used today: **POSIX Pthreads** (used on UNIX based OSs, can be implemented both at a user-level and at a kernel-level. Such library is an extension of the POSIX standard, which only defines the specification for the library, not the implementation. For this library, all global variables are shared between all threads, and one thread can wait for the others before continuing working on something), **Win32 threads** (a kernel library for managing threads on Windows systems) and **Java threads** (a library which varies depending on the OS on which the JVM is running: it adapts itself whether it's running on Pthreads or Win32).

## 3.3 Thread Synchronization

We know that threads can cooperate to work on a same process, in order to achieve better performances, and in order to cooperate, threads must be **synchronized**: when working and executing critical operations we must ensure that **only one thread at a time** is working (such critical operations are also called as **critical sections** or **critical regions**). If two threads enter a critical section together, then we might achieve some data corruption, or not get the result that we wanted.

A critical section can thus be executed by only one thread at a time. In order to access such section, a thread must first acquire a **lock**, which is a sort of "pass". When exiting the critical section, the thread has to release the lock for another thread. Such process of lock acquiring and releasing involves **waiting**: it's a price that must be paid in order for the threads to act synchronously.

Any synchronization solution to the critical section must satisfy 3 key properties:

- **mutual exclusion**: only one thread at a time can be in the critical section;
- **liveness**: if no process is working in the critical section, then any concurrent thread that wants to enter in the critical section must be able to enter it;
- **bounded waiting**: if a process requests to enter in the critical zone, then it will access it eventually, and there is a limit to how much it has to wait, so that starving is avoided.

Synchronization is implemented through various tools, such as the aforementioned locks. Another tool are the **semaphores**, which are a generalization of the locks; **monitors** are used as well: they are a synchronization construct which checks for a certain condition to become either true or false, such as checking if the lock for a thread is acquired or not. Such tools might need waiting and hardware support.

### 3.3.1 Locks

Locks have 2 main **atomic** methods that satisfy the **mutual exclusion** property: `lock.acquire()` and `lock.release()`. Such methods are both **atomic** (they can't be interrupted) and they respectively do the following:

- `lock.acquire()`: acquires the lock if such lock is free;
- `lock.release()`: release the lock and notify the waiting threads.

There are some rules for using locks: the lock must be **acquired** before entering the **shared data** and then **released** upon finishing and, if it's not used by anyone at a given moment in time, it should be **free to pick** for everyone. Plus, the lock can be acquired by only **one thread at a time**.

How do we make an operation **atomic**? There are two types of operations that help us: **high-level** and **low-level** operations. **High-level** operations are implemented on the software level, and some examples are `locks`, `monitors`, `semaphores`, `send` and `receive`. Such high-level operations need some **low-level** operations in order to work though. **Low-level** operations are implemented on the hardware level, and some ways to achieve them is for instance via `disabling` the `interrupts`, or by using `atomic instructions` such as `test&set`.

Why is disabling the interrupts an operation that could be done in order to achieve atomicity? Because without interrupts we would suspend the context switches, so the scheduler won't take over the execution of some parts of code. The scheduler would act in two cases: if there is an internal event, such as the thread releasing voluntarily the CPU, or if there is an external event, such as an interrupt. Our goal is to avoid that the CPU scheduler takes control of the CPU while an `acquire()` method is being run.

On single CPU systems, internal events can be disabled by prohibiting threads to make I/O operations during a critical section, while external events can be avoided by disabling the interrupts. In this case we cover all the cases where the current thread would lose control of the CPU, might that be a voluntary or an involuntary lost. Here follows a rough implementation of locks:

```
Class Lock {
  public void acquire(Thread t);
  public void release();
  private int value; // 0=FREE, 1=BUSY
  private Queue q;

  Lock() {
    // lock is initially FREE
    this.value = 0;
    this.q = null;
  }
}
```

```
public void acquire(Thread t) {
  disable_interrupts();
  if(this.value) { // lock is held by someone
    q.push(t); // add t to waiting queue
    t.sleep(); // put t to sleep
  }
  else {
    this.value = 1;
  }
  enable_interrupts();
}
```

```
public void release() {
  disable_interrupts();
  if(!q.is_empty()) {
    t = q.pop(); // extract a waiting thread from q
    push_onto_ready_queue(t); // put t on ready queue
  }
  else {
    this.value = 0;
  }
  enable_interrupts();
}
```

We need to implement `disable_interrupts()` and `enable_interrupts()` as **system calls**, because interrupts are handled by the OS, while threads are handled by the user.

Another way to implement threads in an atomic way is to use the **hardware atomic instructions**: we know that in machine language we have some instructions for loading a value from memory, manipulate it and then store it back. **Loading** and **storing** are usually **atomic instructions** because interrupting them would lead to data corruption. in `RISC-V` we saw the `lw`, `add` and `sw` instructions, which are respectively the read, modify and write instructions. Some machines implement the **read-modify-write** logic, which does all the three things in one shot. On single-core processors, this would be fine and easily implementable, but on multi-core processors we would need to make sure that at one given time only one processor executes this instruction. Some atomic instructions that follow the **read-modify-write** logic are:

- `test&set`: sets 1 in the memory location provided and returns the old value stored inside that memory location;
- `fetch&add`: increments the value specified in a memory location by a specified value;
- `compare&swap` (called `compare&exchange` on `x86`): compares the value contained in a memory location with a specified value and, if the two values are equal, it modifies the content of the specified memory location with a new given value.

Let's give a look at the various implementations of locks with the previously mentioned atomic instructions: we consider the code on the right for the `test&set` instruction:

For such code, we can examine 2 cases: whenever `value` is `0` or `1`. If the value is initially 0, then the lock is said to be **free**. Now, if we try to acquire it, `test&set` will set 1 (so the lock will become **busy**) and return 0; the `acquire()` method will then exit the while loop and the acquisition terminates. If the lock is initially busy (so `value = 1`), then the `acquire()` method will continue to loop until we use the `release()` method. There is a problem with such implementation: the CPU, while it waits, it's busy doing nothing; moreover, there is no possibility to implement a priority queue for the threads that might need more than others a certain resource.

```
Class Lock {
    public void acquire();
    public void release();
    private int value;

    Lock() {
        // lock is initially free
            this.value = 0;
    }
}

// Implementation of the method

public void acquire() {
    while(test&set(this.value) ==
        1) {
        // while busy do nothing
    }
}

public void release() {
    this.value = 0;
}
```

We now see that both the **disabling of interrupts** and the **use of atomic instructions** have some problems: the first one is both **overhead** (it needs to invoke the kernel via a system call) and **unfeasible** with multiprocessor architectures: disabling the interrupts for one core means interrupting them for all cores; the latter one is both **busy waiting**

(while a thread waits it does nothing) and **unfair** (there is no priority queue for establishing who's the next thread that will acquire the lock).

We can improve the previous lock code in order to reduce the busy-waiting:

**Lock class**

```
Class Lock {
    public void acquire(Thread t);
    public void release();
    private int value;
    private int guard;
    private Queue q;
    Lock() {
    // lock is initially free, so this.value = 0;
    }
}
```

**acquire() method**

```
public void acquire(Thread t) {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(this.value) {
        q.push(t); t.sleep_and_reset_guard_to_0();
    } else {
        this.value = 1;
        this.guard = 0;
    }
}
```

**release() method**

```
public void release() {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    if(!q.is_empty()) {
        t = q.pop(); push_onto_ready_queue(t);
    } else {
        this.value = 0;
    }
    this.guard = 0;
}
```

It's not possible to have no busy-waiting at all, but we can minimize it by atomically checking the lock and, in the case where the lock is busy, the thread can give up the control of the CPU.

In general, locks are a useful low-level tool that can ensure **mutual exclusion** and also protection regarding critical sections of the code. Locks can be implemented either via the disabling of the interrupt signals (which can make the CPU miss or delay important events) or via atomic instructions (which have a busy-waiting time and can go on spinlock, which is the action that a thread does when it loops while it tries to acquire a lock). There are two high-level synchronization primitives that can be used instead of locks, which are **semaphores** and **monitors**.

### 3.3.2 Semaphores

We saw that locks are a low-level data structures that grant protection regarding some critical sections, but are somewhat inefficient because of the possible **spinlock**. Semaphores are an high-level alternative to locks.

**Definition:** Semaphores

An **high-level** data structure that provides **binary mutual exclusion** to critical sections. Such data structure can also act as an **atomic counter**. It's a generalization of the concept of locks made by **Dijkstra** in 1965

Semaphores are a special type of integer which supports 2 atomic operations:

- `wait()`: also called `P()`, it decrements the integer, and blocks every requesting thread until the semaphore is open again;

- `signal()`: also called `V()`, it increments the integer, allowing another requesting thread to access a resource.

Each semaphore has a related **queue** for waiting threads and, whenever `wait()` is called by a thread, if the semaphore is open then the thread continues, otherwise the thread blocks itself and stays in the queue until the semaphore becomes open again. When `signal()` is called, the semaphore becomes open again and, if a thread is waiting blocked in the queue, it gets unblocked and it proceeds to the critical section; if no threads are waiting in the queue, then the `signal` is **remembered** for the next thread that will make a request to access the critical section. The `signal()` state can be considered as **stateful**, since it has an "history" of its past actions.

There are two types of semaphores: **binary semaphores** (also called **mutex**, which are very similar to locks) and **counting semaphores**:

- **binary semaphores** guarantee mutually exclusive access to a resource, and its associated integer value can take only two possible values: either 0 or 1. Whenever it's initialized, it takes as value 1 (so open);
- **counting semaphores** are used whenever there are multiple shared resources that have to be managed at the same time. Whenever it's initialized, it is set to the number of resources. A process can access to a resource as long as at least one resource is available.

The key idea is that whenever a thread wants to access to a critical section, it first calls the `wait()` method of the semaphore, accesses to the critical section, does its operations and, whenever it's done, it calls the `signal()` method of the semaphore. Of course, if the semaphore is not open (so it's 0), then the thread is blocked, and such thread will resume only when the `signal()` method will be used. In the case where a process acts in place of a thread, the blocking and unblocking is done by the OS.

Here follows an implementation of semaphores:

**Semaphores class**

```
Class Semaphore {
    public void wait(Thread t);
    public void signal();
    private int value;
    private int guard;
    private Queue q;

    Semaphore(int val) {
        // initialize semaphore
        // with val and empty queue this.value = val;
        this.q = null;
    }
}
```

**`wait()` method**

```
public void wait(Thread t) {
    while(test&set(this.guard) == 1) {
        // while busy do nothing
    }
    this.value -= 1;
    if(this.value < 0) {
        q.push(t);
        t.sleep_and_reset_guard_to_0();
    } else {
        this.guard = 0;
    }
}
```

**`signal()` method**

```
public void signal() {
    while(test&set(this.guard) == 1) {
    // while busy do nothing
    }
    this.value += 1;
    if(!q.isEmpty()) { // this.value <= 0
        t = q.pop();
        push_onto_ready_queue(t);
    }
    this.guard = 0;
}
```

Notice how the `wait()` and `signal()` method are **atomic**, since they are implemented with `test&set`. Of course, this is a specific implementation, and semaphores can be implemented in many different ways, but still, the implementation must be **atomic**.

Why should we use semaphores instead of locks? First, semaphores ensure a safe **mutual exclusion** to guard critical sections (in order to do so, we set the semaphore with `value = 1` at the initialization and we guard the critical section by using `wait()` and `signal()`) and can impose **scheduling constraints** (by setting the value to 0, we impose some waiting time on threads. Some examples of this constraint are the `join()` and `waitpid()` system calls).

Let us consider a simple synchronization problem: the **producer - consumer** problem. Simply put, a producer process produces an item which is then put in a common buffer, while the consumer consumes the items in the buffer. Let the following code:

```
void producer() {
    while(true) {
        /* produce an item in nextProduced */
        while(counter == BUFFER.SIZE) {
            /* do nothing */
        }
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER.SIZE;
        counter++;
    }
}

/* On pseudocode, that would be:
 *
 * register(1) = counter
 * register(1) = register(1) + 1
 * counter = register(1)
 */
```

```
void consumer() {
while(true) {
    while(counter == 0) {
        /* do nothing */
    }
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER.SIZE;
    counter--;
    /* consume the next item in nextConsumed */
    }
}

/* On pseudocode, that would be:
 *
 * register(2) = counter
 * register(2) = register(2) - 1
 * counter = register(2)
 */
```

With the `counter` variable we are keeping tracks of the items that are currently in the buffer. Why do we say that a race condition may happen? Because the counter is subject to changes from both the producer and the consumer, there is no synchronization on who acts first.

For instance, let us assume that the following happens (properly because there is no synchronization mechanism):

| | | | | |
|---|---|---|---|---|
| $T_0$ | *producer* | executes | $register_1$ = `counter` | [$register_1 = 5$] |
| $T_1$ | *producer* | executes | $register_1 = register_1 + 1$ | [$register_1 = 6$] |
| $T_2$ | *consumer* | executes | $register_2$ = `counter` | [$register_2 = 5$] |
| $T_3$ | *consumer* | executes | $register_2 = register_2 - 1$ | [$register_2 = 4$] |
| $T_4$ | *producer* | executes | `counter` = $register_1$ | [`counter` = 6] |
| $T_5$ | *consumer* | executes | `counter` = $register_2$ | [`counter` = 4] |

At the end of the fifth thread, the counter will have value 6, but the real value should be 5. We got 6 because there was no synchronization between threads, and thus a race condition was formed. In order to fix this problem, we should insure:

- **mutual exclusion**: the access to the buffer should be granted to one thread at a time;

- **scheduling constraints**: the producer can put a new item in the buffer if and only if the buffer is not full; vice versa, the consumer can consume an item from the buffer if and only if the buffer is not empty.

So, in general, semaphores are useful when **mutual exclusion** has to be granted (similarly to locks), when a pool of **common resources** has to be shared and its access has to be controlled, but also when some specific **scheduling constraints** have to be enforced. They have some **issues** though: firstly, they are essentially **shared global variables** that have **no direct connection** with the data that they control; secondly, they serve **multiple purposes**, and thus they are harder to implement for only one purpose rather than another; lastly, their **correctness depends on** their **implementation**. We can fix these problems by using another more high-level primitive: such primitive are the **monitors**.

### 3.3.3 Monitors

A monitor is a construct that has direct control access to the shared data, and it's similar to a Java or C++ class which embodies together **data**, **operations** and **synchronization**. Differently from locks and semaphores, the synchronization code is **added by** the **compiler** and is **enforced** at **runtime**.

Unlike classes, monitors **guarantee mutual exclusion** (so only one thread at a time can execute a monitor's method) and require all **data** to be **private** (this is required such that only one getter or setter at a time can access to the data). A more formal definition of monitors follows:

**Definition:** Monitor

A monitor defines a **lock** and zero or more **condition variables** for managing the access to the shared data. The lock is used to ensure that **only one thread** at a time **is active** within the monitor at any time, providing then **mutual exclusion**.

For instance, let us consider the Java class `Queue` on the left: in order to transform such class in a monitor, it's necessary to just make all the data private and all the methods synchronized, be them private or not. In our case, the monitor will be like the code on the right:

| Class | Monitor |
|---|---|
| `class Queue {`<br>`  public ArrayList<Item> data;`<br><br>`  public void add(Item i) {`<br>`  data.add(i);`<br>`  }`<br><br>`  public Item remove() {`<br>`  if (!data.isEmpty()) {`<br>`  Item i = data.remove(0);`<br>`  return i;`<br>`  }`<br>`  }`<br>`}` | `class Queue {`<br>`  private ArrayList<Item> data;`<br><br>`  public synchronized void add(Item i) {`<br>`  data.add(i);`<br>`  }`<br><br>`  public synchronized Item remove() {`<br>`  if (!data.isEmpty()) {`<br>`  Item i = data.remove(0);`<br>`  return i;`<br>`  }`<br>`  }`<br>`}` |

Let us suppose for instance, given the monitor class `Queue` on the right, that a thread tries to remove an element from an empty list? In the class on the left it would just not

do anything, but with the class on the right, the thread will wait until something is available on the queue, thus **sleeping inside** the **critical section**. By doing this though, the thread would keep for itself the lock, never releasing it; by never releasing it, no other thread can access in general the queue or try to wake up the sleeping thread. This condition is called **deadlock**.

How can we avoid this situation? We can use **condition variables**: conceptually, they are a **queue of threads**, associated with a lock, on which a thread waits until some condition becomes true (they are **not a boolean object**); they enable then a thread to **sleep within** a **critical section**. The difference with the previous monitor class is that, with the condition variables, monitors **atomically release** the **locks** held by any thread whenever such thread goes into the sleep status.

Each condition variable has 3 possible operations:

- `wait()`: releases the lock and puts the thread into the sleep queue (the queue of waiters) **atomically**;
- `signal()`: wakes up a waiting thread in the waiting queue;
- `broadcast()`: wakes up all the waiting threads in the waiting queue.

An important rule with monitors is that a thread must hold the lock while executing condition variable operations, such that we still have mutual exclusion.

In Java, condition variables are implemented in the following way (let us consider for instance the previous example where a thread wants to remove an item from an empty list):

1) we first use `wait()` to give up the lock and put the thread in a waiting state;
2) whenever a thread will add an item into the list, we can use either `notify()` to signal to the sleeping thread that the condition it was waiting on is finally satisfied, or eventually we could use `notifyAll()`, in order to wake up all the waiting threads. Concretely, we have only **one condition variable per object**.

Here follows a possible implementation of a `Queue` class:

```
class Queue {
    private ArrayList<Item> data;

    public void synchronized add(Item i) {
        data.add(i);
        notify();  // the same as signal(), it's the Java'ish version
    }

    public Item synchronized remove() {
        while (data.isEmpty()) {
            wait();
        }
        Item i = data.remove(0);
        return i;
    }
}
```

Condition variables, while they could seem equal to semaphores regarding the operations, they have entirely different semantics. First, the **access** to the **monitor** is controlled by a **lock**, and that's not the case with **semaphores** because they are a **generalization** of locks; second, the `wait()` **method** has a complete different meaning: on **monitors**, it not only **blocks** the calling thread, but it also **gives up** the **lock** in possession of the thread; on **semaphores**, it **only blocks** the thread on the queue. Also the `signal()` method has some fundamental differences: with monitors, it wakes up a thread in the queue but, if the waiting queue is empty, then the signal gets lost; with semaphores instead, since `signal()` increases a counter, it's like as it had a "memory".

With monitors, there are two types of `signal()` methods: **Mesa** and **Hoare** style:

- **Mesa style**: such implementation is used with various OSs (such as Nachos, Java and much more) and is preferred over the Hoare implementation by many programmers. For such style, whenever a thread uses the `signal()` method, a waiting thread from the `waiting` queue is then moved to the `ready` queue; meanwhile, the signaling thread continues in the critical section, thus making the conditional variable not necessarily true. Whenever the new thread will have to enter the critical section, it'll have to check again until the conditional variable becomes true;
- **Hoare style**: this implementation, even if it's not that much used in real world applications, it's actually used by many textbooks. Differently from the **Mesa style**, whenever a thread uses the `signal()` method, it immediately switches to a waiting thread, allowing the next thread from the `waiting` queue to execute: in fact, there is a guarantee that the condition variable will be true whenever the new thread will execute.

We can summarize the two styles with the following two pieces of pseudocode:

**Mesa**

```
// Pseudocode
while(empty) {
    wait(condition);
}


// Java implementation
class Queue {
    private ArrayList<Item> data;

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public synchronized Item remove() {
        while (data.isEmpty()) {
            wait();
        }
        Item i = data.remove(0);
        return i;
    }
}
```

**Hoare**

```
// Pseudocode
if(empty) {
    wait(condition);
}


// Java implementation
class Queue {
    private ArrayList<Item> data;

    public void synchronized add(Item i) {
        data.add(i);
        notify();
    }

    public synchronized Item remove() {
        if (data.isEmpty()) {
            wait();
        }
        Item i = data.remove(0);
        return i;
    }
}
```

We can apply monitors to solve another problem: the **readers - writers problem**. Let us consider a shared database system, where there are two classes of users: the **readers** (which only read the database and can never modify the database) and the **writers** (which can both read and modify the database).

The **simplest solution** to regulate the access to the database is to **use** a **single lock** on the data for each operation: such solution is **too restrictive** though, since with such implementation only one user between readers and writers can use the database at a given time, be it for reading or writing. The **optimal solution** would be to let **only one writer** at a time to write into the database and **multiple reading users** at a time to read the database data.

We want to put some **constraints**: **readers** can **access** the database only when there are **no writers** acting on the database, while **writers** can **access** the database when **no writers or readers** are using the database. Moreover, we want **only one thread** to manipulate the state variables at a time.

This problem has two variations, depending on who we want to have the priority:

- **First readers - writers problem** (gives priority to the readers): if a reader wants to access to the data and there are no writers accessing, then the access to the database is given to the reader. **Writers could starve** though, since the priority would always be given to the readers;
- **Second readers - writers problem** (gives priority to the writers): whenever a writer wants to access the queue, it jumps to the head of the waiting queue. Similarly to the first problem, we could have a **starvation** of the **readers**.

## 3.4 Deadlocks

Sometimes, we may have a situation where we have multiple threads that in order to work need multiple resources all at once. For instance, let us assume the following situation: we have 5 threads $A$, $B$, $C$, $D$ and $E$ and 5 sections in the memory $M_1$, $M_2$, $M_3$, $M_4$ and $M_5$. A thread can either be comparing data from two nearby memory sections or wait until two nearby memory sections are free. For this particular example, $M_1$ and $M_5$ are considered nearby memory sections. The question is: how can we make all the threads complete their task without making any thread starve?

The most trivial solution would be to make only one thread acquire the memory sections at any given time. This is not optimal though, because we could have some concurrency, and this way we are wasting some space. We could have a solution like the following:

```
Semaphore memorySections[5];

while(True) {
    memorySection[i].wait(); // acquire left memory section
    memorySection[(i+1)%5].wait(); // acquire right memory section
```

```
        compare();

        memorySection[i].signal(); // release left memory section
        memorySection[(i+1)%5].signal(); // release right memory section

        waitUntilNextComparison();
    }
```

For such solution, we consider with $i$ the abstract and figurative space between one memory space and the other. The idea is to make all threads start, let them acquire both the left and right memory section, start comparing and then finally release the memory sections. Would it work though if we released 5 threads? It wouldn't.

The reason why it wouldn't work is because all the threads would, on the same moment, acquire the left memory section and then they couldn't acquire the right memory section, because it would've been already taken by another thread as its left memory section. This situation is called **deadlock**: some resources needed by multiple threads are locked behind a lock and can't be acquired because the threads will never release them.

**Definition:** Deadlock

A condition where two or more **threads** are waiting for an **event** that can be only generated by the very same threads.

Deadlock can also be described as a competition between threads in order to acquire a possession over a finite number of resources.

We need to find another solution. We could check, before acquiring the memory sections, if they are available and not already taken by a thread. We could introduce a state variable, for which we identify whether a thread is preparing to take control over a memory section or not, and then depending on its state we would act accordingly. This would work only if all the methods were synchronized.

This is just a silly example, but this kind of patterns is really common when dealing with threads. For instance, whenever we have to print multiple files, we could have two threads responsible for printing such files; if the two threads access to the printer and to the disk in different moments, they could block each other in a deadlock situation, where one thread holds one resource and the other holds the other resource.

Deadlock can be prevented, avoided and detected: we call such actions **deadlock detection** (finding instances of deadlocks and recovering them), **deadlock prevention (offline)** (imposition of restrictions and rules to follow in order to write a deadlocks-free program) and **deadlock avoidance (online)** (a runtime process support checks whenever a deadlock might happen and tries to avoid such situation by managing the resources).

Deadlocks are a different situation than starvation: starvation is the situation where a thread is waiting indefinitely for a resource because other threads are using it and making some progresses; deadlock is the situation where a thread is waiting for a resource but the other threads are not making any progress with the acquired resources that are needed by the first thread.

In general, deadlock happens whenever these 4 following conditions are all simultaneously satisfied:

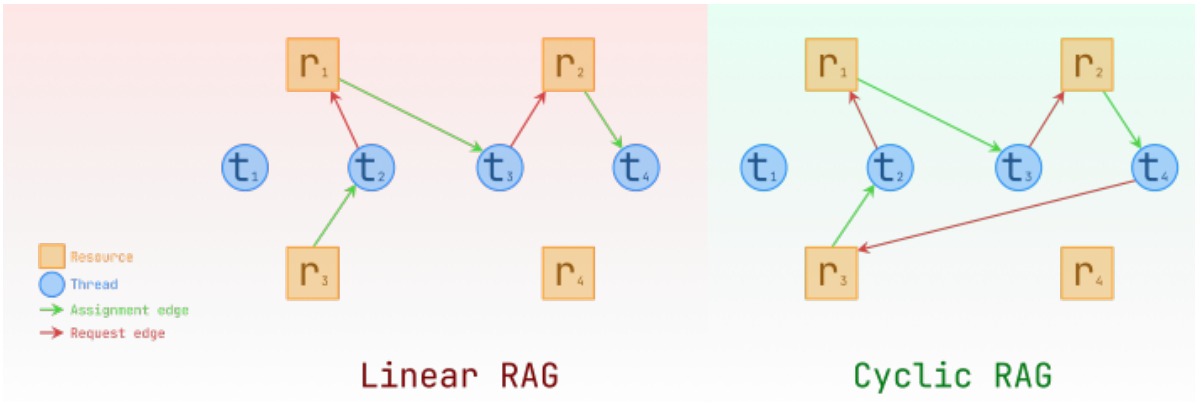
- **mutual exclusion**: at least one thread holds one non-shareable resource;
- **hold and wait**: at least one thread holds a non-shareable resource and is waiting for another thread to release a locked resource;
- **no preemption**: a thread can only release a resource voluntarily: the OS or another thread have no power regarding the resources acquired by such thread;
- **circular wait**: there is a set of waiting threads $t_1$, ..., $t_n$ where $t_i$ is waiting for thread $t_{(i+1)\%n}$.

### 3.4.1 Deadlock Detection

How can we detect a deadlock? In general whenever a **cycle** is formed: a thread asks for a locked resource, which is held by a thread which is asking for another locked resource, etc... until we make a cycle between threads and resources. Such cycle can be expressed as a **directed graph** $G(V, E)$ called **R**esource **A**llocation **G**raph (**RAG**) where:

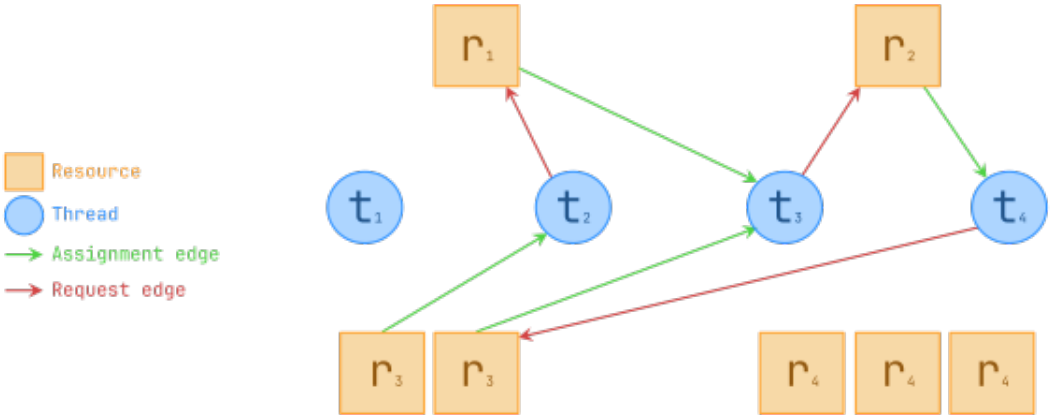
- $V$ is the set of **vertices**, which comprehend both **threads** $\{t_1, ..., t_n\}$ and **resources** $\{r_1, ..., r_m\}$;
- $E$ is the set of **edges** between threads and resources. There can be two types of edges:
  - **request edges**: a **directed** edge $(t_i, r_j)$ indicates that $t_i$ has requested $r_j$, but it hasn't acquired it yet;
  - **assignment edges**: a **directed** edge $(r_j, t_i)$ indicates that the resource $r_j$ has been assigned by the OS to the thread $t_i$.

If the RAG has no cycles it means that **no deadlock** will ever **exist**, but why is is to? Let us consider the following two graphs:

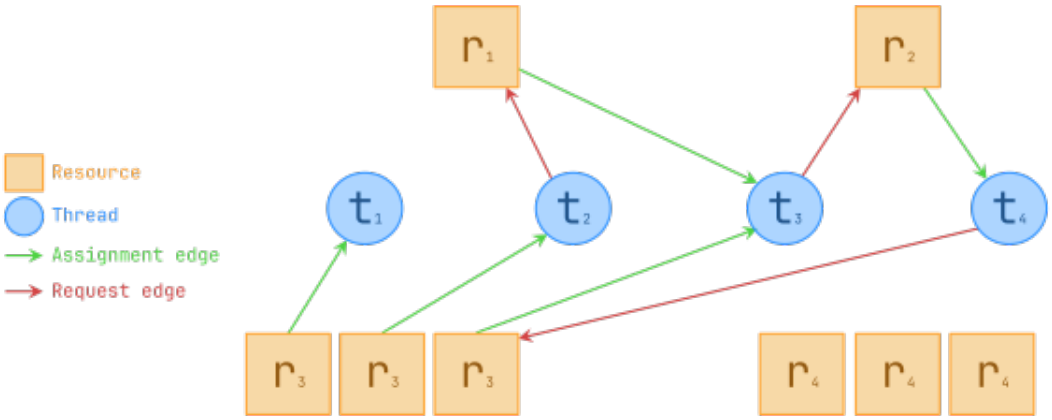



In the linear RAG, each resource is assigned for a thread, which is in turn waiting for another resource. Since $t_4$ isn't waiting for any resource, whenever it'll finish to execute its code it will release $r_2$, which will then be acquired by $t_3$ once it'll have done with $r_1$, and so on and so forth...

In the cyclic RAG, every thread is waiting for the acquisition of a resource, and won't be able to continue unless one of the threads releases one of the resources. Mind that this doesn't mean that there is a deadlock, but that it **might** exist at a given point in time. This is because we are assuming that there is only one copy of a resource in the whole machine. What if there were multiple copies of some resources?




Even in this case we still have some cyclic paths, because even if there are 2 copies of $r_3$, they still are connected cyclically.




In this last case we are avoiding the possibility of a deadlock, and that's because $r_3$ is assigned to $t_1$, which doesn't form a cyclic path. This means that as soon as $t_1$ finishes to execute, then $r_3$ could be given to $t_4$, which once its execution, it would release $r_2$, and so on... In general, if **any** resource involved in a cycle is assigned to a thread which is not in the cycle, then progress can be made and no deadlocks can occur.

RAGs are very useful to detect deadlocks, but what can we do once we detect a deadlock? There are several ways of dealing with them:

- to **kill all the threads** in a cycle: this is a quite harsh solution though, and should be employed only in rare cases (if not avoided at all);
- to **kill** all the **threads one at a time**, forcing the releasing of all the held resources;
- to **preempt the resources**, one at a time, rolling back to a consistent status (used for instance when doing database transactions).

Detecting if a cycle is formed on a directed graph is costly though, since the CPU would have to go through all the nodes. The RAG-scanning algorithms are based on the **D**epth-**F**irst **S**earch algorithm (**DFS**), and usually take $O(|V|+|E|)$ time. Usually though, when the graph is particularly dense, $O(|V|+|E|) \simeq O(|V|^2)$, where $|V|$ is the number of threads plus the number of resources. This is a particularly high timing, so we should be careful regarding *when* we should run the algorithm. It should be run only on the following occasions:

- **Before granting a resource**: each granted request will take $O(|V|^2)$;
- **When a request can't be fulfilled**: each non-fulfilled request will take $O(|V|^2)$;
- **On a regular schedule or when the CPU is under-utilized**.

### 3.4.2 Deadlock Prevention and Avoidance

We previously saw all the conditions that determine, if all simultaneously true, a deadlock. In order to prevent deadlocks, at least one of the following conditions has to be broken:

- **mutual exclusion**: all the resources should be made shareable. It's hard although to do it because of some resources like disks and I/O devices;
- **hold and wait**: a thread can't hold a resource while it requests for another, so a solution would be to enforce threads to make requests all at once. The problem is that we can't know in advance which resources will be asked by a thread;
- **no preemption**: if a thread requests for a resource that can't obtain, the OS releases all the resources held by the thread. It's not always easy to preempt the resources though;
- **circular wait**: impose an ordering on the resources and enforce threads to request them in a certain order. It's hard though to establish an order.

In the same way as we have methods to prevent deadlocks from happening, we can also avoid them. A method used to avoid deadlocks is to implement a **resource reservation** system: each thread has to provide the maximum number of resources that it might need during execution. If we consider $m_i$ as the **maximum** resources that a thread $i$ might request, $c_i$ as the number of resources that thread $i$ is **currently holding**, $C$ as the number of total resources **currently allocated** (which is equal to a sum like $\sum_{i=1}^{n} c_i$) and $R$ as the number of resources **overall available**, then we can say that a thread is **safe** if

$$\underbrace{m_i - c_i}_{\text{resources } t_i \text{ might still request}} \leq \underbrace{R - C}_{\text{resources currently available}} + \underbrace{\sum_{j=1}^{i-1} c_j}_{\text{resources currently allocated up to } t_j,\, j<i}$$

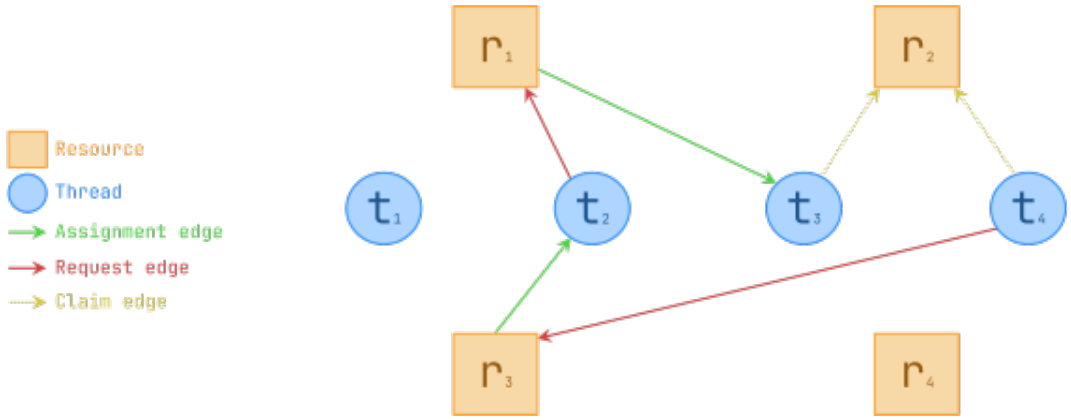
What do we mean by **safe state**? It's simply a sequence where threads are safe to run with the knowledge that deadlocks won't occur. An unsafe state doesn't necessarily mean that there is a deadlock, but that it might happen. For this new state, resources will be allocated only if the safe state is granted, otherwise they won't be allocated and the thread will wait until the safe state is granted. This avoids circular-waits conditions. Let us examine an example:

Suppose that there are 12 tape drives, and 3 threads ($t_1$, $t_2$ and $t_3$) that are requesting each a different number of tape drives. Suppose that 11 tape drives were already allocated as per the following table and 1 tape drive hasn't been assigned yet:

| **Thread** | $m_i$ | $c_i$ | $m_i - c_i$ |
|---|---|---|---|
| $t_1$ | 4 | 3 | 1 |
| $t_2$ | 8 | 4 | 4 |
| $t_3$ | 12 | 4 | 8 |

This is a **safe state**: the unallocated tape drive can be assigned to thread $t_1$, which, once it will complete its execution, will free 4 tape drives, allowing thread $t_2$ to use $t_1$ unused tape drives, and so on until also $t_3$ ends executing. If for instance $t_3$ started with $c_3 = 5$, then there would be no available drives, and a deadlock might happen.

We can update our RAG by adding another type of edge: the **claim edge**. Such edge is a directed edge $(t_i, r_j)$ which indicates that thread $t_i$ might request $r_j$ in the future. To satisfy a request means to convert a claim edge into an assignment edge. This way, a cycle in an extended RAG would indicate an unsafe state, and it can be easily fixed: if the allocation of a resource results in an unsafe state, then the allocation would be denied even if the resource is available. The claim edge would then become a **request** edge, and the thread will simply wait until there is the certainty that no cyclic path will be formed. Note that this **won't work** with **multiple instances** of the **same resource**.




What happens when $t_4$ and $t_2$ try to claim $r_2$?

- if $t_4$ acquired $r_2$: a cycle would form, because then $t_3$ would still try to acquire $r_2$, resulting in a deadlock. Because of this, $r_2$ **can't be acquired** by $t_4$;
- if $t_3$ acquired $r_2$: no cycles would form. Since we would remain in a safe state, we can grant $t_3$'s request.

With this new extended RAG, a request can be granted if and only if we move from a safe state to another safe state. There is an algorithm that is widely used regarding the granting of threads' requests, and which uses the concepts that we just mentioned: **Banker's algorithm**.

Such algorithm can handle multiple instances of the same resource, and **forces** threads to **provide information** on which resources they might need in **advance**. A constraint on the number of resources that can be request is that of course they shouldn't exceed the total number of resources. A request can be granted only if the system moves from a safe state to another, otherwise the thread waits. Let's define some variables for better explaining the algorithm and for giving an example:

- $n$: number of threads;
- $m$: number of resource types;
- `available[`$j$`] = `$k$ (with $j \in [1, m]$): $m$-dimensional vector
  - means that there are $k$ resources of type $j$ available;
- `max[`$i$`, `$j$`] = `$k$ (with $i \in [1, n]$ and $j \in [1, m]$): $n \times m$ matrix
  - means that thread $i$ may require at most $k$ resources of type $j$;
- `allocation[`$i$`, `$j$`] = `$k$ (with $i \in [1, n]$ and $j \in [1, m]$): $n \times m$ matrix
  - means that thread $i$ has allocated $k$ resources of type $j$;
- `need[`$i$`, `$j$`] = `$k$ (with $i \in [1, n]$ and $j \in [1, m]$): $n \times m$ matrix
  - equals to `max[`$i$`, `$j$`] - allocation[`$i$`, `$j$`]` $= k$, and it means that thread $i$ might need $k$ more resources of type $j$ to complete its task.

The algorithm is divided into two main tasks:

- `isSafeState()`: given the current state of allocation of resources, the algorithm checks if the state is safe;
- `resourceRequest()`: given a thread and its resource request, check whether the request can be granted or not.

Since a request can be granted only if the system goes from a safe state to another, it's clear how in order to perform `resourceRequest()`, the algorithm needs to use the output of `isSafeState()`. Here is a rough pseudocode implementation of the algorithm's first part, `isSafeState()`:

1) Let `work` and `finish` be two vectors of length $m$ and $n$ respectively; then initialize them such that

```
for i in allThreads {
    work = available;
    finish[i] = false;
}
```

2) Find an $i$ such that

```
finish[i] == false && need[i] <= work
```

If no such $i$ exists, then go to step 4;

3) Assume that thread $i$ executes the following:

```
work = work + allocation[i];
finish[i] = true;
// go to step 2
```

4) If the following statement is true then the system is considered in a **safe state**:

```
for i in allThreads {
    finish[i] == true;
}
```

Now, here is the implementation of `requestResource()`:

In → take $i$ (a thread) and an $m$-dimensional vector of requests;

1) If

```
request > need[i]
```

then raise an error, since thread $i$ is attempting to request more resources than the claimed ones. If such condition is false, go to step 2;

2) If

```
request > available
```

then thread $i$ must wait since the resource is not available. If such condition is false, go to step 3;

3) Test if the allocation leads to a safe state and, if it does, grant the request:

```
available = available - request;
allocation[i] = allocation[i] + request;
need[i] = need[i] - request;

// test if the state will be safe
if (isSafeState() == true) {
    grantRequest();
} else {
    rollback();
    i.wait();
}
```

# Chapter 4
## Memory Management

We saw how a computer now is made out of multiple things: a CPU (with one or multiple cores), the memory and the I/O. But how can we manage the memory in an efficient way, in order to **serve multiple processes**? We have to account also for the fact that each process has its **own isolated memory space**, but at the same time we have to grant to both computer and programmer the **illusion** that **memory is unlimited**.

In some programming languages, user programs refer to data and to instructions with some characters, such as "+", "&&", "x", "count", and so on... For an instruction, in order to be executed, it first needs to be **translated** from source code to a binary executable and then saved to the disk, then they must be **loaded** into the main memory (RAM) and then finally get executed.

The translation of a source code into an executable binary code is done via a **compiler**, an **assembler** or a **linker**, while the loading of the program is instead done via the **loader**. In case of some purely-interpreted languages (such as the most common distribution of Python, `CPython`), the translation from source code to binary executable is done on the fly.

The frequency on which a CPU accesses the memory is actually very high: the CPU continuously **fetches**, **decodes** and **executes** instructions from the memory while it executes a program. The CPU, whenever it has to retrieve data from the memory, uses fixed **memory addresses**. When using links in the source code, they don't actually represent a fixed location in memory, and they have to be defined eventually. We have 3 steps that we take when translating from a reference to a physical address:

- **symbolic name**: it's the symbolic reference that we use on the source code. For instance, a variable. If we had a code like

  ```
  let x: i32 = 54;
  let y: String = String::from("Hello world");
  ```

  then both `x` and `y` would be two symbolic references;

- **logical address**: it's the memory address generated by the user programs via the CPU;

- **physical address**: it's the actual memory address on the physical memory.

There are 3 ways that define the way a **logical address** is then mapped to a **physical address**: on **compile time**, on **load time** or on **execution time**.

- **Compile time**: with this method, the starting physical location of a program in the memory is $k = 0$. Whenever the compiler gets called, it starts generating the **absolute code**: for such code, there is no intervention by the OS, and the **logical address** calculated by the CPU is the **same as** the **physical address**. The downside of this approach is that if the program gets moved for any reason in the memory from an address to another, then the program must be recompiled from scratch, because the memory addresses changed;

- **Load time**: if the starting physical location $k$ is **not known**, then the compiler will generate **statically relocatable code**; such code will reference to $k$ and be replaced then with relative addresses. The **addresses** that will be then replaced are **determined** by the OS **loader**. In this case as well, the **physical address** would be **equal** to the **logical address**. The downside here is that if the program gets moved into the memory, then it just needs to be reloaded, and not recompiled;

- **Execution time**: if the program can be moved in main memory during its execution, then we determine the address at execution time. The compiler will generate for the process some **dynamically relocatable code** or **virtual addresses**. The way a virtual address is mapped to the physical memory is by using a little hardware unit, called **MMU**. In this case though, the logical address will be different from the physical address. This is the solution that most OSs nowadays implement, because it gives flexibility to the system. Whenever the CPU runs the code, all the **logical addresses** that the CPU references will be then **translated** at **run time** in physical addresses.

When dealing with **uniprogramming systems**, it's pretty easy to manage the memory: the OS keeps a part of the memory for itself and processes are allocated one by one, starting from an address $k = 0$ up to the maximum address. Each process moreover executes on a contiguous segment of memory, and the **address binding** happens at **compile time**. Again, this is a fairly simple way of allocating memory, but it allows for **only one process at a time** to execute, and there is **no OS protection**.

## 4.1 Multiprogramming Memory and Relocation

There are some goals to achieve whenever we have to share a memory between multiple processes. One of them is **sharing**: several processes must coexist in the memory at the same time, and cooperating processes must be able to share portions of the address space. Another goal that must be achieved is **transparency**: processes should not be aware that memory is shared, and they should also not be aware of which portion of memory is given to them. We also must ensure **protection and security**: processes should not corrupt each other or the OS, and should also not read other processes' data. Finally, we must ensure **efficiency**: the CPU and the memory performance should not degrade because of this system, and we should avoid **fragmentation** as much as possible.

In order to manage multiple processes in the memory, we want to relocate sometimes the processes, in order to better manage the available space. Let's assume for instance that the OS gets the highest memory addresses (such as in MS-DOS), and the rest of the space is given to the user for the processes. We would have the following quantity of space:

$$\texttt{memory\_for\_processes} = \texttt{memory\_size} - \texttt{os\_size} - 1$$

What happens when we have to load a process? We reserve the first contiguous segment of memory in which the process fits. We must allow a transparent sharing of memory, since each process' memory space may be placed **anywhere** in the main memory.

How can we relocate processes in the main memory then? A first method is called **static relocation**: the OS loader, whenever a process must be copied to the main memory, it rewrites the addresses generated by a process with the main memory addresses (having thus **load time binding**). The good side of such approach is that no hardware support is needed, but there are plenty of downsides: first, there is **no protection** or **privacy** regarding the OS: a process could potentially corrupt the OS or other processes, since there is no check regarding if a memory segment is actually free or not; second, the address space is allocated **contiguously**: if a process' stack or heap exceed in dimensions, then they could corrupt other processes; thirdly, the OS **cannot move a process**, since it uses load time binding, which uses fixed logical addresses.

Another relocation technique is called **dynamic relocation**: it allows for **OS** and **processes protection**, but it requires **hardware support** (more specifically, it requires the **M**emory **M**anagement **U**nit (**MMU**)): this means that address binding happens at **run time**. The MMU translates each logical or virtual address generated by a process with a physical address.

### 4.1.1 Memory Management Unit (MMU)

The MMU is a little circuit that has the task to translate logical/virtual addresses into physical addresses at run time. It's made by at least two **registers**: **base**, which indicates the start of the address space on the physical memory with a physical address, and **limit**, which is the size limit of the address space. The CPU can change the content of these registers only in kernel mode.

How does the MMU work in practice? Whenever a process must load, a **contiguous segment** of the main memory is **associated** to such process, and such process can **access only** to its **assigned memory segment**. The **protection** is implemented by using the two MMU registers: **base** and **limit**.

When in user mode (so while the process is running), the CPU checks that every access from the process to the main memory is within its correct range, which is `[base, base + limit)`. If the CPU isn't in user mode, then it has free access to the memory.

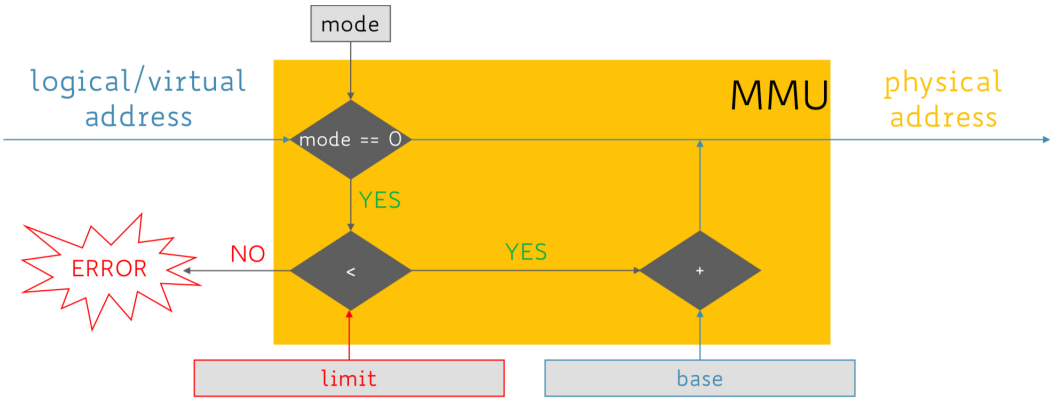



**Figure 4.1:** Logic behind the MMU

This approach has various good sides: it provides **OS** and **processes protection**, be it in read or write mode whenever the CPU is in user mode; the **OS** can easily **move the process** in the main memory if needed, since it just needs to adjust the values of `base` and `limit`; the process' **address space** can also easily **grow** if needed, and the OS can do that easily; it's a **simple** and **fast implementation** which requires a **few hardware pieces**: it just needs two special registers and an `add` and `compare` operations (they can also be done in parallel).

There are also some downsides to this approach: there is a **little hardware overhead** that has to be paid at each memory access, and still, processes must be **contiguously allocated** in the main memory. The process is, moreover, still **limited** to the **physical memory size**, and the degree of **multiprogramming** is **bound** to the size of the **physical memory**. Lastly, processes **can't share** part of their **address space** (for instance their program's text section).

Relocation must follow the following 3 properties:

- **Sharing/Transparency**: processes shouldn't be aware that they are sharing memory, that's something that only the programmer and the OS are aware of;
- **Protection/Security**: each memory reference is checked, be it a read or write reference. If the reference isn't valid, then the access to the memory gets denied;
- **Efficiency**: it's somewhat achieved by both the methods, but it becomes very inefficient when the OS has to move a process.

**Static relocation** satisfies **only** the **first** property, while **dynamic relocation** satisfies **both** the **first** and the **second** property.

## 4.2 Memory management problems

Handling memory isn't easy, especially with multiprogramming systems where there is a constant moving of data between the main memory and the disk. Let us see some of the most important problems and some solutions.

### 4.2.1 Allocation

With both static and dynamic relocation we had a problem: the processes were allocated contiguously. Back in the days, a simple allocation method was used: the physical memory available for the user-processes was divided into multiple sections, all **equally-sized**; upon the creation of a process, one of such segments would be given to a process, where it would allocate. Such method though implicitly poses a great **restriction** on the **multiprogramming degree**. Nowadays, this method isn't used anymore.

Another approach is the one of **keeping track**, for each terminated process, the **space** in memory that became free. Once another process is created, part of the now free space will be given to it. The unused segments of memory that were created in this way are called **holes**, and the OS keeps track of each hole. How can the OS know in which hole a process should be allocated though? There are many policies:

- **First-fit**: the OS first linearly scans the list of holes until a hole is found that is big enough for the new process. If another process makes a request for some space, the OS can either restart from the beginning of the list or continue from where it left. It takes, for each request $O(n)$, where $n$ is the number of holes in the list. This approach has a flaw though: if for instance the holes list is as follows:

  `holes[] = [20B → 400B → 120B]`

  then the OS, if it had to allocate a process of 100B, it would allocate it on the 400B hole, thus creating another hole of 300B. What happens though if another process, which is 350B wide, makes a request to be put into the main memory? The OS wouldn't be able to satisfy such request;

- **Best-fit**: the OS linearly scans the list of holes to find the smallest hole that would fit the process. This can save large holes for other process requests that may need them. This would continuously create smaller and smaller holes, that would probably be unusable because of their small nature. The OS would take either $O(n)$ time, if the list was unsorted, or $O(\log(n))$, if the list was sorted. The holes could be ordered in a **B**inary **S**earch **T**ree (**BST**). Let's consider our previous example: in that case, if we had a process of size 100B, it would be allocated into the 120B hole, creating thus a 20B hole, and the incoming 350B process would be placed into the 400B hole;

- **Worst-fit**: for this method the OS allocates the largest hole available. It seems counter intuitive, but the reasoning behind this method is that much likely the created hole will be usable for future requests.

Many simulations have shown that **first-fit** and **best-fit** are the most **efficient** methods, and surprisingly first-fit is generally **faster** than best-fit.

### 4.2.2 Fragmentation

Let us suppose that the main memory, after allocating various processes, becomes nearly full, leaving various small holes in between the processes. By themselves, such holes can't host a process, but if they were combined together they could eventually be used to store a process' data. Memory fragmentation can be of two types:

- **External fragmentation**: the frequent loading and unloading of processes' data from the main memory creates small, unusable chunks. External fragmentation usually happens when there is enough memory to load a process, but the space isn't allocated contiguously. Benchmarks over time have shown that for every $2N$ allocated blocks, $N$ are lost due to external fragmentation: in other words, on average, ⅓ of memory space is wasted due to external fragmentation. In order to reduce external fragmentation, it's necessary to adopt an allocation policy that minimizes the wasted space.

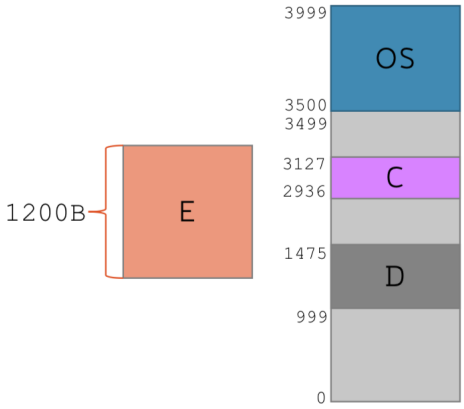



**Figure 4.2:** Example of process not fitting in main memory due to external fragmentation

- **Internal fragmentation**: it happens when the memory inside a segment is wasted. For instance, if we had a process whose size was 8.464B and a segment of 8.466B, instead of creating a 2B segment, we could give all the free segment to the process; if we decided instead to create a hole, we would have to keep track of a 2B hole, which just makes the OS waste time whenever it has to look for a fitting segment.

There is a solution to memory fragmentation, and it's called **full compaction**: the OS tries to compact the process, moving them in the memory and placing them contiguously: this way, all the holes get basically unified into a single hole, and more processes can be fit inside the hole. This solution is costly though: all the processes in the memory have to be moved in order to be compacted.

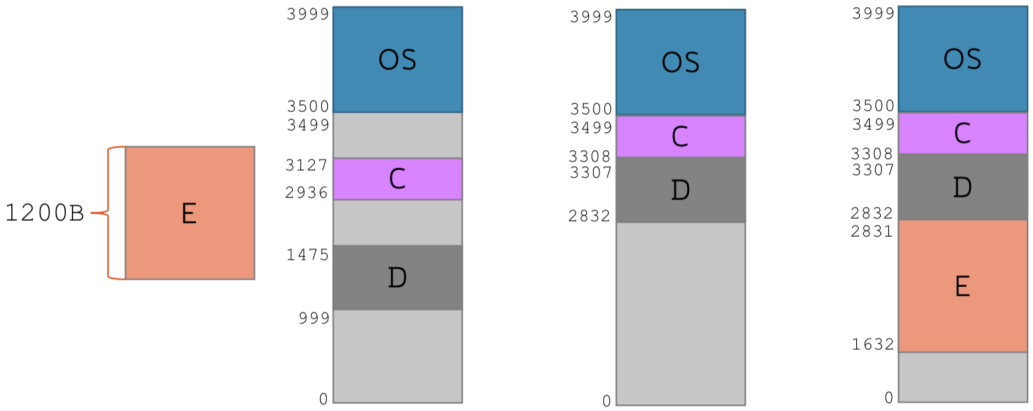



**Figure 4.3:** Full compaction with three processes $C$, $D$ and $E$

Another approach is the one of **partial compaction**, where instead of moving all the processes, we only move the necessary processes that would allow to expand the most similar sized hole with respect to the size of the new process. This doesn't solve the fragmentation problem fully, but it saves the CPU from making unnecessary computations when they are not needed.

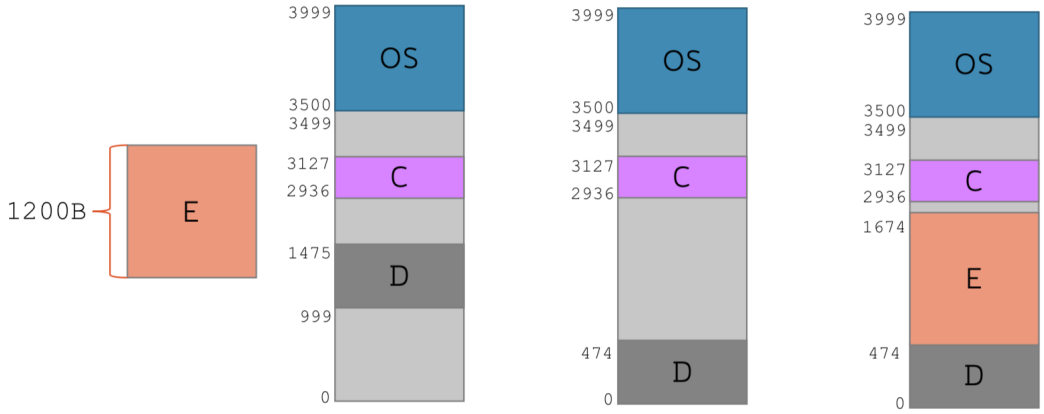



**Figure 4.4:** Partial compaction with three processes $C$, $D$ and $E$

### 4.2.3 Swapping

So far we always assumed that all the processes, when loaded, were **entirely** loaded into the main memory: it's crucial that the process' data, whenever they need to be read by the CPU, reside in the **main memory** rather than in the disk. This means that whenever a process requires an I/O operation, it's **not necessary** that it remains inside the main memory: it could be **swapped out** from the memory **to the disk**, freeing some space for other processes, and, whenever the process is ready again to execute, it can be reloaded again by the OS into the main memory.

Depending on the type of address binding that is used, the swap changes:

- if the binding is done in either **compile** or **load time**, then the process must be reloaded in the **same memory location** where it was before being swapped;
- if the binding is done during **run time**, then it can be swapped in **any free space** inside the main memory: it will just need to update the values of base and limit registers accordingly.

With swapping (in particular if the process' binding is done at run time), fragmentation can be tackled easily: it just needs the OS to compact the main memory before swapping in again a process. Although it helps with the memory management, swapping is a very **slow process**, because it has to rely on the disk, which is a particularly slow I/O operation: this is the reason why nowadays swapping isn't a method anymore used by the majority of OSs, and more advanced techniques such as **paging** are preferred. The following example shows why:

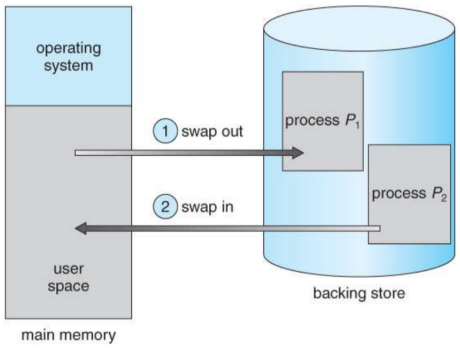



**Figure 4.5:** Visual representation of the swapping process

**Example 4.2.1**

Let us consider that each user process takes 10MB of memory, and that the disk transfer rate is 40MB/$s$. In order to swap in or out a single process, 250$ms$ are

needed. Considering that a swap in and a swap out must be performed, the overall required time will be of around $500ms$. In comparison, a time slice is way smaller than $500ms$.

## 4.3 Paging

The three main problems when it comes to memory management are the ones listed above: **contiguous allocation** (it's hard to grow or shrink a process' memory segment), **fragmentation** (it requires frequent compaction from the OS) and **process loading** (in order to entirely load a process, swapping could be used, but it's extremely inefficient). There is a memory management scheme which is able to address all the problems listed above, and it's called **paging**.

For this technique, the logical address space of a process is still **contiguous**, but it's divided into fixed-size blocks, which are called **pages**. The contiguous allocation of a process isn't anymore necessary, and logical pages can be mapped to non-contiguous frames. This way, there isn't any external fragmentation (although internal fragmentation could still occur).

There is a rule (which is an observation based on several benchmarks) called the **90/10 rule**: for this rule, processes spend the 90% of their time **accessing** only 10% of their **allocated memory space**.

How is a process allocated into a pages-divided memory? First, the process is divided into multiple pages of the same size of the pages in the main memory (which usually is 4KB) and then each page is then mapped into a free frame in the main memory. The pages can be stored also in a non-contiguous way. The OS has then 2 main responsibilities: to **map** all the **correspondences** between the **process' pages** and the **memory's frames** and to **translate** the **process' logical addresses** with the **physical addresses**. Since memory is accessed very frequently, this system needs to be efficient and fast. In order to store all the mappings between the processes' pages and the memory's frames, the OS uses a table called **page table**, which is a **lookup table** that indicates in which memory frame a page is stored. It's not always the case that all the pages of a process are mapped to the physical frames: in most of the times only a percentage of a process' pages are loaded in the main memory.

Whenever a process must refer to the physical memory for any purpose, it refers via the virtual address, and **not** with the page number. In contrast to the physical address space, the virtual address space is contiguous, and starts from 0. The page table must translate each virtual address with a physical address, but how does a translation work?

The virtual address consists of two parts:

- `p`: the **page number** where the address resides;
- `offset`: an **offset** which is relative from the beginning of the page.

Whenever the memory is referenced, the CPU takes the part `p` from the virtual address and uses it to determine on which physical frame `f` the referenced content is. Once the

frame has been determined, the virtual offset is also applied to the frame number, thus creating the physical address, which is equal to the union of `f` and `offset`. Once the physical address has been composed, it is then used to access into the physical memory.

Paging is an advanced form of **dynamic relocation**, since it translates the virtual addresses into physical addresses. The page table is logically similar to a set of `base` registers. The mapping between a virtual page and a physical frame is invisible to the user processes, since the page table is maintained by the OS, and the translation happens on the hardware (specifically on the MMU). The protection is provided similarly to the dynamic relocation system (so with the use of a `limit` register).

More in practice, how does the MMU use the page table to translate the virtual addresses? We can explain it with an example: let us suppose that the available physical address space for user processes has a size of 50B. We assume that a page is 10B wide (`S`), and each process can generate virtual addresses in the range [0, 49]. Let us suppose that a process generates a virtual address `x = 47`, then the MMU would translate the address in this way (consider it as a `RISC-V ASM` pseudocode):

```
.data
x:  .word 47
s:  .word 10

.text
    p = x div S           # p = 47 div 10 = 4
    offset = x mod S      # offset = 47 mod 10 = 7

    # retrieve the physical address from
    # the page table and apply the offset
```

The page/frame numbers and sizes depend on the architecture of the CPU; the size typically is a **power of 2**, ranging between 512B and 8192B (so around 8KiB). Powers of 2 are convenient because the translation from a virtual to a physical address is easy and doesn't need any division or modulo operation: if we have a virtual address made by $m$ bits, then the virtual address space would range between $[0, 2^m - 1]$; if the page size is $2^n$, with $n \leq m$, then the higher $m - n$ bits represent the **page number**, while the low order $n$ bits represent the **offset**.

The typical values of the virtual address' size $m$ are either **32** or **64 bits** (having thus the virtual address space ranging either $2^{32}$ (4GiB) or $2^{64}$ (16EiB)), while the typical value of a page/frame size $n$ is $2^{12}$ (which is, as we said earlier, **4KiB**). Assuming then $m = 32$, we see that there are $2^{m-n} = 2^{20} = \sim 1M$ pages/frames. As a consequence, the page table has roughly $1M$ entries, one for each page.

### 4.3.1 Translation Look-aside Buffer (TLB)

Whenever a user process references a virtual address, we saw that the MMU steps in to translate the virtual address into a physical one. In order to do so, the MMU must **access** the page table. But where should the page table be **placed**? It **can't be placed** in some **register set**: they are fast, but they also are too much expensive; they **can't be placed** in the **main memory** either, since it would require one extra memory access. There is a **cache** called **T**ranslation **L**ook-aside **B**uffer (**TLB**) which contains the page table.

> In a computer, all the memory accesses are equivalent, it doesn't know the purpose of each memory part. The CPU for instance can only directly access only its **registers** and the **main memory**, since for any other device it needs to perform an I/O operation. Accessing a register usually takes one clock cycle, and is thus very fast, while accessing the main memory usually takes several clock cycles and is thus slower.
>
> **Caches** are a type of memory that bridges the gap between fast and slow memory. A **cache memory** is an **on-chip** intermediary memory that is built into most of the modern CPUs. Whenever data must be transferred from the main memory to the CPU, usually such data are also moved to the cache. Since accessing the cache is faster than the main memory, the scope of caches is to let the CPU access to the main memory's data through them.
>
> As we said earlier, the TLB is a very fast L1 cache: it allows to store in a fully-associative memory the mappings between page numbers (keys) and frame numbers (values). Usually memory accesses follow the **locality principle**, which states that memory references are often **close** to each other. Locality does usually hold also for address translation. A TLB cache size usually ranges from the 8 to the 2048 entries.

So, let's trace back the steps taken when a virtual address is mentioned: the virtual address is separated into two parts, `p` and `offset`; `p` is used as key into the TLB and the `offset` is then summed to the value obtained from the TLB. The summed address is then used to retrieve the data from the memory, since such address is the physical address. But what happens if the TLB doesn't have the value needed? The TLB is very small compared to the number of entries of the actual page table (we said that with a 32-bits architecture there would be around 1 million pages).

The TLB stores only some values of the page table, which resides in the main memory. Pages are loaded with a **L**east **R**ecently **U**sed (**LRU**) policy, because the most recent pages have an higher probability of being read by the process, rather than some other page. Whenever the TLB doesn't have the frame number of the requested page, we say that a **TLB miss** occurs: whenever that happens, the system must recover from the memory the page and frame number tuple, but it must also update the TLB. The data from the page table in the memory are recovered in the following way: there is a register called **P**age **T**able **B**ase **R**egister (**PTBR**) which contains the base address from where the page table starts in the main memory. Once the frame has been recovered from the table, it is not only used with the `offset` to determine the physical address, but it is also used to update the TLB.

Now, this system would work under a uni-programming situation, but would it work under a **multiprogramming** setting? In such setting, the TLB must change continuously, depending on the running process: each process does use similar virtual addresses, but for each of them the physical address changes radically. The TLB must be them **up to date** with respect to the **running process**.

There are two setups that allow the TLB to work under a multiprogramming scenery:

- **basic**: at each context switch the TLB is fully flushed and cleaned. As a result, the

first accesses will generate TLB misses, since the TLB will be empty;

- **advanced**: the TLB entries could either be dumped into the PCB or or they could be each related with a single **P**rocess **C**ontext **ID** (**PCID**). The CPU will use an entry of the TLB if and only if the PCID is the same of the running process.

Let us denote the following elements:

- $t_{MA}$: the physical memory access time;
- $t_{TLB}$: the lookup time on the TLB cache. Note that $t_{TLB} \ll t_{MA}$;
- $p$: the probability of a TLB cache hit (also called hit ratio);
- $T_{MA}$: total time required to actually get to the physical memory each time a virtual address is referenced.

Depending on the configuration, we have the following timings:

| Situation | Timing |
|---|---|
| Without TLB | $T_{MA} = 2 \cdot t_{MA}$ |
| With TLB | $T_{MA} = p \cdot \underbrace{(t_{MA} + t_{TLB})}_{\text{TLB hit}} + (1-p) \cdot \underbrace{(2 \cdot t_{MA} + t_{TLB})}_{\text{TLB miss}}$ |

Usually, the larger the TLB is, the higher the hit ratio.

The page table can also be set up in order to allow further protection layers: for instance, it could prevent unauthorized processes to access certain memory sections that do not belong to them. A bit (or some bits) can be added in order to mark a page as `read-write`, `read-only`, `read-write-execute` or a combination of those. These security steps are ensured upon each memory reference.

Processes could also mark with a "`valid/invalid`" bit the pages that are not used anymore by the process itself: this is useful for the OS because not all the page table entries are used by the processes, and this way some clean-up operations can be done. Basically, whenever a new entry will be added, the OS will put the new entry either in an empty slot or where an invalid entry is. We said that a PTBR exists, but some tables also use a **P**age **T**able **L**ength **R**egister (**PTLR**) that specifies the length of the page table.

So how does the OS proceed whenever a new process requests to load some pages in the main memory?

1) the process requests $k$ pages to be loaded in the main memory: if there are $k$ free frames in the main memory, then the OS allocates the pages in the free frames, otherwise it proceeds to scan for any invalid page;
2) once the request is granted, the OS allocates the pages in the various frames;
3) the OS proceeds to update the page table in the main memory and updates the TLB. The TLB-update process consists in either marking all the previous TLB entries as `invalid` or restoring the TLB entries from a saved PCB.

4) as the process runs, the OS loads any missed TLB entry by either finding a free spot in the TLB or by replacing older entries.

This has some implications on the PCB, which now must contain the value of the PTBR and, possible, a copy of the TLB entries. On a context switch, the OS must copy into the PCB these two values, flush the TLB and restore the PTBR and TLB entries from another PCB.

Paging makes **block sharing** easy, since now there isn't anymore the need to allocate the blocks contiguously. All the OS needs to do is to duplicate the TLB entries. Mind that this is only possible whenever the code is **reentrant**, so if it can stop and resume safely: in other words, the code **can't write** or change **itself**. The reentrant code can be shared by multiple processes, but the important part is that each process has its own copy of their data and their registers, including the PC.

### 4.3.2 Segmentation

Most of the times, a lot of programmers do not consider the memory as a linear and contiguous storage of their programs, but rather they consider it as a **segmented container**, where each sub-container (such as the code, the data, the stack, the heap, etc...) is devoted to a specific function. **Memory segmentation** allows to consider the memory as a container or containers, by providing each **address** with a **segment number** (which refers in turn to a segment base address) and an **offset**.

For instance, let us consider the following example: a compiler generates from a code 5 segments, one for each part of the memory (the user code, the library code, the global variables, the stack and the heap). Now, the compiler will generate all the addresses identifying both the segments and the offsets and will store the correspondence between a **segment-offset address** and a **physical address** in a table called **segment table**. The segments are **stored** via the use of `base` and `limit` addresses, and each entry of the table can have some **additional protecting information** (such as sharing or read/write permissions). The table needs just a few `base` and `limit` hardware registers, something that doesn't happen with the page table, since it usually is way more capable than the segment table; a segment table usually stores a very limited amount of segments per process (from the 3 to the 5 segments).

A segmentation address of $m$ bits is made in the following way: the $n$ top-most significant bits indicate the **segment number**, while the $m - n$ bits indicate the **offset**. Segmentation can be **combined** with both **static** or **dynamic relocation**, although **external fragmentation** can happen again, since the segments are allocated **contiguously**. In the case where the number of logical segments per process increases, a cache similar to the TLB might be necessary, since with only registers it would cost too much.

The optimal solution would take the best of both the approaches between segmentation and paging: segmentation's main feature is how **easy** it is **to share data** between processes, while paging is very **efficient** for what concerns the **memory usage**. A solution is to apply, for instance, **paging to segments**. For such technique, each process' virtual space becomes a collection of segments of arbitrary size, while the physical address space remains a sequence of fixed-size frames; each segment gets divided into multiple page frames (since usually segments are larger than a page).

A logical address now becomes like the following:


The **segment number** $s$ is used only to **index** into the **segment table**, which returns the **base address** of the **page table** relative to the specified segment. Some checks are performed during the lookup to see if the address is within the allowed range. The **page number** $p$ is then used to **index** the **page table** and get the **physical frame number**; after retrieval, the **offset** is **added** to the frame number and the **physical address** has now been composed.

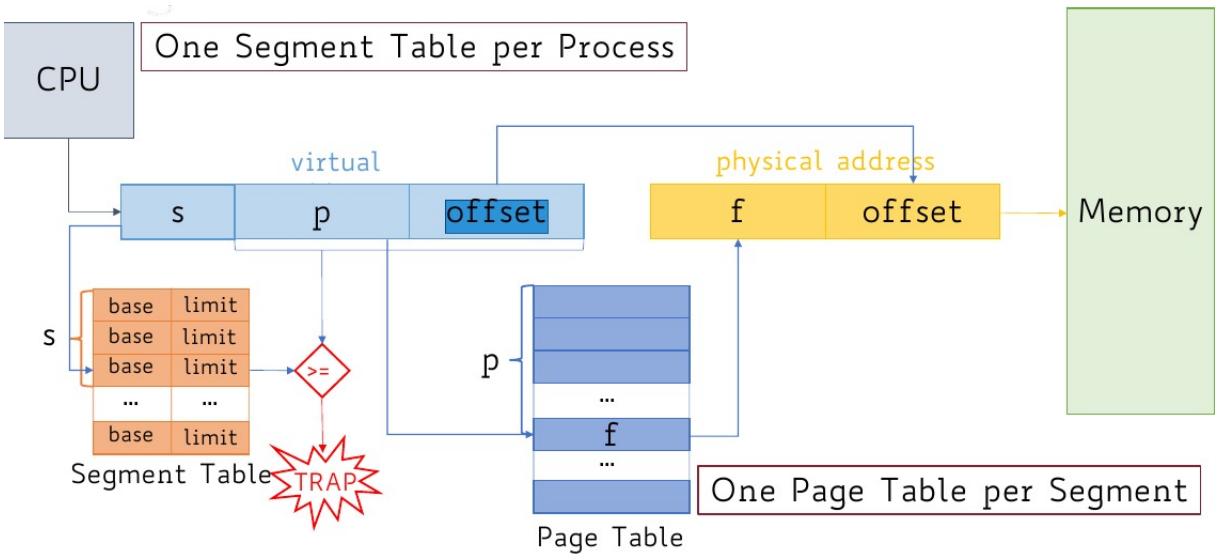



**Figure 4.6:** Address examination process

Since for each segment there is a page table, and since each process has its own segment table, where can we store them? There are two options:

1) we could store the segment tables in a small set of registers and store instead the page table in the main memory with the TLB cache: this option is **faster**, but it poses a **limit** to the **number of segments**;

2) we could store both the segment table and the page table of each process in the main memory with the TLB cache, and the TLB lookup could be done by using the segment index and the page index: this option is **slower** than the first one, but is **more flexible** since it allows for a larger set of segments.

Just like pages can be shared, also segments can be **shared** in the same way: the shared segment gets mapped to two different processes, and both the processes will then **access** their **own segment table** and then access to the **same segment's page table**. This allows for an even more flexible configuration.

The **benefits** to the segmented paging method is that the **compiler** and **OS view of memory** is now **merged**, it is more **flexible** and presents **no external fragmentation**, and the **sharing** between processes is possible. There are **two major counter sides**: the **context switches** and the **address translation processes** are now **slower** because they have to respectively change more data and make more operations.

What about internal fragmentation? It still is an issue with paging. On a pure paging setting with no segmentation, assuming a random process' memory footprint, the **average number** of **internal fragmentation pages** (so the sum of all the holes expressed in pages) is $0,5$. Of course, the larger the page size, the larger the chance of having internal fragmentation, but the smaller the page size, the larger the page tables will be. When designing the size of a page, a tradeoff must be done.

Nowadays computers support logical address spaces that range from $2^{32}$ to $2^{64}$ bits. Usually with a $2^{32}$ address space, the page size is 4KiB ($2^{12}$), so we can have up to $2^{20}$ (nearly 1 million) entries in the page table. If each entry takes 4B, then the full page table will amount to 4MiB, which is too large in order to be kept contiguously in the main memory. Since also page tables are paged, with 4KiB-wide pages we have $2^{10}$ (1024) pages just to hold the page table. Some other techniques must be used in order to keep everything in the main memory.

An idea is to have a two-tier page table, so we would have a page table where each entry corresponds to another page table. Always considering a 32-bits architecture, we would have, in an address, 20 bits for the **page number** and 12 bits for the **page offset**. The page number could be broke down into two page numbers of 10 bits each, where one page number refers to the "table of tables" and the second refers to the indexed table. The remaining 12 bits of the offset will still be used to navigate within the 4KiB frame.

Another possible way is to use **hash tables** to store highly sparse page tables, and thus **index** the **tables** via **hash functions**, rather than with indexes. Another method is to have **inverted page tables**: such tables list, instead of all the pages belonging to a process, all the pages loaded in the main memory for all the processes. The accessing process to a particular page takes linear time, although it could be sped up via the use of hash tables. Since each frame is mapped to exactly one process, it's not easy to allow memory sharing.

## 4.4 Virtual Memory

In practice, real processes do not always need all the pages loaded in memory, or at least not all at once: this is why **virtual memory** uses backing storage (like HDDs) to store the pages, which gives the illusion of having infinite memory. The ability to load only partial portions of memory has several benefits, such as the possibility to **write** much **larger programs**, which wouldn't fit in a physical memory, have **more memory** for the other processes and have **less I/O operations**.

So at any time, a page can be stored either in the **memory** (so in a physical frame) or on a **backing store** (such as a disk). Usually, virtual address spaces are highly **sparse**: a lot of virtual addresses usually remain **unreferenced**.

The idea of virtual memory is to use the **main memory** as a **cache** to store the location of the pages on the disk, and via the use of a **single bit** the pages are flagged depending whether they are **stored** on the disk or on the memory: this needs the **OS** to **update** the **table** each time that a page is moved from the memory to the disk and vice versa. Let's recall that **accessing the disk** is a **slower operation** rather than accessing the mem-

ory, and this means that we should access to the memory as much as possible, avoiding whenever we can to access to the disk.

For the 90/10 rule, the 90% of a process' memory accesses are on a 10% portion of the memory: such area is called **working set of the process**. Since the working set has a very limited space size, it will likely fit in memory, so we want it to be **always located** in the main memory. Of course, that's a statistic rule, so it may be that sometimes a process shifts its working set, but for the rest of the time, we want the most-needed working set to be always loaded in the main memory, and we'll see why it's better to avoid page faults.

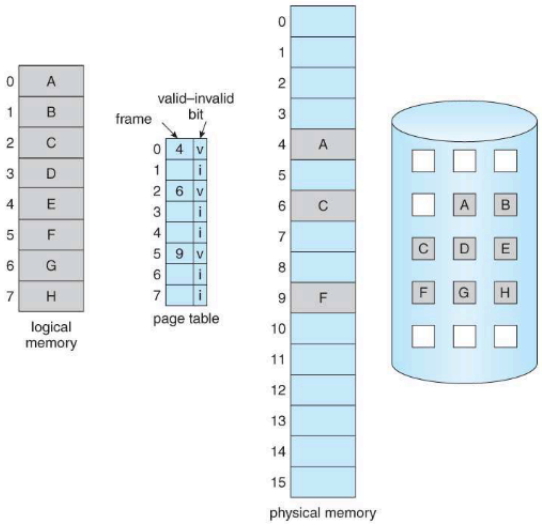



**Figure 4.7:** Big scale image regarding how Virtual Memory works

Virtual memory works in the following way: whenever a process has to access to a logical memory reference, a **page table lookup** is performed, to see where the page is located within the machine. A page table will be composed by two fields: a **frame** which contains the address in the physical memory and a **flag bit** which can be either **valid** or **invalid**. If the bit states **invalid**, then a **page fault trap** occurs, which will initiate the retrieval of the page from the disk. In UNIX-like systems, there is a partition in the disk called `swap` which contains a table with the position of the pages on the disk.

More in detail, the following happens whenever we try every time to reference to data in the memory:

1) the memory address is first checked, to see if it's valid: if the address is valid, then the data is retrieved from the memory, otherwise, a **page fault trap** occurs, and the page must be retrieved from the disk;

2) from a possible free-frames list, the OS **assigns a free frame** to the page that has to be retrieved (in the case that the memory is full, the OS will have to unload some memory first);

3) a **disk operation** is **scheduled** among all the I/O operations, in order to bring the page from the disk to the main memory. The currently executed **process will be blocked**, and some other process will be run instead;

4) whenever the I/O operation is completed, the OS will **update** the **page table** with the new frame address, and will set the validity bit to **valid**;

5) the process that was running instead of the original one will be **interrupted**, and the original process will have to **start again**.

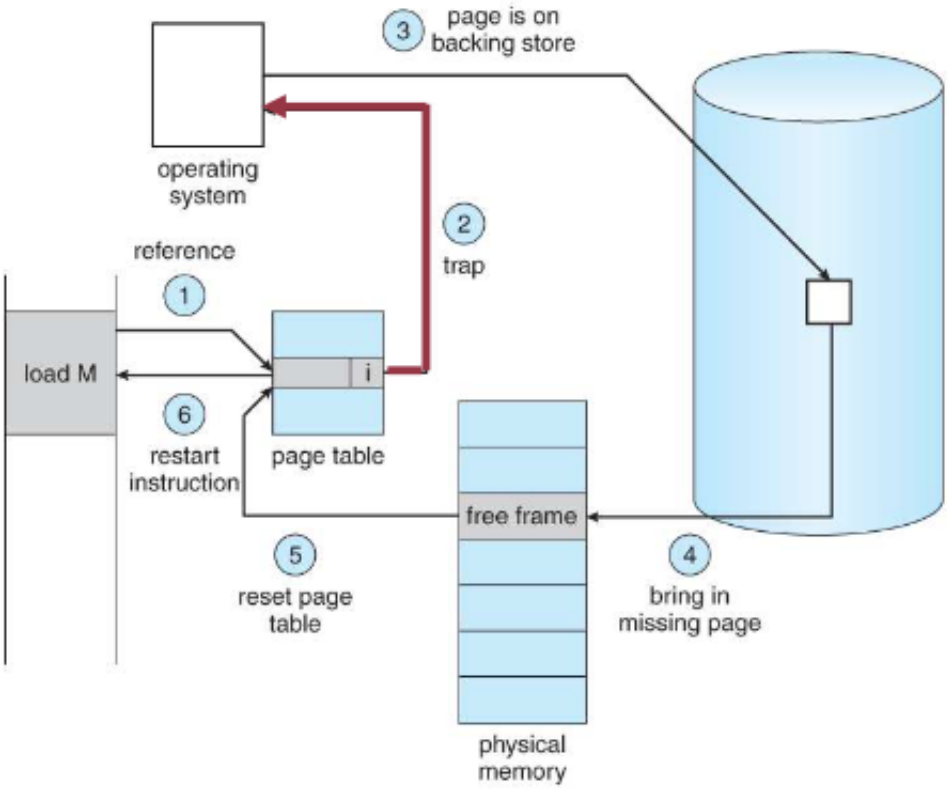



**Figure 4.8:** Visual explanation of the previous steps. Step 1 and 2 in the image correspond to step 1 in the notes

The valid bit in the page table is also used by the TLB to indicate whenever the page is in the memory or in the disk. If we have a **TLB hit**, it means that the **referenced page** is loaded in the **memory and** the **page entry** is loaded in the **cache**. What happens is there is a TLB miss then? It basically means that the page table in the TLB doesn't contain the entry the process is looking for, which means that both the page table and the page must be fetched from the disk. Supposing that the requested page is not in the cache but it's present in the memory, the OS will **replace** one entry in the TLB with the entry corresponding to the missing page; in the case instead where the page is not **neither** in the **cache nor** in the **main memory**, the OS will then **invalidate** one **TLB entry**, **perform** the **page fault trap operations**, **update** the **TLB entry** and then **restart** the faulting instruction.

How does the OS **recognize** which was the page that generated the page fault? The mechanism **depends** on the various **architectures**: the `x86` architecture for instance saves in the hardware the address that caused the fault (specifically in the `CR2` register), while on other platforms the OS must simulate the instruction and find by **bruteforce** the address that generated the fault.

We said that the OS restarts the faulty instruction whenever the page gets fetched from the disk, but if for instance the fault occurred in the middle of an instruction is hard to know where the OS should start again. In order to restart from scratch an instruction we need to know the **faulty instruction** and the **CPU state**.

There are some differences between the restarting of an **idempotent** and **non-idempotent** instruction: with an idempotent instruction the OS just saves the instruction and restarts

it from scratch, but with non-idempotent instructions is a bit trickier: suppose for instance that the instruction `MOV [%R1], +(%R2)` is being executed (so the value of `R2` gets incremented and stored in the memory address specified by `R1`), what happens if the page fault was caused by `R1`? We can't repeat the instruction from scratch, since the value would get incremented twice. Or for instance, let us suppose that there are some instructions that are used to move parts of the memory all at once: if not all the pages are in the memory, a page fault will occur. How can these two situations be solved? By **checking** before executing an instructions if **all the pages** that the instruction needs are **in** the **main memory**.

By the previous examples, we could infer that a page fault might happen at each process instruction, but luckily processes exhibit what it's called **locality of reference**: for this principle, if a process accesses an item in the main memory, it is likely that it will **reference** to **it again soon** (**temporal**) and that it will also **reference** a **close item soon** (**spatial**).

Regarding the performance, let us denote the following elements:

- $t_{\text{MA}}$: the **physical memory access time**;
- $t_{\text{Fault}}$: the **time needed to handle a fault**;
- $p \in [0, 1]$: the **probability** of a **page fault**;
- $t_{\text{Access}}$: the effective **time for each memory reference**;

We can calculate $t_{\text{Access}}$ with the following formula:

$$t_{\text{Access}} = (1-p) \cdot t_{\text{MA}} + p \cdot t_{\text{Fault}}$$

We can see how the time heavily depends on $p$: if the probability of a page fault is high, then the time will increase radically. If we want the effective access time to be at most $x\%$ slower than the basic memory access, what we have to do is the following:

$$x \cdot t_{\text{Access}} = (1-p) \cdot t_{\text{MA}} + p \cdot t_{\text{Fault}}$$

and then solve for $p$. More generically, if we want to set a threshold until which we allow for a page fault to happen (so we set $t_{\text{Access}} = (1+\epsilon) \cdot t_{\text{MA}}$), then we can find $p$ by solving the following equation:

$$p = \frac{\epsilon \cdot t_{\text{MA}}}{t_{\text{Fault}} - t_{\text{MA}}}$$

### 4.4.1 Page Fetching

There are still two problems related to virtual memory: when should the OS load a process' pages into the main memory (**page fetching**) and which page should it remove whenever the memory gets filled (**page replacement**)? Let's start with the **page fetching**: the goal of such action is to **make** the **physical memory look larger** than what it actually is, and we can do so by **exploiting** the **locality reference principle**. The ideal result would be a system that has the same performance of the main memory at the cost

and capacity of a disk. There are three page fetching strategies:

- **Startup**: all the pages of a process are **all loaded at once** in the main memory (in this case, the virtual address space **can't be larger** than the physical memory);
- **Overlays**: the programmer decides which pages must be loaded or removed. In this case the virtual address space **can be larger** than the physical memory, but it's **hard** to set up such a system, and it's also **error-prone**;
- **Demand**: the process tells to the OS whenever it needs a page stored on the disk; this way, the **page requests** are **managed entirely** by the **OS**. This is the **most used technique** out of the three.

The pure demand paging system works in this way: initially, at startup, **none** of the pages related to a process are loaded, but rather, a page is **swapped in only** whenever a process **references** it (thus whenever a page fault occurs). This way of swapping pages is called **lazy swapper** or **pager**. In practice, it's the opposite of loading all the pages at the process startup.

How does the pager work? It basically "guesses" when the pages will be needed, and it loads them ahead of time, trying then to avoid page faults. It requires though to "predict the future" (in general, predict the pages that will be referenced next), which is not easy. A possible (and more realistic) approach is, upon a page fault, to **load many pages** instead of just one; this approach works only if the process **access** the memory **sequentially** though.

In order to ease the swap of the pages, some OSs allow for a technique called **swap space**: it's a specific space on the disk which is reserved to the memory evicted pages. Various OSs implement this in different methods: on Linux for instance it's just an empty space of the disk, while on Mac it's a part of the file system (called `swapfiles`). Whenever a page needs to be swapped out, it will be copied to the disk. Processes' pages are divided into two main groups: **code pages** (which are read-only) and **data pages** (which are either initialized or uninitialized). Depending on the type of page, different optimizations might be applied. For instance, code pages (which don't change because of their read-only property) once they get removed they just get **loaded back in the disk,** and **make use** of the **filesystem**; data pages can change, and once they get removed they get **saved** in a **different paging file**, in order to not lose the content whenever the page will be loaded in the future; this kind of pages **needs to use the swap space**.

### 4.4.2 Page Replacement

Upon a page fault, a new page must be loaded from the disk to the main memory. If the memory has enough free frames, the pages will be loaded in these frames; if the memory doesn't have enough free space, then it must be freed. Depending on the algorithm used to replace the pages, the results may vary. There are mainly 4 types of algorithms: **random**, **FIFO**, **MIN** and **L**east **R**ecently **U**sed (**LRU**).

- **Random**: it's pretty much self explanatory: it picks a random page that will be swapped. It may look and sound inefficient, but it proved to be surprisingly good over some benchmarks;

- **FIFO**: removes the page that has been in the memory for the longest time (so the oldest). While it's easy to implement, it may remove frequently accessed pages;
- **MIN**: it's the most optimal algorithm, and it removes the page that will not be accessed for the longest time; its downside is that it needs to predict the future, which is very hard to implement;
- **L**east **R**ecently **U**sed (**LRU**): it's an approximation of the MIN algorithm, and it removes the page that hasn't been used in the longest time; this algorithm uses then the past in order to make observations regarding the future, and while such approach may work, it's not always true.

Is adding more memory to a system a possible solution that could reduce the number of page faults? The answer is that it depends on the used algorithm. For the **Belady's anomaly**, adding more page frames may cause more page faults with some algorithms. With FIFO for instance it augments the number of page faults, while instead with LRU it **always decreases**: why is it so? Because if with $n$ frames we have $m$ page faults, with $n+1$ frames we can't do any worst than $m$, since the situation with $n$ at each time $t$ is a subset of $n+1$.

## 4.5 LRU Implementation

How could we implement the LRU page replacement algorithm though? Let's go through some possible ideas:

1) we could store, for each page, the **timestamp** of the last time when a page has been accessed. Each time the table is full, we just remove the page with the highest **difference** between the current timestamp and the page timestamp. This method has a drawback though: each time we search for a page, we need to **update the timestamps**, which is somewhat costly. Moreover, if the table becomes full, then we have to **linearly scan** the list in order to identify the page that has to be replaced;
2) we could keep a list of the most recently used pages in front, while the least recently used ones are put at the end of the list (we would then be using an **heap**). The problem with this method is that the OS would have to **update the pointers** in the heap each time a page is referenced.

How can we do the algorithm then? Well, we **don't** really **need** a **perfect LRU algorithm**: a lot of systems provide some hardware support which approximates the LRU system fairly well. The most common way is to use one of the following two methods:

- **single reference bit**: for each page, there is a single bit which states whenever the page has been recently referenced. At the beginning, all the reference bits are set to 0, and on each access the bit related to a page is set to 1; such method is enough to distinguish the referenced pages from the non-referenced ones. There is a problem though: it could happen that all the pages have the reference bit equal to 1, and in that case the OS would have to remove one page at random;
- **additional reference bits**: for each page, instead of having only one bit, we have multiple bits. For instance, let us suppose that we reserve 8 bits for each page: each time a page gets referenced, we **shift to the right** all the bits and we set as the most significant bit a 1. The bits are not shifted only when a page is read,

but also **periodically** by the OS. At any time, the page with the **lowest number** given by the 8 bits is the least recently used page. Of course, the number of bits is adjustable, and can be changed depending on the needs and on the architecture.

One of the algorithms used for implementing the LRU policy is the **second chance algorithm**, which uses a **single reference bit** and a FIFO structure. The algorithm works as follows: all the pages are kept in a **FIFO circular list** and, on each memory access, the OS sets the reference bit to 1. Whenever the OS must fetch a new page, the OS goes through the list linearly and, depending on the value of the reference bit, the undertaken actions are different:

- if it finds a page with the reference bit equal to 0, it replaces such page and updates the reference bit to 1;
- if it finds a page with the reference bit equal to 1, it sets it to 0 and proceeds to the next page.

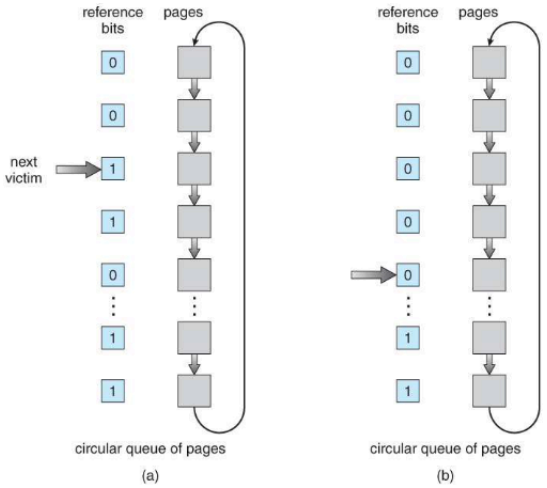



**Figure 4.9:** Example of the second chance algorithm

This algorithm is more **inaccurate** regarding the **age of the pages** with respect to the additional reference bits system, but it's way **faster** since it only has to deal with one bit instead of shifting multiple bits. The page fault management is way faster as well, since it's rare that the OS will scan linearly the whole list (unless of course all the reference bits are set to 1, but that's a very unlikely situation). This algorithm is also called **clock**, since it mimics the hands of a clock.

Whenever a page has to be replaced, the OS usually needs **two I/O operations**: one to **read the page** from the disk and one to **write the page** back. As intuition, we can say that it's **cheaper** for the OS to **replace** a **page** that **hasn't been modified**, since it won't need to be copied back to the disk. In order to make the system faster, the OS should give preference, when using the second chance algorithm, to the unmodified pages, and leave for last the modified pages.

| Ref | Mod | Meaning |
|---|---|---|
| 0 | 0 | The page hasn't been referenced recently and it hasn't been modified; it's the best candidate for a replacement |
| 0 | 1 | The page hasn't been referenced recently but it was modified |
| 1 | 0 | The page has been referenced recently but it hasn't been modified |
| 1 | 1 | The page has been referenced recently and it was modified; it's the worst candidate for a replacement |

In order to implement this priority, the OS will have to keep **one additional bit** to distinguish the modified pages from the un-modified ones. With the **modify bit**, if such bit is equal to 1 it means that the page was modified, otherwise if the bit is 0 it means that the page hasn't been

modified. We'll have then **different tuples** depending on the values of the two bits, and the way the algorithm searches for a swappable page is the following: it first looks for any page with a (0, 0) tuple, then a page with (0, 1), then (1, 0) and finally (1, 1). The **first page** found with the **lowest category** gets then **replaced**, and clean, un-modified pages are **prioritized**.

## 4.6 Multiprogramming and Thrashing

So far there was the assumption that only one process at a time was using the memory, but in reality now it's hard to find only one process accessing the memory at a time. How do we deal with multiple processes accessing the memory at the same time? First, each process, by the **locality principle** and by the **90/10 rule**, accesses most of the time on a **specific working set**. By having multiple working sets simultaneously allocated in the memory, we can increase the degree of multiprogramming. When the degree of multiprogramming is too high, the memory gets **saturated** with the various working sets of the processes, and thus **thrashing** occurs.

**Definition:** Thrashing

Event that occurs whenever the multiprogramming degree on a multi-core system is too high, causing the **tossing** of memory pages that are being **currently used** by the active processes. As a consequence, the **memory access time** becomes similar (approaches) to the **disk access time**, alongside with a **drastic degradation** of the **performance**

When thrashing occurs, the CPU utilization drops, since there are more I/O and memory access operations than actual CPU-bound ones. What can be done to avoid thrashing? We can't limit the degree of multiprogramming a priori, since it would result in an inflexible system; we can adopt though one of two different types of **allocation/relocation policies**:

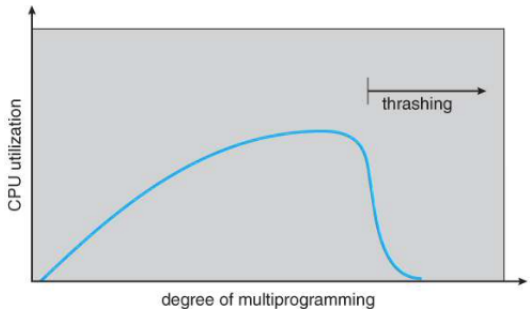



**Figure 4.10:** CPU's utilization upon thrashing

- **Global allocation/replacement**: for this policy, all the pages are put in a global, shared pool (or LRU queue); whenever a page must be replaced, any page might be the replaced one, be it a page belonging to the working process or not. The good side of this policy is that it's **flexible**, while the negative side is that **thrashing** is **still an issue**;

- **Local allocation/replacement**: for this policy, each process has its own queue, and the number of processes that can be run is limited by the size of the main memory. The LRU replacement affects only the frames given to one process. The good side is that there is an **isolation** between the various processes, so to better control the sharing of the memory, while the major downside is that the **performance** will eventually **decrease**, since each process won't have that much space in memory. In general, if we denote the number of physical frames available as $m$,

the number of processes as $n$ and the size of the $i^{\text{th}}$ process as $S_i$, then the total size of the processes $S$ is equal to

$$S = \sum_{i=1}^{n} S_i$$

The number of relocatable frames per process (where each process has the same number of frames as the other processes) is equal to $\frac{m}{n}$, while the proportional relocatable number of frames is equal to $\frac{m \cdot S_i}{S}$. The proportion for which the space gets reserved could be based on different factors: the space needed, the priority, etc...

Intuitively, we would think that a large process would need a large memory space, although that might not always be the case. Suppose that for instance a process generates a big piece of data (say a 1 GiB array) but uses only a portion of such data: in that case the working set of the process doesn't correspond to the correlated memory footprint. In general the OS must make sure that each process will have in the main memory **enough frames** to contain all the **working set's pages**. The working set doesn't refer only to the pages used at a time $t$, but to all the pages used and referenced during the past $T$ units of time.

How large should $T$ be though? Whenever a page fault occurs, the OS takes a unit of time $T$ in the order of 10ms to handle it. In 10ms, usually 10 million instructions can be executed, but $T$ needs to account for much more than 10 million instructions. The selection of the working set $\Delta$ must be **precise**, because if $\Delta$ is **too small** then it **won't encompass all the pages** that the process needs, while if $\Delta$ is **too large** then it will **include pages** that **won't be accessed anymore**. Even tracking $\Delta$ is an **expensive task**, since it has to be done for each process.

We can consider the working set $\Delta$ as a sliding window: at each memory reference one page gets out of the window and a new one gets in the window. Since keeping a window is expensive, the size of $\Delta$ is kept with a technique called **sampling**: each $k$ memory references, we consider the working set to be equal to all the pages referenced within these $k$ memory references.

In order to minimize the page fault rate $r(p_f)$, we can take a direct approach: we can put two thresholds $\tau_1$ and $\tau_2$ (such that $\tau_1 > \tau_2$), and we can make that if $r(p_f)$ goes above $t_1$ then we allocate more frames to a process, while if it goes below $\tau_2$ then we allocate less frames to that process. This way the memory is **dynamically managed**, so that the page faults rate is in between a specific area.

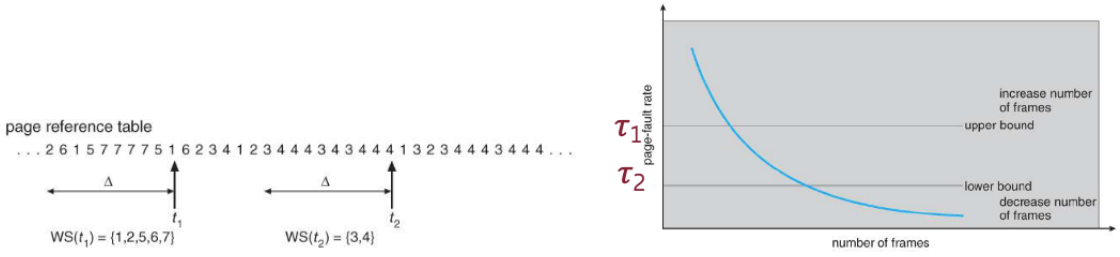



**Figure 4.11:** Graphic showing the working set $\Delta$ (on the left) and the two thresholds $\tau_1$ and $\tau_2$ (on the right)

# Chapter 5
# File System

So far we've seen some examples of file system APIs (to manage files) and some possible OS implementation of file systems. Now, let's have a look to *how* files are stored on **mass storage devices**.

## 5.1 Mass Storage Devices

There are 3 categories of mass storage devices: **magnetic disks** (**HDD**s), **solid state disks** (**SSD**s) and **magnetic tapes**. Magnetic disks and solid state disks are said to be **secondary** storage, while magnetic tapes are called **tertiary** storage: this is because usually magnetic tapes were used as backups, even if now they are being replaced with magnetic disks.

Magnetic disks are made of several **platters** of fixed radius, all connected to a central **spindle**, which rotates at a constant speed. Depending on the platters, we can make a further distinction: if the platter is made of rigid metal, then we have an **hardisk**, while if the platter is made of flexible plastic, we have a **floppy disk**. In both cases, all the platters have 2 working surfaces. Each surface is divided into concentric **tracks**, and each track is then divided into **sectors**, of usually 512B. Each sector contains a **header** and a **tail**, alongside some **checksum information**. Just like the main memory, a larger sector would reduce the space taken by each head and tail on the overall sector space, but it would increase the **internal fragmentation**. The set of tracks that are at the same distance from the center is called **cylinder**.

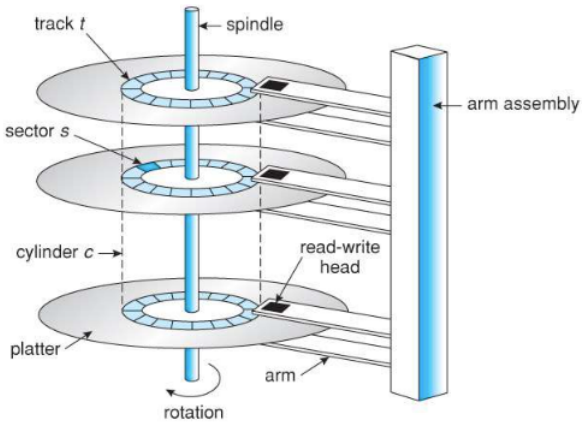



**Figure 5.1:** Teardown of a magnetic disk

The data on an hard drive is read and written by **read-write heads**, each head placed on a separate **arm**; all the arms are controlled by a common **arm assembly**, moving all **simultaneously** from one cylinder to another. A standard configuration uses one read-write head per surface.

The overall capacity of an hard drive is given by:

$$C = \underbrace{H}_{\text{Number of heads}} \cdot \overbrace{T}^{\text{Number of tracks per surface}} \cdot \underbrace{S}_{\text{Number of sectors per track}} \overbrace{B}^{\text{Number of bytes per sector}}$$

Until the end of the 80's, each track had the **same number of sectors**, with the consequence that the outer sectors would be larger than the inner ones. Since the disk controllers have no intelligence, there were multiple problems related to the **frequency** and **timing** from the innermost to the outermost tracks of the disks, but also the **capacity** of the disk was determined by the **maximum bit density** that a controller could handle.

Nowadays, the **number of sectors** per tracks **varies** with the radius, so that the innermost track holds less sectors than the outermost one. The **bit density** is also kept **constant**. **Smarter controllers** allow to refer to the disk through **logical addresses** rather than physical addresses: this new way to partition a disk is called **Z**one **B**it **R**ecording (**ZBR**).

### 5.1.1 Data Retrieval Timings

Each physical block of data is specified by a **tuple** (`Head`, `Cylinder`, `Sector`). Each disk block is numbered from the outermost cylinder, which is numbered 0. The disks rotate at a **constant speed** of 7200rpm, which is equal to 120rps. Naturally, **outermost tracks** spin **faster** than the innermost ones. Whenever some data must be transferred from the disk to the memory, it must account for 3 times: the **positioning time** (also called **seek time** or **random access time**), the **rotational delay** and the **transfer time**. The first two are **mechanical** delays, while the transfer time is **electronic**.

The **positioning** (or **seek**) **time** is the time needed to **move** the **heads** to a **specific track**, and it **includes** the time needed for the heads to **settle** on the disk. It heavily **depends** on how **fast** the hardware moves, and it's typically the **slowest** step of the aforementioned three. We could define it as the **bottleneck** of the overall disk data transfer.

The **rotational delay** is the time needed for a desired **sector** to **rotate under** the **read-write head**. It can range **from 0** (so the sector being already beneath the read-write head) up **to** a **full revolution** of the disk (since the disk can spin only in one direction). On average, the rotational delay is equal to $0,5$ revolutions, and it's usually equal to 4ms. The rotational delay represents the **second highest bottleneck** between the three times.

The **transfer time** is the time needed to **electronically move the data** from the disk to the memory, and it's expressed as a bandwidth (so in Bps). In total, the **data transfer time** (sometimes, the total transfer time is also called transfer rate) is equal to

$$\text{Data Transfer Time} = \text{Seek Time} + \text{Rotational Delay} + \text{Transfer Time}$$

### 5.1.2 Structure of the Disks and their Interfaces

A hard drive is usually addressed as a **large one-dimensional array of logical blocks**, where each logical block is the **smallest unit of transfer** (such as 512B). Each sector gets **mapped** to one logical block sequentially, starting from sector 0 up to sector $n$. We denote as sector 0 the **first sector** of the **first track** of the **outermost cylinder**. The mapping proceeds in order through the **outermost track** which contains sector 0, then it passes to the **other tracks** in the **same cylinder** and, when a cylinder is fully mapped, the mapping continues to the **next cylinder**. The last sector $n$ will be a sector in the innermost cylinder.

The disk read-write heads **do not directly touch the disk**, but they rather "fly" on a tiny air cushion: if an head touched the disk, then we would call such event an **head crash**. Such crashes may **permanently damage the disk** or, in the worst-case scenario, **destroy it**. In order to avoid such a risk, the disk heads are always **parked** when the computer is turned off. Usually the disks are **not hot-swappable**, but depending on the drive, some are. Each disk drive is connected to the computer via an **I/O bus**. Some of the most common interfaces for disks are **EIDE**, **ATA** and **SATA**, **USB**, **FC** and **SCSI**.

Each disk drive has two controllers: one is the **disk controller**, which is placed into the disk drive itself, and the other one is the **host controller**, which is located at the **end** of the **computer's I/O bus**. The CPU **only communicates** with the **host controller**, which then handles the communication with the disk controller. The data is transferred through an **intermediary cache** located on the disk, so that if the disk transfer rate is higher than the transfer rate on the bus between the host and the disk controller then the data doesn't get lost.

Let's go back to the time analysis of the disks: the general bottleneck on a disk is given by the seek time and the rotational delay: in order to minimize the data transfer time of a disk, those two timings must be minimized. How can we minimize them? There are different ways, and they divide in two categories: **hardware optimization** and **software optimization**. Regarding the hardware optimization, an option is to make **smaller disks**: there will be a **lower seek time** (since the arms must travel a smaller distance) or via the use of **fast spinning disks**, which will **lower** the **rotational delay**.

Regarding the software optimization strategies, how can we approach this problem? An idea could be to **schedule disk operations** in order to minimize the head movement, or the **data** on the disk could be **layout** so that the related data is located on close tracks. Another way could be to place **commonly used data** on a **specific portion** of the **disk**. Moreover, the **block size** of each sector **matters**: a **too small** block size would need **more seeks** to transfer the same amount of data, while **too large** blocks would cause more **internal fragmentation**.

## 5.2 Disk Scheduling

We saw some possible hardware solutions regarding on how a delay could be lowered, but there's little that can be done at the hardware level: as we said, the **OS can help** a lot in these cases. During the execution of a process, a lot of I/O read and write operations are done, so, in order to have lesser waiting timings (with these algorithms we mainly aim

to **reduce** the **seek time**), we could **arrange** all the **I/O requests** such that the heads won't have to travel too much between one request and the other. There are multiple **scheduling algorithms** that can help with that, such as **FCFS**, **S**hortest **S**eek **T**ime **F**irst (**SSTF**), **SCAN** and **C-SCAN**.

### 5.2.1 First Come First Serve (FCFS)

It's a classic algorithm, not that much different from the FCFS algorithm used for the processes scheduling. For this algorithm, we have a **queue of I/O requests** that get granted one by one, **following** the **order of the queue**. It's an **easy** and **fair** algorithm, which **works well** whenever the system is **underloaded**. As the number of requests increases though, the algorithm's **performance** quickly **drops**. It's **used by SSDs** though, since they do not require any mechanical movement and the data retrieval on them is nearly immediate (similarly to a random access in the main memory).

### 5.2.2 Shortest Seek Time First (SSTF)

The **S**hortest **S**eek **T**ime **F**irst (**SSTF**) selects the next I/O request depending on which has the **shortest seek time**. The list gets **automatically sorted** with respect to the aforementioned criteria and **followed linearly**. However, it's an **unfair** algorithm, since it may cause **starvation** for the requests that are too far from the head, and it's **not optimal** since it only **minimizes** the **seek time** in a **local area**, not on the overall disk. It is said to be greedy since it **only looks one step ahead**.

### 5.2.3 SCAN and C-SCAN

For the SCAN algorithm, the head moves continuously back and forth from the beginning of the track to the end of it, and the requests are **served** as the **head passes** through the disk (just like an elevator). It needs to have a sorted list of the requests. There is a simple **optimization** of this algorithm, which is called **LOOK**: for this optimization, we don't let the head go from the first to the last sector each time, but we let the **head** go only **until** the **last request** that must be served. For instance, if on a track there were 100 sectors, and if the last request was at sector 78, then we would let the head go until 78 and then go backwards.

**SCAN**, unlike SSTF, **cannot suffer** from **starvation**, since each request will be guaranteed. Of course though the **seek time** and the **waiting time** are **larger** with **SCAN** than with SSTF. Now, we assumed up until here that the time that an head takes to travel from a point $A$ to reach a point $B$ is linear, but it's actually **not linear**, since the head accelerates and decelerates. This is why SCAN takes more time, since the heads must account for the acceleration and deceleration. Another version of SCAN, called **C-SCAN**, tries to reduce these timings by making the **head return** to the **beginning** of the **track** whenever it reaches the end. It's identical to SCAN for the approach that it takes with the requests. Of course, there is C-SCAN and there is also **C-LOOK**. C-SCAN moreover doesn't prioritize more recent requests, and it avoids the start and the stop of the mechanical head movements.

All these algorithms are **implemented** in the **disk controller**, and each disk comes with the algorithm ready to run. It's possible to implement more efficient algorithms, but they should be implemented in the **disk driver** (in the OS kernel).

## 5.3 Magnetic Tapes and Disk Failure

Magnetic tapes are nowadays used only for **backup** purposes, since accessing data on them is particularly **slow**: there is **no random** or **direct access**, but rather only **sequential** access. After the reading or writing starts, the speeds are basically the same of an hardisk. The capability of a tape ranges from 20GB to 200GB. It's not frequent to see tapes today: they are gradually being replaced by disks.

Real-world systems today use multiple disks, but the more disks a system uses, the more it's likely for some of the disks to go bad at any given time. Increasing the number of disks of a system actually increases the **M**ean **T**ime **T**o **F**ailure (**MTTF**) of the system. For instance, let us suppose that we have a system with $N$ disks, where each disk has a MTTF of 10 years (roughly 4000 days). The probability for a disk $i$ to fail is then the following:

$$p = \frac{1}{4000} = 0,00025 = 0,025\%$$

If we associate each disk to a random variable $X_{i,t}$ (where $X_{i,t} \sim \text{Bernoulli}(p)$ that outputs 1 (a disk failed) with probability $p = 0,025\%$ and 0 (a disk works) with probability $(1-p) = 99,975\%$. For simplicity, we assume that $p$ is equal for all disks), then we can compute the expected number of failures in a certain moment of time $t$. Denoting with $T$ the sum of all $X_{i,t}$, then we consider $T$ to have a behaviour like the one of a binomial random variable:

$$T \sim \text{Binomial}(N, p)$$

The expectation of $T$ is equal to:

$$E(T) = Np$$

In order to avoid that the content of a disk gets lost forever in case of failure, the data of a disk can be copied in multiple backup disks or, in some systems, in multiple working disks: redundancy in a system greatly improves the chances of a single disk to not fail.

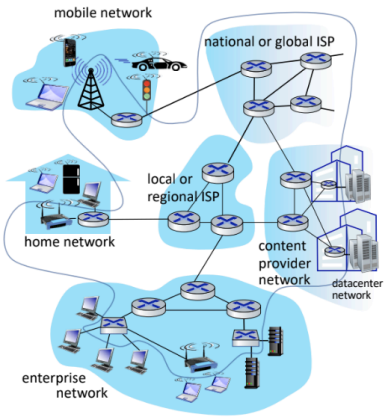
# Chapter 6
## Networks

In recent history of computing, nearly every device is always connected to the Internet and exchanges information through it. But what is the Internet? We can describe the Internet as a **computer network**, which connects billion devices altogether. All these devices are called **hosts** or **end systems**, and act on what is called the **Internet's edge**. All end systems are connected through **communication links**, which allow for the data to move from one source to another, and **packet switches**, like routers and switchers, which decide the most optimal path that the data must follow in order to reach the destination. So, given these pieces, how can we define what a network is?

**Definition:** Network

A **network** is a **collection**, managed by an organization, of **devices**, such as **hosts**, **packet switches**, and **links**

We can consider the Internet as a network of networks: multiple networks, handled by multiple organizations, are all united under a unique network, which allows for the single networks to exchange information between each other.




Communication between devices is established through **protocols**. An example of protocol is the `HTTP` protocol: whenever an user searches for a site through a browser, an `HTTP` request to a web server is sent, and if the request is accepted, then the **payload** (the content requested by the request) is returned.

**Definition:** Protocol

A **protocol** is a **set of rules** which defines the way a connection between two devices is established. Such rules regard the **standard** that has to be used to communicate, the **bandwidth**, the **format** and **order** of the messages and the **action** taken upon every request

A commonly used standard on the Internet is, for instance, the **RFC** (**R**equest **F**or **C**omment), developed by the **I**nternet **E**ngineering **T**ask**f**orce (**IETF**).

# 6.1 Structure of a Network

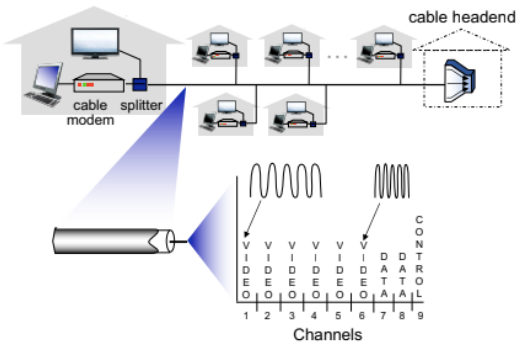
A network is made of multiple sections, which all cooperate to exchange information and data upon set standards and protocols. There are 2 important parts: the **network edge** and the **network core**.

## 6.1.1 Network Edge, Access Networks and Physical Media

The network's edge is made of all those devices that "sit" at the edge of the network and use the network's services. The devices that act in the network's edge are called **end systems**, or **hosts**, because they host a web application that exchanges information through the network. Hosts are divided in two other subcategories: **clients** and **servers**.

We call **access network** that network that allows all the end systems to be connected to the first router. There are various ways that allow an end system to connect to the network, depending on the **link**, which is the physical media that **connects** not only the end systems to the first router available, but also connects the various router creating a net of connections.

In many countries, Internet links (such as optic fiber or copper lines) are shared with other services: in the US for instance, the Internet line is shared with the television line, while in Italy the Internet line is shared with the telephone line. In order to differentiate the various signals, different **frequencies** are used. Each frequency is used only by a certain type of signal: this way, regularity is ensured.




There are various types of links (or **physical media**): they may be **wired** or **wireless**. When we talk about wired links, we can have either a **coaxial cable** or a **fiber optic cable**.

**Coaxial cables** are made out of two concentric copper conductors, they are **bidirectional** and can have **multiple frequencies** per channel. **Fiber optic cables** instead are made of glass fibers, which allows light to flow through it. Each light pulse is equal to a bit. It is more fragile than the coaxial cables, but it's substantially **faster** (from 10 to 100 Gbps). Fiber optic has also a **lower error rate**, and is **immune** to **electromagnetic noises**.

Wireless links are possible through **radio frequencies**: the signal is carried out through the use of various radio waves and other bands of the electromagnetic spectrum. There is no physical wire, but is **half-duplex** (both sources can communicate through the same channel, but not simultaneously). Plus, there could be some obstacles in the way of the signal, which usually leads to signal loss.

Devices on the Internet exchange **messages**, may them be text messages or data mes-

sages. Each message is divided in multiple data chunks called **packets**, which travels through **communication links** and **packet switches** (such as **routers** and **switches**).

When an host has to **send** a message, it takes the following steps: first, the message is loaded and split into multiple packages. Each package has a specific **length** of $L$ bits; once a packet is done, it gets sent to the network at a **transmission rate** $R$. The transmission rate is defined, among many things, from the **link capacity** (or **link bandwidth**).

The time needed to transmit an $L$-bit packet into a link is called **packet transmission delay**, and it's calculated through the following formula:

$$\text{PTD} = \frac{L}{R} \qquad \left[\frac{\text{bits}}{\text{bits/s}}\right]$$

## 6.1.2 Network Core

Whenever we want to send a content through the Internet, we usually send it as a **message**. Messages can be of any type: a JPEG image, a text message, a MP4 file, and so on... In order to send any kind of message through the Internet, it must first get decomposed in **packets**, which are smaller data chunk. Each packet, in order to each a destination, goes through a net of packet switches (which can be either **routers** or **link-layer switches**). The way the path is determined is pretty much a Dijkstra's algorithm.

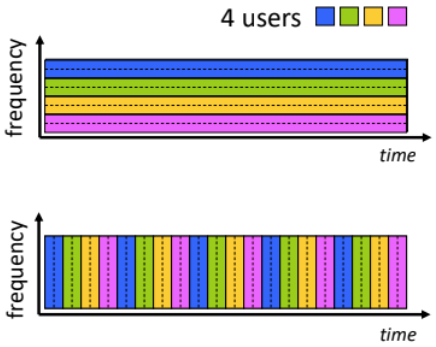
A message on the Internet is sent in the following way: after dividing the message in various packets of $L$ bits each, then the packets get **stored** in the **router** on a speed of $R$ bits per second. Once the packets get stored on the router it gets finally forwarded to the final destination (or to the next router). This process is called **store-and-forward** because a package is first stored and then sent.

Routers forward multiple packets from multiple sources. Since the capacity of the router is limited, if there is more request than the usual that goes beyond the router's limit, then **queuing** happens. More specifically, this happens when the **arrival rate** of the **link** (in bps) **exceeds** the **transmission rate** for the **link** on a given period of time. Routers have a **buffer**, which allow for packets to be temporarily stored in a virtual queue, but if the buffer becomes full, then there is a point where **packet loss** happens.

There is an alternative to packet switching, which is called **circuit switching**: following this method, each link is **partitioned** into **multiple sections**, and each section is **reserved** for **only one type of communication**. The channel though can be **occupied by only one user at a time**, so if two users request simultaneously access to the line, only one of them (usually the first one) will be accommodated. If the channel is not used, then it's said to be in **idle**, and it's free for any other user to use it. This approach is an alternative, but it's not optimal since only one user at a time can use it.

There are two ways to make **circuit switching**: **F**requency **D**ivision **M**ultiplexing (**FDM**) and **T**ime **D**ivision **M**ultiplexing (**TDM**).

For the Frequency Division Multiplexing, the link is divided in multiple frequencies, and each frequency is allocated to one single type of communication. For the Time Division Multiplexing though, instead of partitioning the link, time is partitioned. In each time slot, one particular frequency is allowed, and in such slot the communications are sent at the maximum capacity of the link.




## 6.2 Internet Structure

In the world there are thousands and thousands of various access points that need to connect to some other clients or servers at the edge of the network, but how are they connected? They can't be connected between each other directly, since it would be too much confusing (also, we would have $O(n^2)$ connections). How can we solve this? We could have for instance some smaller networks that redirect and handle all the traffic between access points, such that there would be less connections and they would be handled gracefully. We call such small networks **I**nternet **S**ervice **P**roviders (**ISP**). There isn't a single ISP in the world though, but there are many of them, even if they connect between each other through some physical **peering links** or via some **I**nternet E**x**change **P**oints (**IXP**). Some smaller regional ISPs may coexist in some regions, in order to better handle the traffic in a restricted area.

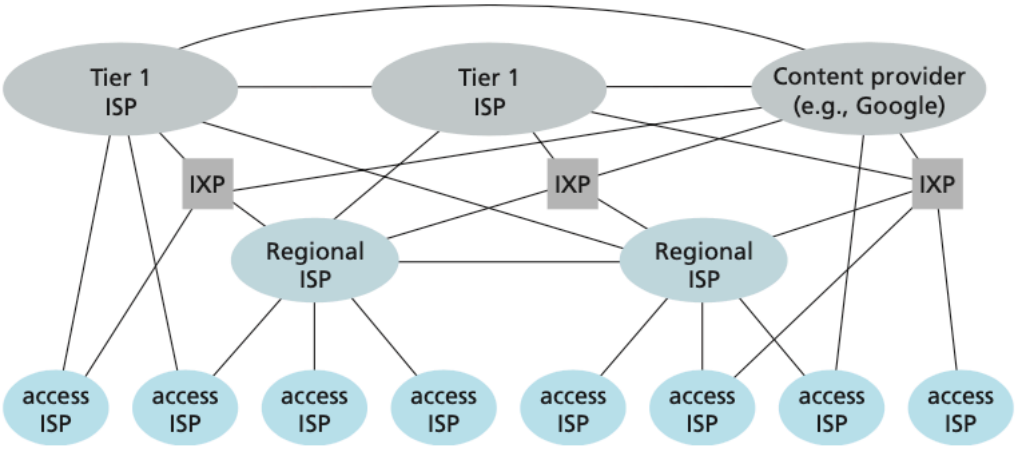



**Figure 6.1:** Structure of the Internet, with the **ISP**s and **IXP**s

On top of the **ISP**s there are the **content provider networks**, like Google, Microsoft, etc... We can also see from **figure 4.1** that some **Tier 1 ISP**s are mentioned: such ISPs are commercial companies such as Fastweb, Vodafone or Wind that provide a national (and sometimes international) Internet coverage. **Figure 4.1** also shows an interesting detail: the content providers may sometimes try to access to the access ISPs directly, without passing through the Tier 1 ISPs. It may happen though that some access ISPs might need to first connect to a Tier 1 ISP in order to connect to a content provider.

## 6.3 Performance of the Network

There are lots of elements which influence the speed and the performance of a network. Queues for instance noticeably influence the performance of routers, and as we saw it's

also possible for packets to get lost if the queue is too long.

The 4 main delays that determine the **total nodal delay** of information sending are:

- the **nodal processing delay**;
- the **queue delay**;
- the **transmission delay**;
- the **propagation delay**.

The **nodal processing delay** is given because before sending the packets, the router checks the header of the packets and for eventual bit errors if the transport protocol that is being used is the TCP; after that, the **output link** gets **determined**, and the packet gets sent into the queue. The nodal processing delay usually takes some microseconds.

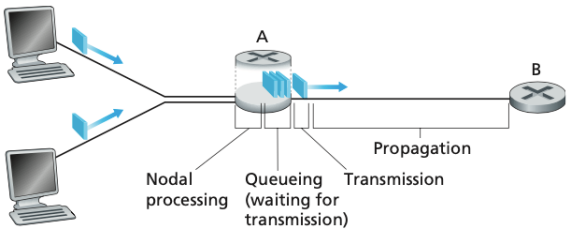



**Figure 6.2:** All the different types of delay

The **queuing delay** is given by the **amount of queuing packets** in the router's cache, and indirectly depends on the capacity of the router's cache. The more the queue is empty, the less time it takes for packets to depart from the router. It can take from some microseconds to some milliseconds.

After the packet has arrived to the end of the queue, it needs some time to be transmitted, which is called **transmission delay**: such delay depends on the length of the packet $L$ and on the capacity of the link $R$. We can find such delay with the following formula:

$$t_{\text{trans}} = \frac{L}{R}$$

Usually transmission delay are in the order of microseconds to milliseconds.

When a packet has to go from a router $A$ to a router $B$ it may take some time: such time is called **propagation delay**. Packets travel as fast as a link allows them (so for instance, in fiber optic the speed is near to $c$), and the time needed to go from a source $A$ to a source $B$ it's then defined by

$$t_{\text{prop}} = \frac{d}{s} \qquad \text{where } d \text{ is the distance between } A \text{ and } B \text{ and } s \text{ is the packet's speed}$$

Now, given all these delays, we can define the total nodal delay as the sum of all the delays:

$$t_{\text{nodal}} = t_{\text{proc}} + t_{\text{queue}} + t_{\text{trans}} + t_{\text{prop}}$$

Regarding the routers: it's possible that sometimes packets get lost while transmitting. This usually happens when the queue is full, and so when new packets come to the router they can't be stored inside the cache. Depending on the transport protocol that is used in a context, packets might be sent again from the source or not.

We can still explore a bit further the queue delay: such delay doesn't stop simply to the length of the queue, but it also depends on the intensity of the traffic. Such intensity is measured as

$$Ti = \frac{\overbrace{L \cdot a}^{\text{arrival rate of bits}}}{\underbrace{R}_{\text{service rate of bits}}} \quad \text{where } a \text{ is the average packet arrival rate}$$

Depending on the value of such intensity, we can say different things regarding the traffic:

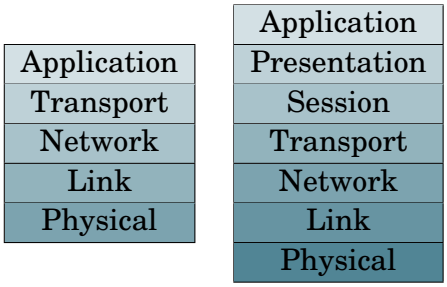
$$\begin{cases} Ti \simeq 0 & \text{the average queuing delay is small} \\ Ti \to 1 & \text{the average queuing delay is large} \\ Ti > 1 & \text{too many packets are getting queued, the waiting time is infinite} \end{cases}$$

The effectiveness of a network is also measured in terms of **throughput**, which is the rate at which bits arrive at the receiving end. We can measure both the **instantaneous** (in a given moment in time) or the **average throughput** (distributed over a certain amount of time). On a path made of multiple links, if a link has a lower bandwidth with respect to the other links, then such link is said to create a **bottleneck**: the maximum throughput on the whole path will be equal to the bandwidth of the weaker link.

## 6.4 Protocol Layers

In our daily lives, networks handle everything: money transfer, text messages, videocalls, etc... Each scope has different requirements though: for instance, transferring money has to be a precise operation, where no bits must be lost in the communication between the user and the bank. In videocalls instead, bits might be lost, but it wouldn't be that much of a problem. Depending on the scope, each protocol is designed differently. There are two types of protocol stacks: the **Five-layer Internet protocol stack** and the **Seven-layer ISO OSI reference model**.

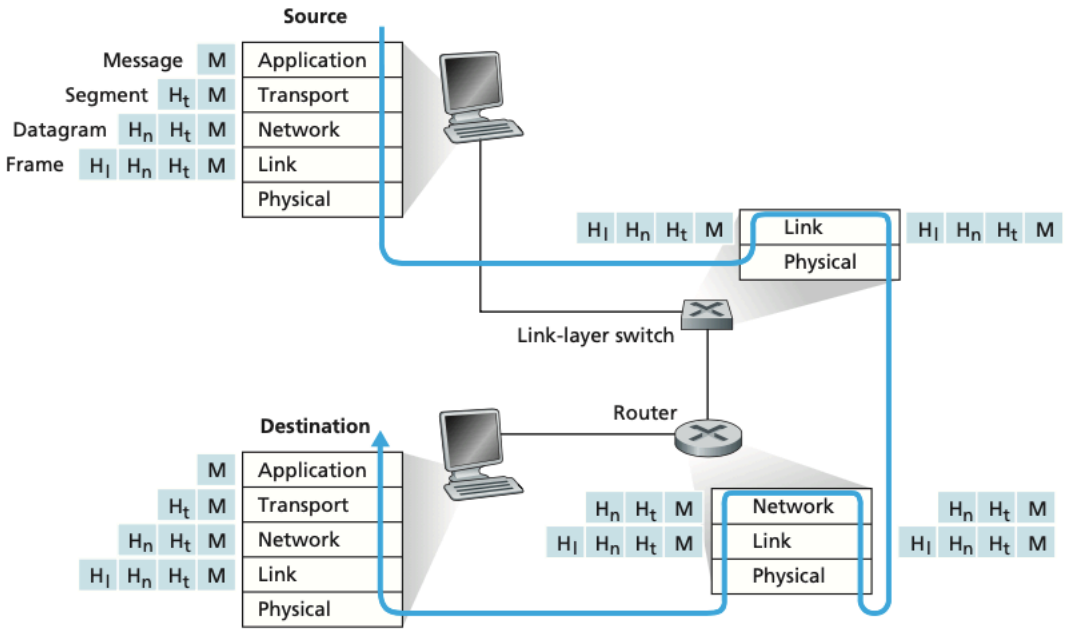
Protocols offer different capabilities and services, depending on their final use. In general, a protocol is divided into **multiple layers**, which are basically a set of steps which packets go through. A **service** is a **functionality** given by **one or multiple combined layers** below the examined layer. Some examples of layers are for instance `HTTPS` or `SMTP`. The internet in particular is made of a **five-layered protocol stack**. The five layers are the following:




- **Application layer**: it contains the network applications. Multiple protocols are part of this layer, such as `HTTP`, `IMAP`, `SMTP` and `DNS`. The packet exchanged from this layer is called **message**;
- **Transport layer**: it transports application-layer messages by encapsulating them: an **header** $H_t$ is added to each message, creating a **segment**. There are two transport protocols used on the internet: `TCP` and `UDP`. `TCP` is a connection-based protocol, while `UDP` is a connection-less based protocol;

- **Network layer**: this layer encapsulates the segment, by adding another header $H_n$, and thus creating a **datagram**. All the internet uses the `IP` protocol;
- **Link layer**: the datagram is encapsulated with yet another header, the link header $H_l$. The final packet is called **frame**. The link layer must trace the most optimal route among the network so that the packets go from a point $A$ to a point $B$;
- **Physical layer**: this layer is entrusted to move the single bits of the frames. It's different from the link layer because the link layer moves bigger data packets (thus the frames), while the physical layer moves the single bits from a point $A$ to a point $B$ or subpoint $A_2$.

Encapsulation and sending works in this way: a message that must be sent to a specific destination is encapsulated by the various layers, which add an header on each step. The header specifies the protocol that each layer uses. The final package (called frame) is sent to the switch, which analyses the link and physical headers and re-compiles them, depending on the protocols that it uses. The same happens when the frame arrives at the router, which changes the headers of the network, link and physical layers, and then sends the frame to the final destination which, upon deconstruction of the frame, receives the message and processes the request.




There are two layers that are usually not used in the Internet protocol stack: the **presentation** and **session** layers, which are placed between the application and the transport layer. The **presentation layer** allows the applications to interpret what the sent data actually mean

**Example 6.4.1**

Suppose there is exactly one packet switch between a sending host and a receiving host. The transmission rates between the sending host and the switch and between the switch and the receiving host are $R_1$ and $R_2$ respectively. Assuming that the switch uses store-and-forward packet switching, what is the total end-to-end delay to send a packet of length $L$? (Ignoring queuing, propagation delay and processing delay).

The total delay is given by $\frac{L}{R_1} + \frac{L}{R_2}$

# Chapter 7
## Application Layer

We saw that when a message has to be sent from a source to another, then such message is sent through layers. The application layer is the highest layer, and uses services provided by the lower **transport** layer.

There are a lot of different apps that run on the network, such as social networks, text-messaging services, the web, etc... When creating a network app, we just need to write the final application that runs on the clients, we don't have to write an application for the network core devices, since they don't have to (and should not) run them. Network core devices and the physical links have just to send bits of data, regardless of what is their meaning.

When designing an application, is important also to consider the **application's architecture**. One of the most used architectures is the **client-server paradigm**: for such architecture, we have an always-on server with a permanent IP-address, and the clients (the applications) make requests to the same address. There is a reason why the IP address should be fixed and static: because the server is always available to everyone, if the IP address changed every now and then, then the clients would have problems reaching the server. If the IP address remains static, then the connection is "always" guaranteed.

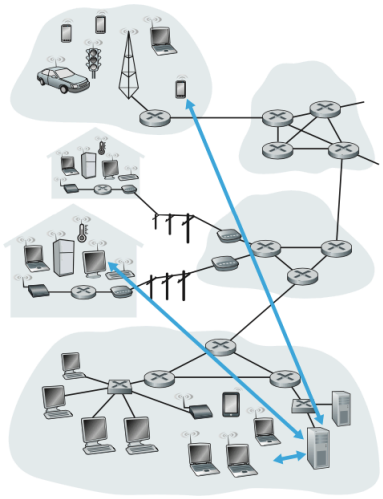


**Figure 7.1:** A server-client connection

Another architecture is the **P2P architecture**. **P2P** stands for **peer-to-peer**, and it's a direct communication between two devices. Differently to the client-server architecture, P2P does not have an always-on server, or well, it doesn't have **any** server (in the traditional way). The communication happens between the two peers, without letting the message go into some servers. The communication is established only when both peers are active. This architecture is **self-scalable**, since the growth of the network is directly related on the number of peers. The management is a little bit more complex though, since the IP addresses change frequently.

# 7.1 Communication Between Processes

How do applications communicate between each other? In this case, we talk more about **processes** rather than applications: a **process** is a program running within a host machine. Within the same host, multiple processes communicate via some inter-process communication standards which are defined by the OS. If a process has to communicate between different hosts though, then the communication is established via the exchange of messages. When using a P2P application, there are two types of processes:

- **client processes** are processes that initiate the communication;
- **server processes** are the processes that wait to be contacted.

Processes are able to communicate and send/receive messages through **sockets**. Sockets are paths that allow the processes to communicate with the transport layer. There is one per host/server. Before sending messages though, the process needs to have an **identifier**, which is functionally similar to the `IP` address, but different in the form. The `IP` address is not enough to identify a process, since the `IP` address specifies only the host/server, not the process (it stops at the network layer). Identifiers include both the `IP` address and the **port** number; the port number is a number specifying a single process. Usually, by default 80 is the default port for the `HTTP` messages, while 25 is the standard one for the emails.

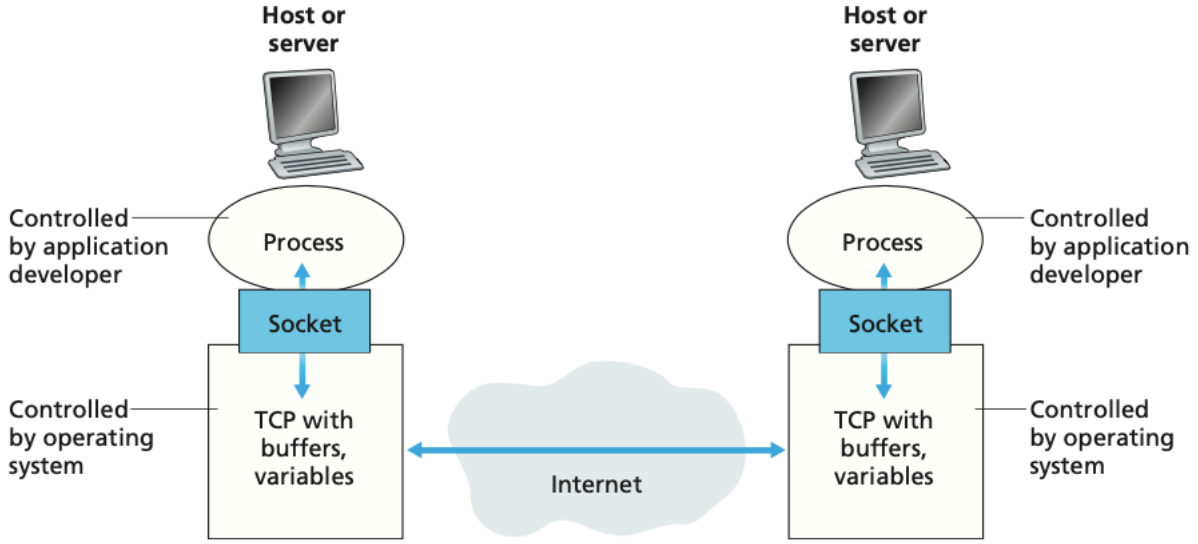



**Figure 7.2:** Processes exchanging messages through their sockets

So, to sum up, an application layer protocol defines these different elements:

- the **type** of message sent, its **syntax** and its **semantics**;
- the **rules** for responding and sending messages between processes.

There are two types of protocols: **open** protocols and **proprietary** protocols.

While designing an application, it needs some transport services in order to be able to communicate with the network:

- **data integrity**: some applications require reliable data transfer performances, such as bank services, while others might not need to have the best performances, such as streaming services;

- **timing**: some applications need the lowest delay possible in order to be effective;
- **throughput**: some applications might need a controlled throughput in order to work fine, while others (called elastic applications) just use as much throughput as they can;
- **security** such as encryption, data integrity, etc...

The Internet provides two main transport protocol services: **TCP** and **UDP**. The **TCP** protocol is a **connection-oriented** service with reliability features. It's connection-oriented because it first exchanges transport-layer control information regarding the type of connection; this process is called **handshaking**. After the handshaking, the connection is established. The established connection if full-duplex, and when there is no need for the connection to exist, it gets closed. It's moreover reliable because it transmits each bit that gets sent. TCP moreover has a congestion-control mechanism, to ensure that all the bits will reach the final destination.

The **UDP** protocol is a lightweight protocol which has less services than **TCP**. The data transfer is unreliable, there is no congestion-control mechanism, and there is no guarantee that the data will reach the end.

## 7.2 Web and HTTP

When it was firstly invented, the internet was just a way for researchers between different cultural centers to exchange information, transfer files from various hosts and to send and receive electronic mail. After the **W**orld **W**ide **W**eb (**WWW**) protocol was established by **Tim Berners Lee** in the 1994, the use of the Internet on-demand contents got into people's lives. The **H**yper**T**ext **T**ransfer **P**rotocol (**HTTP**) is now (alongside its more secure version **HTTPS**) the most used **application** protocol on the web. It allows to share web pages between a server and the hosts, and is responsible for defining the way the connection between server and host must be done.

A web page is made of multiple nested objects, such as an **H**yper**T**ext **M**arkup **L**anguage (**HTML**) page, an image, an appled, and so on... In order to access to each object, we use the **U**niform **R**esource **L**ocator (**URL**), which is made of two parts:

`https://www.leonardobiason.com/home/index.html`

a host name and a path name

The `HTTP` protocol uses the **TCP** protocol as its underlying transport layer. When a host makes a request to a specific `HTTP` address, a connection gets established between host and server. When that happens, browser and server's processes access the `TCP` through their own sockets. The reason why `TCP` is used instead of `UDP` is that it's efficient in its own way if some data packet get lost. With the `HTTP` protocol no data are saved by the server from each request: that's why `HTTP` is called a **stateless protocol**.

Depending on the request that is being sent and on the application, a question arises: when a user uses the `HTTP` with multiple processes simultaneously, do we keep all the `TCP` connections together or do we establish multiple different connections? This is where the

difference between **non-persistent** and **persistent** connections arises. A non-persistent connection establishes a separate connection for each request from the client, while a persistent connection keeps all the requests through the same connection.

**Non-persistent connections** work in this way: suppose that a client wants to browse a URL that links to a page with 10 objects; the procedure would be as it follows:

1) the client process initiates a connection with the server on the standard `HTTP` port (80). The connection passes through the client's socket and reaches the server, passing through the server's socket;

2) the client, after establishing a connection, sends the request for the `HTML` page, and analogously to step **1**, the request passes through the right sockets;

3) the server, once it receives the request, processes the request itself and sends back to the client the requested objects through an `HTTP` message;

4) the server then proceeds to shut down the `TCP` connection;

5) the client receives the response message, seeing that the `HTML` page references other 9 objects. In order to retrieve them, all the steps are executed again until all the objects are successfully retrieved.

Now, we can calculate the time needed from when the client issues the request to when it receives the final file from the server: in order to do it, we want first to consider a smaller quantity of time, called **round-trip time** (**RTT**), which identifies the time needed for a packet to **travel from** the **client to** the **server** and then **back** to the **client**. With the **RTT** we are also including packet-propagation delays, queuing delays and packet-processing delays. The process of establishing a non-persistent connection can be considered a **three-way handshake**: the first and the second handshake are, respectively, the request for a connection from the client to the server and the acceptance or refusal of such request by the server. The third handshake is the `HTTP` request from the client. In total, a connection takes the time of establishing 2 `RTT`s plus the **transmission time** taken to send the `HTML` file.

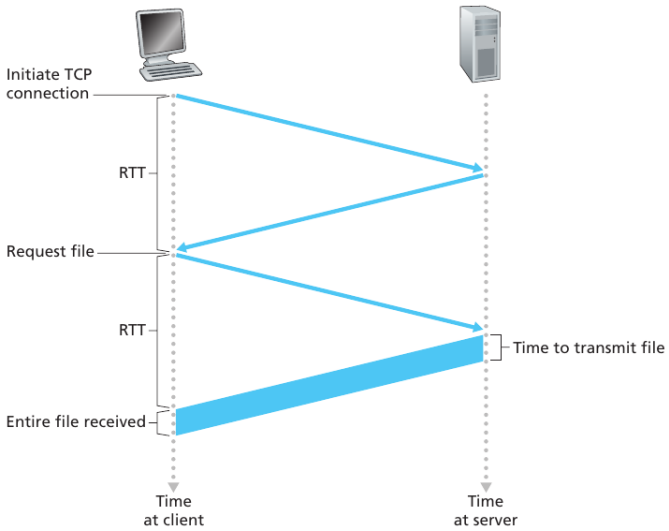



**Figure 7.3:** Visual representation of a non-persistent connection

There are two problems with non-persistent connections: the first is that having to establish a single connection for all the objects in a web page creates a **heavy workload** for the **server**, which usually serves thousands of clients simultaneously; the second is that each object suffers from a **significant delay**, given by the fact that **two** `RTT`s have

to be established in order to exchange one single item. A solution to these problems is represented by the **persistent connections**.

**Persistent connections** had been introduced with the protocol **HTTP 1.1**: such type of connections remain active even after sending a response to the client; this way even an entire HTML page with all its objects can be sent through one single TCP connection and, if multiple Web pages reside on the same server, it will be possible to send them all through that same connection. Moreover, the exchange of multiple items can be made **back-to-back**, without the need of waiting for the server to answer again to other pending requests (which happens in **pipelining**)

## 7.2.1 HTTP Request Messages

Whenever a client sends a request message to a server, there are two possible types of messages: **request** and **response**. All requests are encoded in ASCII, in a human-readable format. Here follows an instance of an HTTP **request** message:

```
GET /somedir/page.html HTTP/1.1
Host:  www.someschool.edu
Connection:  close
User-agent:  Mozilla/5.0
Accept-language:  fr
```

Let's take a closer look at this request message: it consists of five lines, and each line ends with a **carriage return**. The first line is called **request line**, while the other lines are called **header lines**. We can thus separate the request:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

Let's examine the **request line** first: we can divide it into 3 pieces: the **method field**, the **URL field** and the **HTTP version field**

| method field | URL field | HTTP version field |
|---|---|---|
| GET | /somedir/page.html | HTTP/1.1 |

The **method field** specifies the type of the request, and can assume several values such as GET, POST, HEAD, PUT and DELETE. Most of the requests are done with the GET method, which is used to request an object, be it an HTML page or any object within it. The object that has to be retrieved is specified by the **URL field**, while the **HTTP version field** is pretty much self-explanatory. Most of the browsers usually allow only for a query string up to 240 characters, but each request can be bookmarked and reloaded. In case some input parameters are needed, they usually appear after a "?" symbol. In most of the cases, the POST method is used rather than the GET method, and that happens because with the POST method the request parameters are not visible, the amount of data sent with it is not limited and the POST requests can't be bookmarked or copied to be reloaded.

Now, down to the header lines: the header line `Host: www.someschool.edu` specifies where the object resides, and such line, as we'll see further on, is important for the Web proxy caches. The `Connection: close` line tells the server to establish a non-persistent connection, and to close the established connection after the object has been sent. The `User-agent: Mozilla/5.0` line is used to define the user-agent, which is the browser type used to establish a connection. This is useful for instance when there are multiple versions of a Web page for each browser. Finally, the `Accept-language: fr` line specifies that the server has to send a French version of the requested object if such version exists, otherwise it will send a default version of it. Such line is also called negotiation header.

After the header lines there usually is an **entity body**, preceded by a blank line, but in the case of the previous request we don't have anything inside it: why is that? Because for `GET` requests it's **blank**. The entity body is used with the `POST` method whenever the content of the Web page have to depend on the user's input. For instance, whenever the user fills a form, or when an online research is made. The `HEAD` method is used usually for debugging purposes, since it works exactly like the `GET` method, but it doesn't send also the objects from the server. The `PUT` method is used to upload objects on a Web server, and the `DELETE` method deletes an object from a Web server.

We saw what a typical request message looks like, now let's analyze what a response message looks like instead:

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

This answer is made out of 3 sections: the initial **status line**, the six **header lines** and the **entity body**. The **status line** is made of three elements: the **protocol version**, the **status code** and the **status message**. Some of the most famous codes are the following:

- `200 OK`: the request has been approved and succeeded;
- `301 Moved Permanently`: the content has been moved to another location, which will be specified in the `Location` header;
- `400 Bad Request`: the request is not formatted properly, or the server couldn't understand the request. It's usually implicated as a generic error code;
- `404 Not Found`: the requested element doesn't exist on the server;
- `505 HTTP Version Not Supported`: the `HTTP` protocol used by the client isn't supported by the server.

Status codes are divided in 5 sections:

- `1XX`: informational messages;
- `2XX`: successful response;
- `3XX`: redirection messages;
- `4XX`: client error messages;
- `5XX`: server error messages.

Some elements in the **header lines** differ from the request message: for instance, there is a `Date` line now, which indicates when the response was created and sent by the server; the `Server` line substitutes the `User-agent` line, which now specifies the type of server and the OS that runs on that server. The `Last-Modified` line specifies the last time when a content was modified, which can either mean when it was the last time that the content was modified or when it was created (if no modifications were made afterwards). The `Content-Length` line specifies the length in bytes of the content, while the `Content-Type` line specifies the type of content.

### 7.2.2 Cookies

As we have said earlier, `HTTP` connections are **stateless**: they do not save any data regarding the exchanged information. But how can we save some data in order to make further communications between server and client smoother? We use **cookies**. Cookies are made out of 4 components:

1) a cookie **header line** in the `HTTP` response message;
2) a cookie **header line** in the `HTTP` request message;
3) a cookie file kept on the user's end system, which is managed by the browser;
4) a back-end database which stores the cookies.

Cookies are used for every kind of Web practice: to keep a user's session active, or to track its activities. They are a tool that can either provide a better user experience either, in some cases, be a monitoring tool (such kind of tools is usually seen as an invasion of privacy). We can make a distinction between **first party cookies** and **third party cookies**: the first type of cookies includes all the cookies that are restricted to the user's activity on a specific site and that are handled by the site; the latter type includes all the cookies that track a user's activity across multiple servers.

### 7.2.3 Web Caching

Sometimes, whenever we try to access to a remote content two times in a row, we may notice that the second time that we access it the loading of the content is way faster. What we called "loading" is actually the retrieval of the remote content. Whenever this happens, it's because there is a **Web cache**, or **proxy server**, involved. The goal of a Web cache is to store and satisfy Web requests without involving the origin server: this way the workload for the server can be lighter.

A Web cache can be usually set via the browser that the client uses. The exchange of data when a Web cache is used is as follows:

1) When a browser makes a request for a HTTP content, it first establishes the connection with the Web cache;
2) If the Web cache has the item stored inside it, then it proceeds to return the content to the client. If the cache doesn't have the content requested from the client, it will proceed to request it via the opening of a connection with the remote server.
3) The Web server sends the object requested by the Web cache to the Web cache itself, which stores a copy of the item inside it and proceeds to send the original to the client.

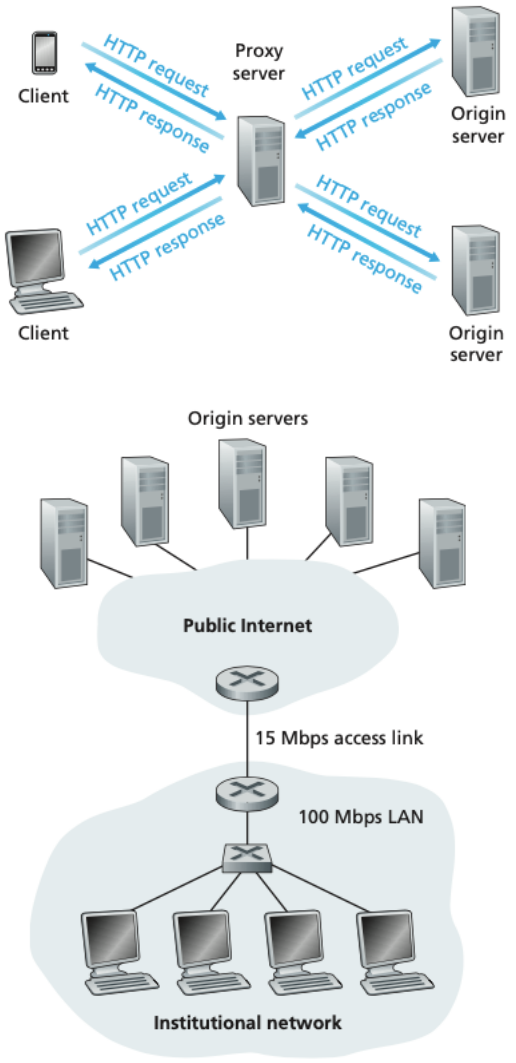



**Figure 7.4:** Example regarding the use of a proxy server. In the second image, there is an example of bottleneck

Usually a Web cache is **simultaneously** a **client** and a **server**, because it serves as a server for the client and acts as a client with the remote servers. Caches are used because they allow for a **lighter workload** on the line, **reducing** noticeably the **time** for the client's request. The Internet is filled with caches, because it helps whenever there is a poor connection with the remote server: if the connection between a client and a Web server is, without caches, poor and slow, then the performance can be improved via the use of a Web cache; if by adding a Web cache a better link can be used between the cache and the client, then there is a better performance whenever a frequent content is asked multiple times. In such case though, a phenomena called **bottleneck** would arise: this usually happens when, like in this case, the link speed from the origin server to the Web cache is smaller than the time link speed from the Web cache to the client. Let's make an example:

**Example 7.2.1**

A network uses a custom LAN, where the link speed is 100 Mbps. The link's speed between the network and the public Internet is 15 Mbps. Suppose also the following data:

- the average item requested is 1 Mb sized;
- the RTT from a client in the network to the Internet and back to the client is 2 seconds (will be called informally "Internet Delay").

The total response time is given by:

$$\text{total response time} = \text{LAN delay} + \text{access delay} + \text{Internet delay}$$

where the access delay is the delay between the network's access point and the router to the Internet. Let's compute the traffic intensity in the LAN and on the access link (from the Internet router to the network's router):

- regarding the LAN delay:

$$\frac{\overbrace{15 \text{ requests / sec}}^{\text{speed between the access points}} \cdot \overbrace{1 \text{ Mbit / request}}^{\text{size of the average request}}}{\underbrace{100 \text{ Mbps}}_{\text{speed of the LAN}}} = 0.15$$

- regarding the access delay:

$$\frac{\overbrace{15 \text{ requests / sec}}^{\text{speed between the access points}} \cdot \overbrace{1 \text{ Mbit / request}}^{\text{size of the average request}}}{\underbrace{15 \text{ Mbps}}_{\text{speed of the link between the access points}}} = 1$$

In practical terms, the LAN delay translates in a negligible delay (it would be around tens of milliseconds), but the access delay would be noticeable. There are two solutions to this problem: **to buy a better link between the two access points** or **to establish a Web cache**. Let's look at the advantages that both situations would bring:

- in the first case, the network's holders would have to buy an access point which should support 100 Mbps wide connections, and also the link between the two access points should be upgraded. The final result would be that both the delay of the LAN and of the access points would be equal (0.15) and thus the final time for requesting an item would be **two seconds**;
- in the second case, a Web cache would be installed in the LAN. Usually hit rates of Web caches go between 0.2 and 0.7. For the sake of the example, we'll assume that the Web cache hits with a 0.4 rate. The Web cache will serve any client which requests an item stored inside itself at a speed of, for instance, 10 milliseconds, and that would correspond to the 40% of the requests. For the remaining 60%, we would make a request to the origin server; this would bring down the traffic intensity from 1 to 0.6. The final delay for a connection would be then:

$$0.4 \cdot 0.01 \text{ seconds} + 0.6 \cdot 2.01 \text{ seconds} \simeq 1.2 \text{ seconds}$$

  We see how it's a better improvement over the implementation of a faster link: it would realistically cost less to implement a Web cache, and it's faster than a better link.

Usually, Web caches are provided by companies who handle **C**ontent **D**istribution **N**etworks (**CDNs**): companies which distributes Web caches geographically. There are two types of **CDNs**: **shared** ones (such as Akamai) and **dedicated** ones (such as Google). A problem may arise with Web caches though: what happens when a content gets modified? If a

cache stored an old version of such content, then it would send the old content rather than the newer one. There is a solution to this, and it's called **conditional GET**.

**Definition:** Conditional GET

An HTTP request is said to be a **conditional GET message** if:

1) the request message uses the `GET` method;
2) the request message includes an `If-Modified-Since:` header line

The **conditional GET** works in the following way: the first time a new content gets requested by a client, the cache will request it for the client, store a copy of it and send it to the client. The cache also saves the `Last-Modified:` header line, alongside its value. Whenever a second client requests that content, the cache performs a **conditional GET** request: such request will include a line, called `If-Modified-Since:`, followed by the `Last-Modified:` value of the content. If the content wasn't modified, the cache will obtain a response like the following:

```
HTTP/1.1 304 Not Modified
Date:  Sat, 10 Oct 2015 15:39:29
Server:  Apache/1.3.0 (Unix)

(empty entity body)
```

If instead the content was modified, then the server would provide the cache with an updated version of the content, which would be then sent back to the client.

### 7.2.4 HTTP/1.1, HTTP/2 and HTTP/3

Throughout time, multiple versions of the HTTP protocol established, each one of them adding some features. Nowadays, there is no such thing as an imposed version of the HTTP protocol, but they all coexist together and can be used freely. With all the various versions of HTTP, major improvements have been done throughout each version, and each update had a different scope.

With the transitioning from **HTTP/1.1** to **HTTP/2**, the goal was to ease the transferring of multiple objects through one connection between two end points. HTTP/1.1 introduced the possibility to have **multiple**, **pipelined GET** requests over one single TCP connection: the server would respond in order to the GET requests, following the **FCFS** logic (**F**irst **C**ome **F**irst **S**erve). This was a problem in the case where big objects had to be transmitted: they would use all the connection to first transmit such objects and then to transmit the other ones (such problem is called **H**ead-**O**f-**L**ine blocking, thus **HOL blocking**); moreover, if an object failed to be sent correctly, then for the TCP connection it would be sent again, and that would be an issue with big objects.

This was then fixed with **HTTP/2**: this version of the HTTP protocol added the possibility to specify an **order** for which objects had to be transmitted and the push of unrequested (thus embedded) objects to the client. Plus, objects were divided into **multiple frames** before being sent, in order to **mitigate HOL blocking** (the way it works is similar to the Round Robin algorithm). Most fields and methods remain unchanged from HTTP/1.1.

There are some cons regarding HTTP/2: it's true that it mitigates HOL blocking and facilitates for the request of embedded objects, but it still **uses one single TCP connection**; plus, vanilla TCP doesn't give any **security** advantages. Only with HTTP/3 there will be some **security** features and **per object error** and **congestion** control

## 7.3 Email, SMTP and IMAP

Email services are popular on the Internet since its release to the public: it allows fast and convenient **asynchronous communication** between two users. Back in the days, emails were just made of plain text, but now they support attachments, hyperlinks and HTML formatting. Each email system is made out of 3 parts: the **user agents**, the **mail servers** and the **S**imple **M**ail **T**ransfer **P**rotocol (**SMTP**).

**User agents** are client applications that allow the users to compose, send, view, reply, forward and save emails, and some examples are for instance Thunderbird, Microsoft Outlook or Apple Mail. Whenever a user wants to compose a message, it does it via the user agent. Whenever the message gets sent from the user agent, it then goes inside the **mail server**: there, it enters the **outgoing message queue**, and will be sent to the other end(s) once it will reach the bottom of the queue. Whenever a user wants to read a message, it has to retrieve it from the mail server via the user agent. Each user has its own mailbox, which is located in one of the servers.

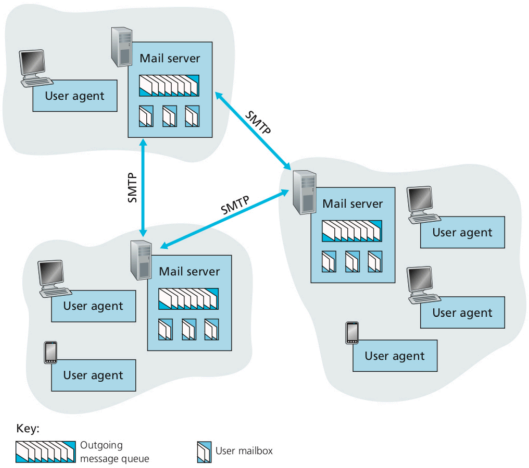



**Figure 7.5:** Visual representation of the Internet email system

In the case of a failure while sending a message on the recipient's end, then the sender has to account for that. For instance, suppose that user $A$ is sending a message to user $B$, but the message can't be sent for any reason: the message will stay in a **message queue** and the server, each 30 minutes (it changes from server to server, but it's usually every 30 minutes). If after several days there is no success regarding the delivering of a message, then the server will remove the message and notify user $A$ that the message couldn't be sent.

**SMTP** is the principal protocol that handles the exchange and sending of emails throughout the Internet. It's based on the TCP protocol and, as every other server, it has two sides: a **client side**, which executes on the sender's mail server, and a **server side**, which executes on the recipient's server. The SMTP protocol is one of the **oldest** in the Internet, and some of its **limitations** are still noticeable right now, which are a bit annoying considering the technological jump that was made in the past 40 years. An example of a limitation which still persists is that the **body** of an email must still be **encoded** in **7-bits ASCII characters**; at the time, it made sense to put a restriction because there was a lack of space in all computers, but now it's kind of obsolete: now, in order to send a message with some attachments, the attachments have to be encoded from binary to

7-bit ASCII and then back to binary when the message reaches the recipient.

SMTP doesn't use intermediate servers: it establishes a **direct TCP connection** between the two servers, no matter where they are located. But how is a connection established between two mail servers? Connections with the SMTP protocol are always established on **port 25**; if the connection can't be established, the server tries again in a second time. If the connection is instead established, a **handshake** is done between the two sides and, once it's done, the sender's server sends the message to the recipient's server. Here follows an example of the exchange of messages between two mail servers:

```
S: 220 biason.org
C: HELO uniroma1.it
S: 250 Hello uniroma1.it, pleased to meet you
C: MAIL FROM: <didattica@uniroma1.it>
S: 250 didattica@uniroma1.it ...  Sender ok
C: RCPT TO: <leonardo@biason.org>
S: 250 leonardo@biason.org ...  Recipient ok
C: DATA
S: 354 Enter mail, end with .  on a line by itself
C: This is a test to check how emails work.
C: I did it for educational purpouses only
C: CRLF.CRLF
S: 250 Message accepted for delivery
C: QUIT
S: 221 biason.org closing connection
```

In the previous message there are two types of lines: `C` and `S`. For `C` we denote the **client**, while with `S` we denote the **sender**. In this message there are 5 commands that have been used: `HELO`, `MAIL FROM`, `RCPT TO`, `DATA` and `QUIT`. The first is used for establishing a reply and a handshake, the second and the third to exchange more information regarding the message that is currently being sent, the fourth to send the data related to the message and the fifth to break the connection. Since SMTP is a persistent connection, if **multiple messages** have to be sent to a same user, then they will all be **sent together** and only after all the messages are sent the connection will be closed with the `QUIT` method.

SMTP and HTTP protocols have some differences and some similarities. Regarding the similarities, SMTP and HTTP both use **persistent connections**, and they are both used to transfer files. Regarding the difference, one is the scope for the file transfer: HTTP is said to be a **pull** protocol (since it asks for a file stored in a server, and the client which initiated the connection is the one which makes the request to the server) and SMTP is a **push** protocol (since it asks to send a file to the server, in fact it's the client's mail server the one which makes the request to push the message); another difference is that **SMTP requires** all messages to be **encoded** in 7-bit ASCII messages, while HTTP doesn't have any such requirements. One last difference is that **HTTP**, whenever it has to send multiple objects, it **encapsulates each object** in its own HTTP message, while **SMTP sends all** the **objects** in the **same message**.

A question may arise: if we use a mail server, then we inevitably take more steps when

we have to send an email, since we first have to send our email to the mail server and then the mail server sends the email to the mail server of the recipient. Why can't we skip such passage? That's because if for instance we had a structure where the email exchange was only between the two computers of the two communicating users, then they would need to be always on in order to receive messages. Plus, if a user wanted to send a message, without a server it would lose the message in the case that it wasn't delivered.

Now though, what happens if the recipient wants to download a message? He can't use the SMTP protocol, since that's used to **push** messages from one source to the other, so what can we do? There are two ways: we can use either the protocol **P**ost **O**ffice **P**rotocol - Version **3** (**POP3**) or the **I**nternet **M**ail **A**ccess **P**rotocol (**IMAP**). Even HTTP can be used. These two protocols are called **mail access protocols**.

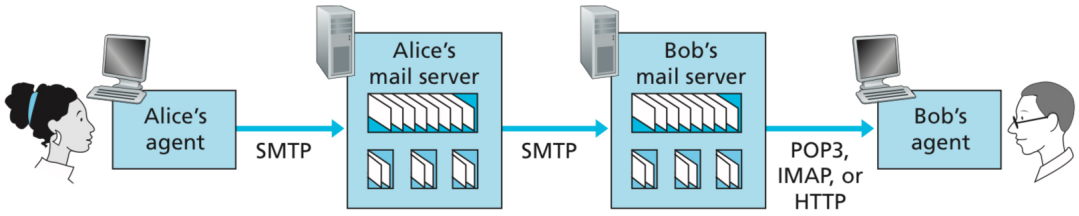



**Figure 7.6:** Example of an user (Alice) sending an email to another user (Bob)

The **POP3** protocol is a pretty much simple protocol with few features, which works in the following way: whenever the user connects to the server through the protocol's standard port, the port 110, it receives all the messages from the server. Once the messages have been received, they are deleted from the server. This can be annoying when a user wants to access to his email mailbox from multiple clients.

The **IMAP** protocol was invented as an alternative to the "download-and-delete" method of the POP3 protocol. It works on port 143. All the messages remain on the server, and can be deleted from the authenticated user from a remote client. Moreover, a user can create different folders in which it can store the mails. As a standard, IMAP maps each message to a folder. IMAP also allows user agents to retrieve only some parts of the email, such as the header or the body.

## 7.4 Domain Name System (DNS)

Whenever we want to establish a connection with a server, we always need an `IP` address (**I**nternet **P**rotocol Address): each server has its own `IP` address, which is unique to that server. Whenever we connect to a site though we rarely use the numeric `IP` address, but we rather use the **hostname**, which is a human readable alias. Each time that we connect to a server through the hostname, then the hostname gets translated into the `IP` address. There are two types of `IP` addresses: `IPv4` and `IPv6`. Each `IPv4` address is composed of 4 bytes separated by dots, and it's **hierarchical**: the more we go from the left to the right, the more specific the `IP` address becomes.

The service that translates hostnames into `IP` addresses is called **D**omain **N**ame **S**ystem (**DNS**), which is basically a translation directory service. More specifically, the DNS is both a **distributed database** implemented in a hierarchy of **DNS servers** and an application-layer protocol, which allows hosts to query the database. `DNS` works with

the UDP protocol, but why is that? Because if we used the TCP protocol, then we would have longer querying times, so since the DNS has to be a very fast service, reliability gets sacrificed in order to boost the querying speed. The translation works in the following way:

1) The host executes the **resolver**, which is the client application of the DNS;
2) The browser extracts the hostname from the URL and it passes it to the DNS;
3) The DNS server gets queried, and it responds with the IP address;
4) Once the IP address gets obtained by the browser, it will start a TCP connection to the server specified by the IP address.

There are some services that the DNS offers, alongside the translation service:

- **Host aliasing**: sometimes hostnames are long and complicated, so it's easier to have shorter hostnames in order for a user to remember it. Host aliasing allows for a hostname to have a shorter alias. For instance, a hostname such as `relay1.company.com` can have as alias `www.company.com`. We call the original hostname (in the previous example is `relay1.company.com`) the **canonical name**. Such reasoning also applied to mail domains;
- **Load distribution**: DNS also allows to redirect the traffic into multiple servers having the same content: such thing happens for instance with big websites such as `amazon.com`. The DNS contains a set of addresses that point to a set of servers with the same content, and each time that a host queries for such site, the DNS will reply with the whole set, only that the order of the items in the set will be always different, depending on the traffic of each server.

How does the DNS work though? One single DNS server can't hold all the IP addresses of the whole internet, they must be split: the whole mapping is instead organized on many DNS servers. There is a three-level hierarchy which organizes the servers: the **root**, the **top-level domain** (**TLD**) and the **authoritative** level. The authoritative level is where all the organizations-related DNS servers are: such servers manage the organization of the traffic towards the organization's Web services. In order to better manage all the DNS requests, there are also some smaller local DNS servers, which also have a cache of the recently used hostnames, so that the connection doesn't have to pass through the root server. Such servers don't strictly belong to the hierarchy.

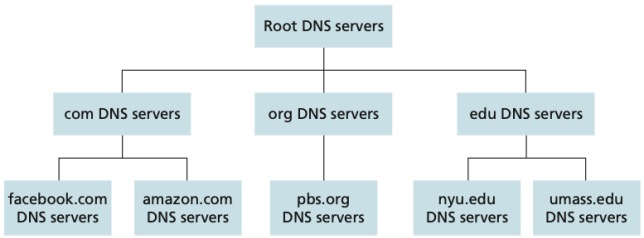



**Figure 7.7:** Organization of the DNS

For example, suppose that we want to connect to `www.uniroma1.it`, how will the connection be established? First, the hostname enters the local DNS server folder: if the hostname is present in the cache then it will just retrieve it, otherwise the root server will be queried. Supposing that the hostname isn't cached and that we have to connect to the root server, after having established a connection, the query will search inside the `.it` DNS servers folder and finally it will search inside the `uniroma1` server folder; after retrieving the IP address, it will return it to the host.

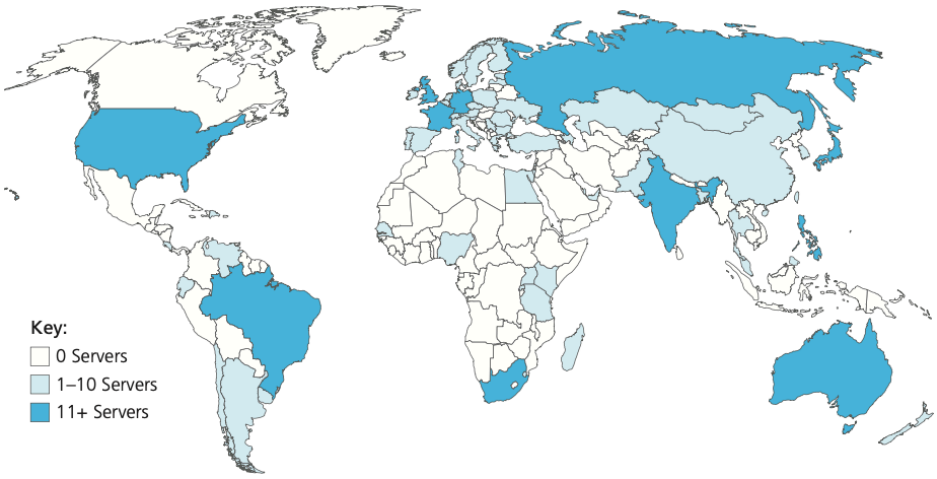



**Figure 7.8:** World map with the distribution of the DNS servers

The previous query method is pretty much general, but there is a more specific search that is done: first, the client sends the query to its local DNS server. The local DNS server, depending if it already queried recently a specific hostname, might either return the address or forward the query to a root DNS. In the second case, the root DNS would reutrn to the local DNS with the indication for a query to the appropriate TLD server, which would then do the same with the appropriate authoritative DNS server. Once the query succeeds, the result gets sent back to the requesting host. There are two types of DNS queries: **iterative queries** and **recursive queries**. An example of recursive query is when the local DNS, if it doesn't have the DNS already stored inside its cache, makes the query on behalf of the client. An iterative query is for instance when the same local DNS queries the root server.

### 7.4.1 Storing of Records and Retrieval

All the data stored inside the DNS servers is called **r**esource **r**ecords (**RR**). Each RR is a tuple containing 4 values:

```
(Name, Value, Type, TTL)
```

TTL stands for **T**ime **T**o **L**ive, and it helps with the managing of the DNS servers: some addresses and hostnames may be temporary, and in this way we automatically disregard them once they are not needed anymore. The Name and Value fields depend on the content of Type:

- `Type = A`: for this type Name is the hostname and Value is the IP address of the server (so for instance Name = `relay1.maxwell.bobcompany.com` and Value = `127.42.91.73`);
- `Type = NS`: in this type the Name is a domain (such as `uniroma1.it`) and the Value is the hostname of an authoritative DNS server, which is used to redirect the query to the other DNS server which contains information on how to obtain the IP address of the required host;
- `Type = CNAME`: in this type Value stands for the canonical hostname of the alias hostname specified in the Name field;
- `Type = MX`: for this type the Value field is the canonical hostname of a mail server, where the alias is specified in the Name field.

Queries for DNS are made through a specific standard which works for both the query messages and the reply messages form the server. The standard is made as follows:

- the first 12 bytes make the **header section**, which specifies in 16 bits the identification of the message's author, and in other 16 bits some flags are specified. Available flags are, for instance, if the message is a query or a reply, if recursion is desired or available, if reply is authoritative, etc...
- the **questions section** holds all the queries made from the client;
- the **answers section** holds the results of all the queries made in the RR format;
- the **authority section** holds all the records for authoritative servers;
- the **additional information** is pretty much self-explanatory: it holds additional helpful information regarding the message.

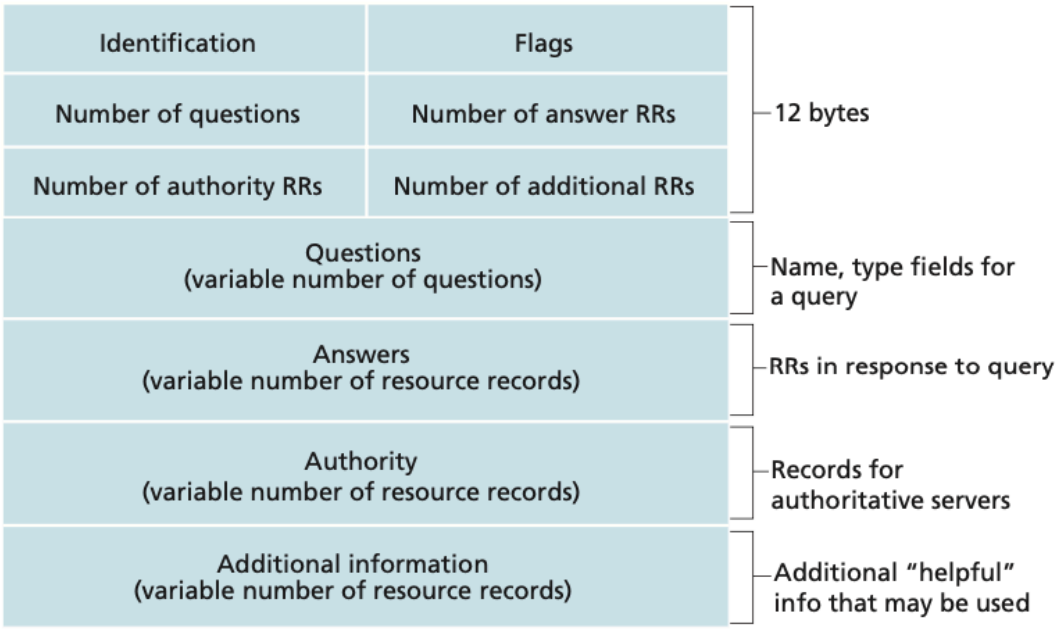



**Figure 7.9:** Composition of a DNS query and reply message

When you want to create a new domain, two paths are available: one way is to refer to a **registrar**, which provides names and `IP` addresses of authoritative name servers and has the authorization to add into the `DNS` some RRs with the information regarding the sold hostnames; another way is to create an authoritative server locally with the `IP` address `212.212.212.1` and define your own RR tuples.

DNS servers also need a certain level of security, since they act just as any other server, they only have a different scope. It could be possible for DNS server to suffer from DDoS attacks. Root servers should be able to block ping requests of type ICMP (will be discussed further in another section), but TLDs might not be always protected. It could also be possible to spoof some data from the queries made from the other users. This is all a theoretical speech: DNS servers have proved to be more than robust against this kind of attacks. Still, it's important to know that there are some risks.

## 7.5 Video Streaming and CDNs

In this period of time, the streaming of prerecorded videos accounts for the majority of the traffic in most of the ISPs. The challenge is: how can such heavy content be delivered to around 1 billion users in the world simultaneously, without using all the bandwidth? Moreover, each user has different capabilities in terms of physical links: some people use a wireless connection, while other use wired ones, some have a better bandwidth, while others have a worse one, etc...

Videos are a collection of frames (usually 24 or 30 per second) and each frame is encoded as an array of pixels. Since the bandwidth can't be used all by the streaming service, there must be some sort of compression used on the videos, in order to remove redundancy. There are two ways used to reduce redundancy: **spatial coding** (which happens **within** the image) and **temporal coding** (which happens **between** the images). For **spatial coding** we denote the process where instead of sending $N$ times the same value, we send the value once and the times that it gets repeated; with **temporal coding** we denote the process of sending, instead of two full frames, only the differences between one frame and the next one. Video can also be sent in two different bit rates: **C**onstant **B**it **R**ate (**CBR**) and **V**ariable **B**it **R**ate (**VBR**). For instance, videos encoded with the MPEG1 format follow a CBR of 1,5 Mbps, while MPEG2 and MPEG4 formats follow a VBR of 3 - 6 Mbps in the case of the first and 64 Kbps - 12 Mbps in the case of the latter.

In a simple scenario, whenever a client requests for a video, the content will be asked to the server, which will reply with the requested content and send it through the Internet back to the host. An example of a video data transmission is showed here in **Figure 6.10**: in the first part the video is sent from the server and, after a while, the client starts to play the video. Such action is called **streaming**. Here a problem arises: the bandwidth between the server and the client will **vary** over time, so we have to account for it in order to have the lowest possible number of packets lost. **Packets loss** translates in **poor video quality** or **buffering**.

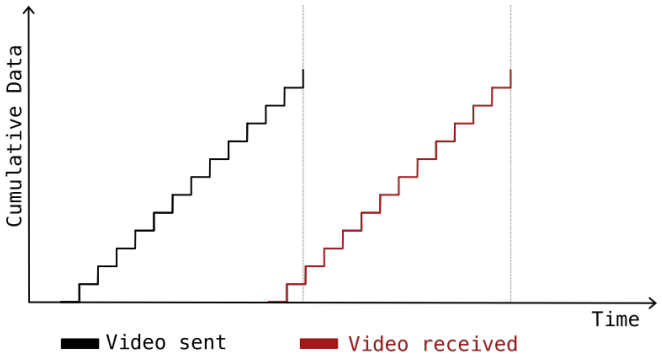



**Figure 7.10:** Data transmission rate of a simple scenario with no jitter

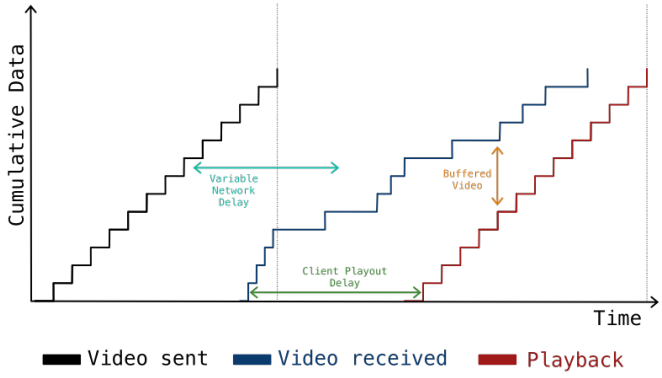



**Figure 7.11:** Data transmission rate of a scenario with jitter and delay

Some other challenges that have to be undertaken is that the client might suffer of some network delays, or that it might request to pause, fast-forward or jump through the video. In such cases packets might be lost, and

may need a re-transmission.

Whenever a video has to be streamed across the Internet, it may happen that the receiving of the video is not as linear as the sending, and it may arrive a bit later. It's for such reason that the **client-side playout delay** and the **client-side buffering video delay** exist: they add some delay from when the content reaches the client, but it ensures a better fruition of the content.

But how is a video sent over the Internet? Back in the days it was sent just as an HTML object, and whenever enough bytes were collected then a buffer would start playing the video. HTTP had a major shortcoming though: all clients receive the same encoding of the video, which was not suitable for all the clients. This led to the development of the **D**ynamic **A**daptive **S**ystem over **H**TTP (**DASH**): for such system, the video is encoded in different versions, all with their URL. Whenever the client wants to request a part of the video with a lower quality or with an higher one, it can do so just by making a HTTP GET request to a different video. The HTTP server has also a **manifest** file, which specifies the URL for each version of the video. The client always requests one chunk of video at a time, so that it can jump between the different versions whenever it needs to.

There is just one last challenge to overcome, which also was mentioned at the beginning: how can contents be delivered to almost 1 billion users simultaneously? You can't have one enormous server for all the users: if such server fails, then there is no other server to which the users can refer to, and moreover it would have too long paths to the clients and too big waiting times. The other option is then to have a network of servers, where each server stores multiple copies of videos in multiple geographically distributes sites. Such servers make the **C**ontent **D**istribution **N**etworks (**CDNs**). CDNs follow one of the following two philosophies:

- **Enter Deep**: CDN servers are placed into the access ISPs all over the world, providing multiple and smaller clusters. Akamai, a society that provides such servers, follows this philosophy;
- **Bring Home**: CDN servers are placed nearby Internet Exchange Points and are distributed in less geographical sites but are bigger clusters than the ones that follow the *Enter Deep* philosophy. Compared to the *Enter Deep* philosophy, the *Bring Home* one is less expensive on terms of maintenance.

Now, whenever a client asks for a content, it asks for the CDN nodes to provide it with the manifest, which contains the URLs for downloading the most appropriate version of the content. Depending also on the congestion of the network, a client might choose to download the content from another server.

# Chapter 8
# Transport Layer

We already saw something related to the transport layer: the TCP and the UDP protocols. And that's what the transport layer is about: providing **logical communication** between different processes from different hosts.

Let's recall how a communication between host and server works: the host fragments its message into multiple packets, and then it sends them through the socket to the server via the IP. The server, once it receives all the packets, proceeds to recompose the message, demultiplex it through the socket, process the request and then send the payload back to the host.

We recall the existence of two main protocols: **TCP** and **UDP**:

- **T**ransmission **C**ontrol **P**rotocol (**TCP**): it's a reliable, in-order delivery protocol. It has a congestion and a flow control mechanism, and it's used to set up a connection;
- **U**ser **D**atagram **P**rotocol (**UDP**): it's an unreliable, not in-order delivery protocol, and it's just basically an "extension" of the IP.

Both the protocols do not ensure that there will be no delays and that the bandwidth will be constant. There are no premises on performance.

## 8.1 Multiplexing and Demultiplexing

When using a computer, it's unlikely that it will establish only one connection at a time: in fact, it send multiple connections at once. For instance, suppose that on a computer a user is running simultaneously Firefox for exploring a Web page and on the same time is using the terminal to establish a SSH connection to another computer: how do the various connections know to which process do they belong? First, let's recall that a process may have different sockets: the connection is established not from a process to another process, but from a socket to another socket. Each socket is, we recall, denoted from an IP address and a port number.

When a client or server has to send something through the transport layer, it groups all the outgoing messages and **multiplexes** them into the network layer: similarly, **demultiplexing** happens when the payload reaches its destination.

**Definition:** Multiplexing and Demultiplexing

In general, **multiplexing** is the action of **gathering** chunks of data and **creating segments** and passing them to the network layer, while **demultiplexing** is the action of **redirecting** the data in a transport-layer message **to** the **correct socket**.

But how do demultiplexing and multiplexing work? Let's start first with demultiplexing:

1) an host receives multiple datagrams, and each one of them contains a source and destination `IP` address;

2) each datagram contains one transport-layer segment, which contains in turn a source and destination **socket port number**.

In general, while creating a socket, we must specify a **host-local port number**; each port number is a 16-bit number that ranges from 0 to 65535. All the first 1024 ports (so the ones in the range [0, 1023]) are well-known ports, which are used for only certain known services. The port number though is not enough for creating a valid datagram for any of the two protocols:

- a datagram valid for the `UDP` protocol must include the destination **port number** and the destination **IP address** (such protocol is said to be **connection-less**);
- a datagram valid for the `TCP` protocol must include the source and destination **ports numbers** and **IP address** (such protocol is **connection-oriented**).

The `UDP` protocol, when demultiplexing, it just needs the destination port number, while instead the `TCP` protocol needs also the destination and source `IP` address and port number. Moreover, `TCP` connections use **different** sockets for each client, while `UDP` connections use one socket for all the connections. Mind that multiplexing and demultiplexing happens at all the layers.

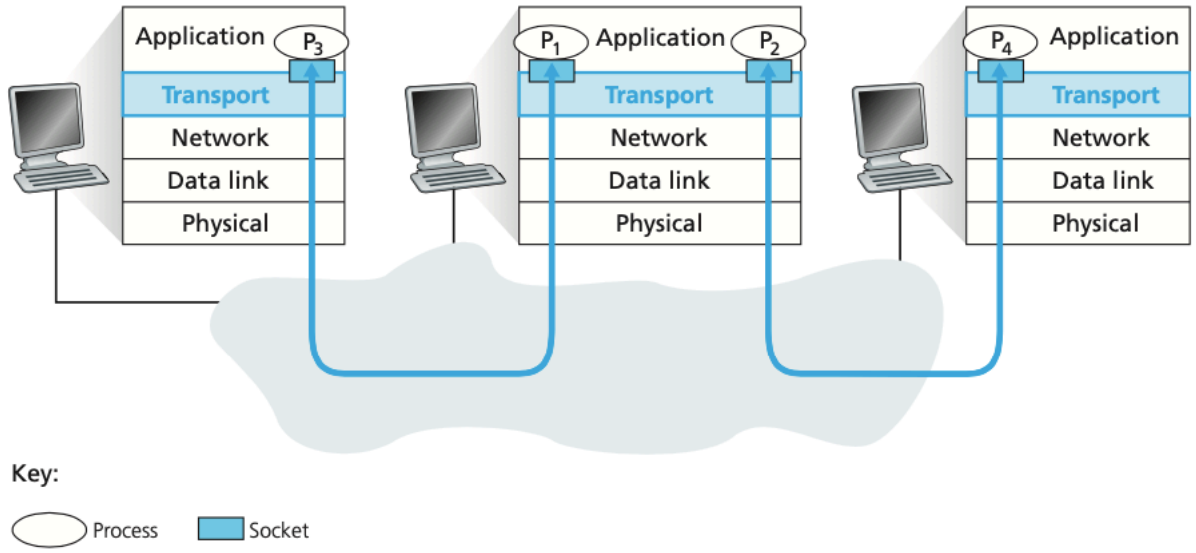



**Figure 8.1:** Example of a connection-oriented demultiplexing

Demultiplexing can be either **connection-less** or **connection-oriented**: a connection-less demultiplexing just needs the destination IP address and port number (in fact, as we mentioned earlier, the UDP protocol is said to be connection-less, since it doesn't

establish any kind of handshake), and such kind of connection is usually implemented within the UDP protocol; a connection-oriented demultiplexing needs instead not only the destination IP address and the port number, but also the source IP address and port number (as we said earlier, this type of connection is used with the TCP protocol).

## 8.2 User Datagram Protocol (UDP)

The UDP protocol adds very little to the network layer: it is a "best-effort" service, and the packets sent through it may be lost or delivered elsewhere, and we wouldn't be able to tell anything since there is no reliability. Such protocol is called **connection-less**: there is **no handshaking** between the clients, and each UDP segment is handled independently of the others.

But why is the UDP protocol still used today, given its unreliability? Firstly because it's connection-less: some applications may benefit of the speed of the connections. Moreover, the connection is very **simple**: it's direct and the headers have a smaller size. Plus, it always sends data, it doesn't wait for any congestion control mechanism to authorize the connection. UDP connections are used with various applications, such as HTTP/3, DNS, SNMP (a protocol used to control all the devices connected to an IP address) and most of the streaming platforms. If, for any of the previous applications, reliability is needed, then it can be added on the application layer.

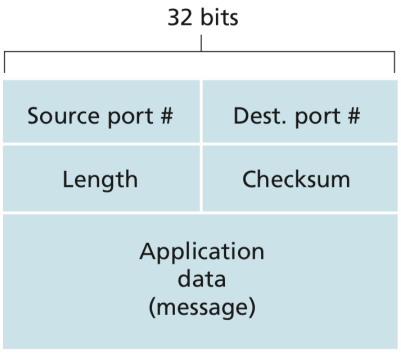



**Figure 8.2:** Structure of a UDP segment

The UDP connection header is made out of the following elements, where each element takes 2 bytes:

- source and destination **port** (total of 32 bits);
- **length** of the UDP **segment** (considering both headers and data), and it has a length of 16 bits (the length of the application data must fit then inside $2^16$ bits);
- **checksum** of the segment, which is used in order to verify whether some errors originated during the transmission of the segment.

### 8.2.1 UDP Checksum

How does the checksum work in the UDP protocol? When the sender prepares the segment, it computes the sum of all the 16-bit words in the segment, and if any overflow occurs, then such overflow gets wrapped around. The result is then stored inside the checksum field. For instance, let us have the following example:

| | | | | | | |
|---|---|---|---|---|---|---|
| $Word_1$ | 0010111010101010 | + | | Result [0;15] | 0000000100111100 | + |
| $Word_2$ | 1101001010010010 | = | $\Longrightarrow$ | Overflow (Result [16]) | 1 | = |
| Result | 10000000100111100 | | | Checksum | 0000000100111101 | |

However, the checksum algorithm implemented by UDP isn't as strong as it may seem: let us consider the following example:

| | | | | | |
|---|---|---|---|---|---|
| $\text{Word}_1$ | 0010111010110010 | + | | Result [0;15] | 0000000100111100 | + |
| $\text{Word}_2$ | 1101001010001010 | = | $\Longrightarrow$ | Overflow (Result [16]) | 1 | = |
| Result | 1000000100111100 | | | Checksum | 0000000100111101 | |

Originally $\text{Word}_1$ [3; 4] and $\text{Word}_2$ [3; 4] were inverted, respectively 01 ⇒ 10 and 10 ⇒ 01

Notice how the two results in the first and second examples are the same: this shows how potentially an error could go undetected. But why does the UDP layer implement such checksum mechanism? Because the two hosts must have a way to check whether the message is correct or not, since it's **not sure** that **each link** will implement an **error detection system,** so it must be implemented on an end-to-end basis. This is based on the **end-end principle**.

In general, UDP is a "**best-effort**" service: reliability is not a perk that is always granted, and some segments might be lost because of the weak error checking mechanism; however, it has its good sides: it's quicker to set up (since it doesn't require any handshake) and might work even when the network service is compromised. Moreover, the application layer can build on top of the UDP protocol more functionalities (such as with `HTTP/3`).

## 8.3 Principles of Reliable Data Transfer

We said that UDP is an unreliable protocol, mainly because it has **no congestion control** and **no effective** way to **check** whenever the sent data is correct or not. But how can we implement an effective and reliable data transfer? We have to implement the reliability on the transport layer, since the layer underneath (the network layer) may not be always reliable: TCP is an example of a **reliable data transfer protocol** built on top of an unreliable end-to-end layer, which is the IP protocol.

Before starting, we'll make the assumption that packets will be sent in a specific order, and if some packets will get lost then they won't be reordered. On figure 8.3, we can tear down what we call a "reliable data transfer protocol": we can see how a reliable channel is actually implemented in the transport layer upon an unreliable channel in the network layer. On the two sides (sender and receiver) we have two related functions: `rdt_send()` and `rdt_rcv()` (here `rdt` stands for reliable data transfer); both functions send data to the RDT protocol, which will then handle the transmission of the data from one end to the other.

What about the outgoing functions? In order to pass the data from the sender to the receiver, the data must flow through an unreliable channel: that's why we have `udt_send()` (where `udt` stands for unreliable data transfer). On the other side, `deliver_data()` sends the data from the transport layer to the application layer.

Based on this data, we can now build, step by step, an unidirectional reliable data transfer protocol, passing through multiple versions.

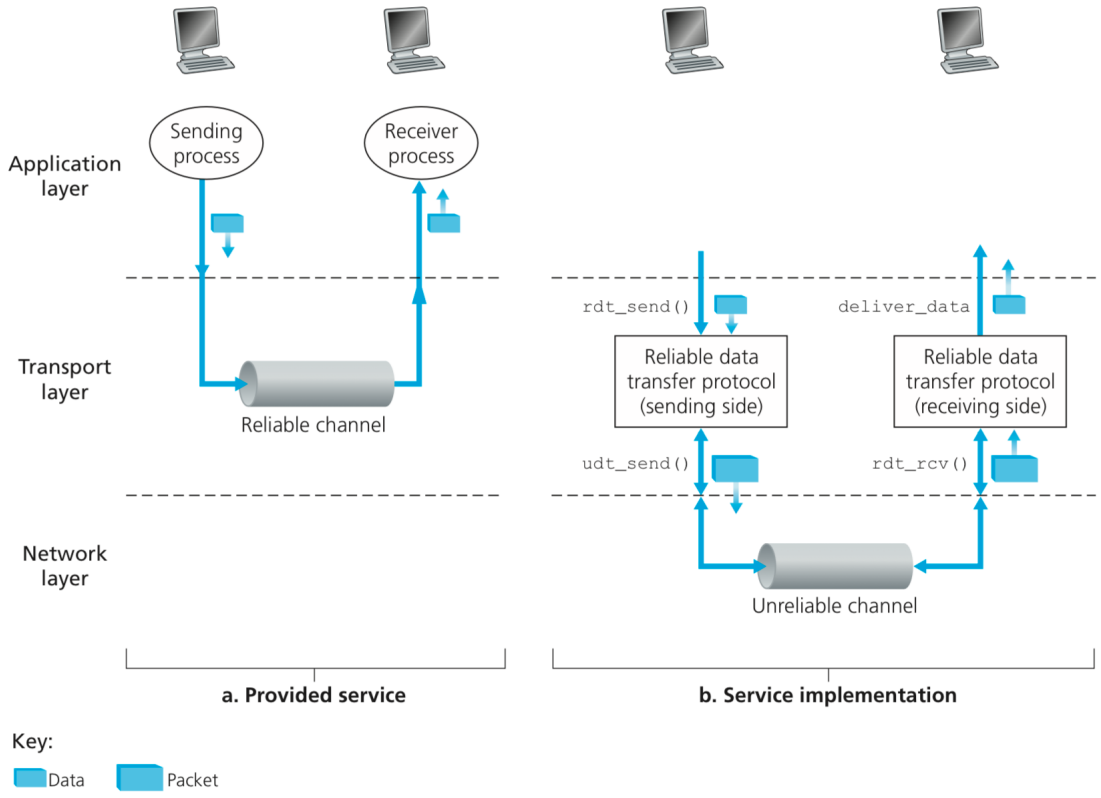



**Figure 8.3:** Example of a reliable data transfer service and the "abstraction" behind it

## 8.3.1 RDT over a Perfectly Reliable Channel (`rdt1.0`)

For this first version, we'll make an assumption: the network layer channel is **perfectly reliable**; this means that there will be **no bit losses** and **no packet losses**.

All the versions that will be explained will be related to a FSM which explains all the steps undertaken. There are different FSMs for both sender and receiver side. The initial state of the FSMs is indicated by the dashed arrow. If no action is taken upon a specific event, we'll use the $\Lambda$ (capital Lambda) symbol.

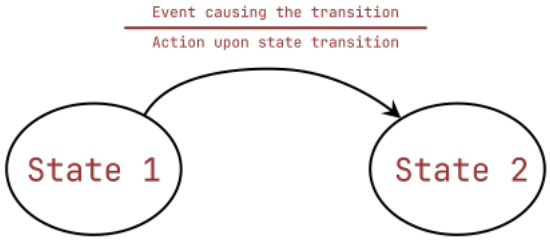
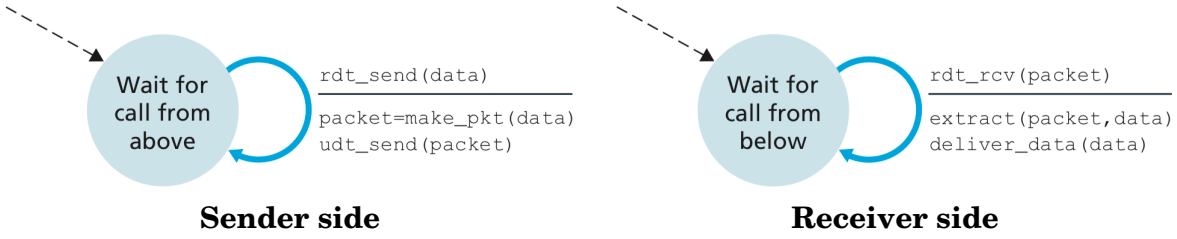



**Figure 8.4:** Legend of the FSMs

So, how does this first version work? Each side will be explained one by one:

- **Sender side**: the transport layer accepts the data (encoded as `data`) from the application layer via the `rdt_send(data)` function; upon receiving, the transport layer creates a packet (with `packet = make_pkt(data)`) and sends it to the network layer via `udt_send(packet)`. Once the packet has been sent, the protocol is again available to listen for new outgoing data;
- **Receiver side**: whenever a new packet arrives (which gets received through the `rdt_rcv(packet)` function), the transport layer extracts the data from the packet via `extract(packet, data)` and sends it to the application layer with the function

`deliver_data(data)`. The protocol then returns to the waiting state, constantly listening for any incoming packet.




**Sender side** **Receiver side**

## 8.3.2 RDT over a Channel with Bit Errors (`rdt2.0`, `rdt2.1` and `rdt2.2`)

For this version of RDT, there may be that some bits get flipped for any reason, but we still assume that all the bits are received in the same order on which they were sent. How could we deal with that situation? Let us make an analogy: when a person gives a command or an instruction to another person, the receiving person usually replies with "Ok" or "I didn't get that, could you say that again?": we have both **positive** and **negative acknowledgments** from the receiver end. If the sender receives back a positive acknowledgment then the communication ends with a positive result, otherwise if the sender receives a negative acknowledgment then it retransmits the message. Retransmission protocols based on acknowledgments are called **A**utomatic **R**epeat re**Q**uest protocols (**ARQ**).

In order to implement an ARQ protocol, we need to add three fundamental capabilities:

- **Error detection**: UDP uses checksums to check whenever a packet has been correctly sent to the other end, but what about TCP? It uses some verification systems that are implemented at the **link layer**, which require some extra bits incorporated in the segment;
- **Receiver feedback**: As we said earlier, ARQ protocols are based on an acknowledgment system: we might thus have either a positive acknowledgment (encoded as **ACK**) or a negative acknowledgment (encoded as **NAK**). Since we only have two cases for the acknowledgment value, we can use just one bit to encode them;
- **Retransmission**: whenever a **NAK** value is sent, then the sender should be able to send back to the receiver another copy of the requested data.

Again, let's analyze both the sender and receiver side separately:

- **Sender side**: at first, the sender waits for any incoming data from the application layer, and whenever some data arrives and is sent via the `rdt_send(data)` function, the protocol creates a packet `sndpkt = make_pkt(data, checksum)` (notice how both the data and the checksum are now included in the packet) and sends it with `udt_send(sndpkt)`. Now, once the packet has been sent, the sender waits for an acknowledgment (be it positive or negative). If the packet has been received and a positive acknowledgment has been sent back (`rdt_rcv(rcvpkt) && isACK(rcvpkt)`), then the sender goes back to the first state, waiting for any other incoming data; in the case where a negative acknowledgment has been sent back

(`rdt_rcv(rcvpkt) && isNAK(rcvpkt)`), then the sender sends back the packet with `udt_send(sndpkt)`. When the sender in in the second state where it's waiting for either an ACK or a NAK, it can't receive any data from the application layer. This way of listening for acknowledgments is called **stop-and-wait**;

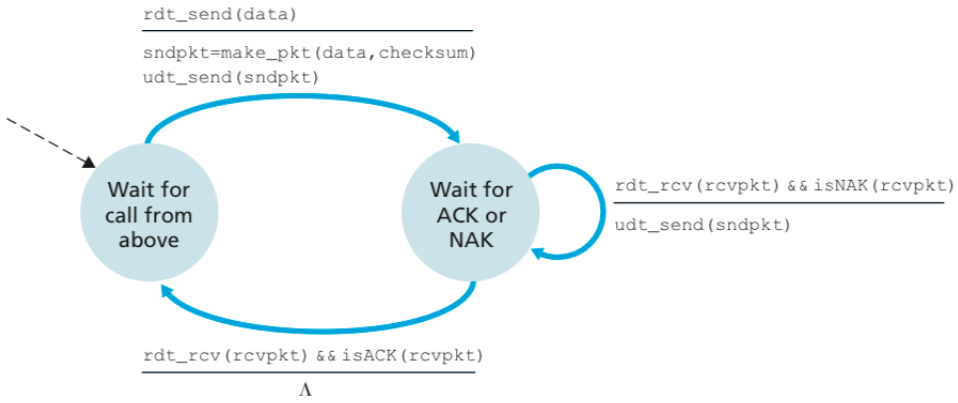



**Figure 8.5:** FSM of the sender side

- **Receiver side**: when the receiver receives a message, it checks whether the packet is corrupted or not. In the case where the packet is corrupted (`rdt_rcv(rcvpkt) && corrupt(rcvpkt)`), the protocol makes a new packet with a NAK signal (`sndpkt = make_pkt(NAK)`) and sends it back to the sender (`udt_send(sndpkt)`). In the case where the packet isn't corrupted (`rdt_rcv(data) && notcorrupt(rcvpkt)`), the protocol extracts the data (`extract(rcvpkt, data)`) and delivers it to the application layer (`deliver_data(data)`); ultimately, it makes a packet with a positive acknowledgment signal (`sndpkt = make_pkt(ACK)`) and sends it back to the sender (`udt_send(sndpkt)`).

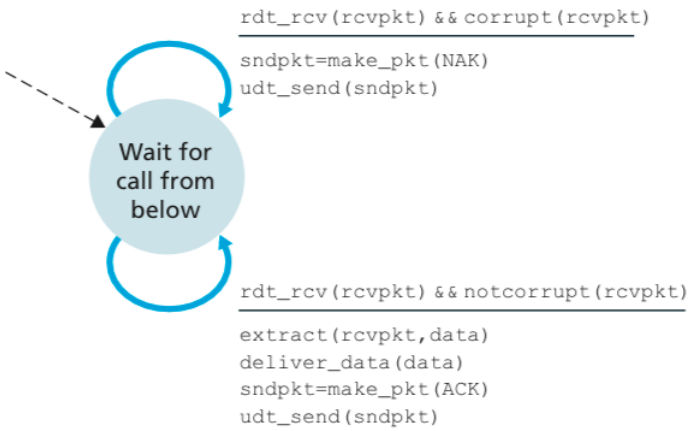



**Figure 8.6:** FSM of the receiver side

This version of the `rdt` has a major flaw: the ACK and NAK packets **might be corrupted**, so how do we handle these cases? If the sender didn't receive correctly the ACK or NAK signal, then it may **ask** back to the receiver **to send it back** another time, but what if also such request is corrupted? An **alternative** to this is to **allow** the receiver to directly **recover**, from the checksum, all the **errors** (so the receiver would be able to detect errors and correct them). A **second alternative** would be to **send again** the ACK or NAK signal. This way though we would have duplicate packets. The difficulty with duplicate packets is that the sender doesn't know a priori which kind of packet it is receiving: is it the first acknowledgment? Or is it a repetition? A simple solution would be to number the packets via a **sequence number**. For such number, we just need it to be 1 bit large: if it's 0 then it means that it's the first time that the packet was sent, otherwise with 1 it means that it's a repetition. Acknowledgment packets **don't** even need to **indicate** to which **sequence number** they

are associated to, since we are using a **stop-and-wait implementation**, which prevents both sides to send new packets until the acknowledgment returns back.

With a new version of `rdt` we can implement sequence numbers. Let's have thus a look at the two sides of a connection that use a protocol based on `rdt2.1`:

- **Sender side**: first of all, notice how the states with sequence number 0 are mirrored with respect to the states with sequence number 1: this is because the steps are basically the same for both the sequence numbers. The FSM is similar to the previous ones: the protocol receives some `data` from the application layer and creates a new packet with the **sequence number**, the **checksum** and the **data**. It then waits for an answer from the receiver: if the sender received the **acknowledgment packet** (`rcvpkt`) but either the packet is corrupted or it receives a negative acknowledgment, the receiver **sends** the **original packet again**, otherwise if the packet is sent, the packet isn't corrupted and the acknowledgment is positive, then the sender is able again to send messages;

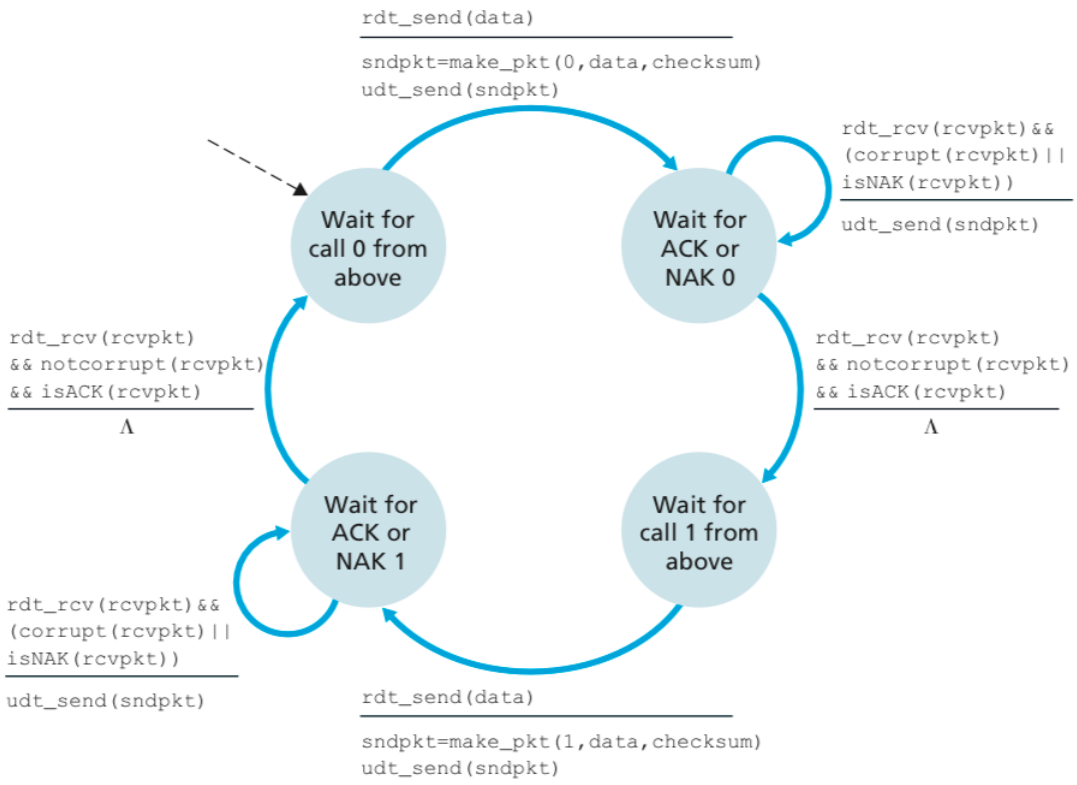



**Figure 8.7:** FSM of the sender side with the `rdt2.1` protocol

- **Receiver side**: whenever the protocol receives a packet, it proceeds to check if it's corrupted and if the sequential number is correct; if both statements are true, then an ACK signal is sent back; if the packet is corrupted instead, a NAK signal will be sent to the sender.

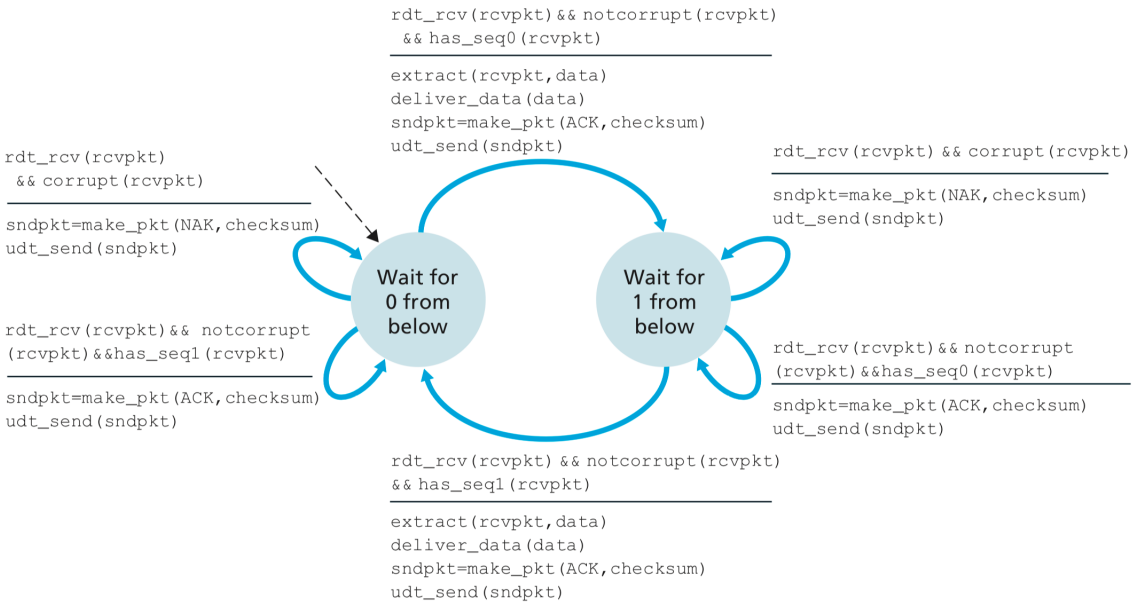



**Figure 8.8:** FSM of the receiver side with the `rdt2.1` protocol

Now, this version works, but we can make one major change: we can accomplish the same effect of a NAK signal if, instead of sending a specific packet, we send back two ACKs for the same packet (that is, a duplicate ACK). In practice, one ACK means that the receiver correctly received the packet, while two ACKs with the same sequential number mean that there was a corruption error. As a consequence, for each packet the sender and the receiver must also include the sequential number of the packet. This version is called `rdt2.2`, but won't be discussed in detail: in facts, it's identical to `rdt2.1`, except for the fact that there are no NAKs.

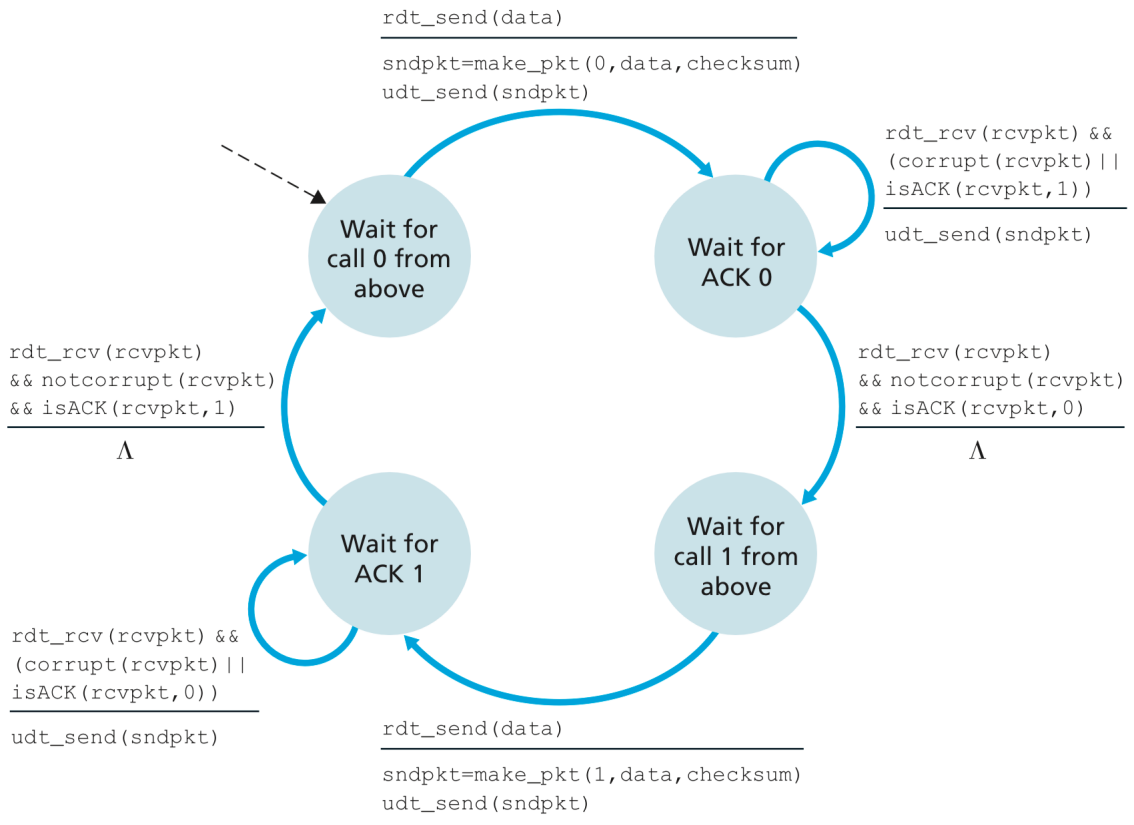



**Figure 8.9:** FSM of the sender side with the `rdt2.2` protocol

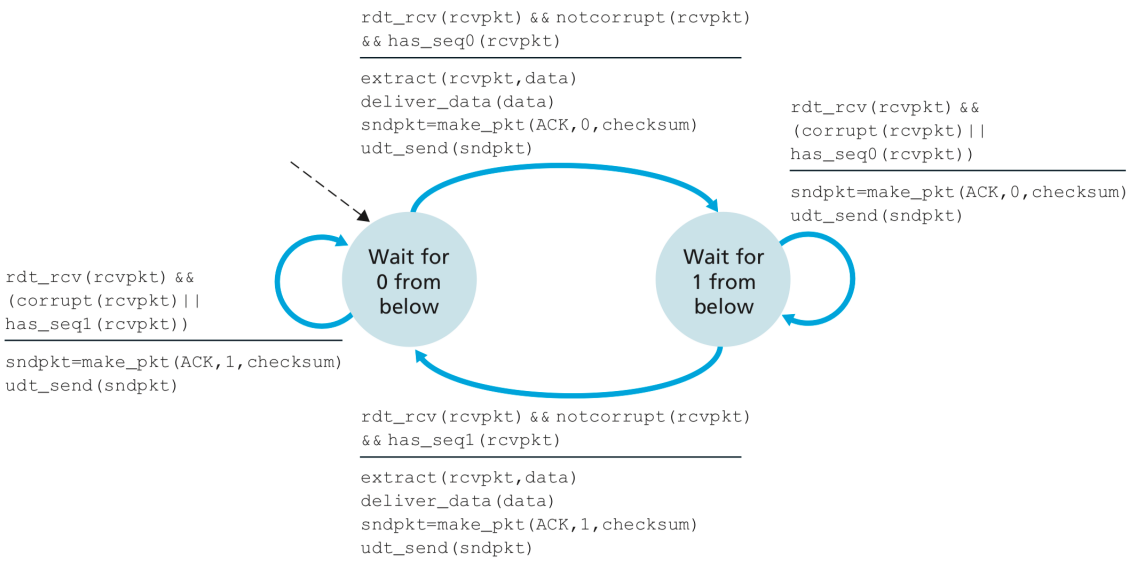



**Figure 8.10:** FSM of the receiver side with the `rdt2.2` protocol

### 8.3.3 RDT over a Lossy Channel with Bit Errors (`rdt3.0`)

For this version we make one more assumption: how can we handle the communication via a channel that might lose some bits? We must have a way to detect packet losses and to recover the lost packets. Via the ACKs and sequential numbers mechanisms we can already solve the second problem, but how do we detect if something gets lost? There are various approaches, depending on which side should take action when this happens. Supposing that the sender side should take care about it, we can think about it in the following way: if after sending the packet no acknowledgment message (either positive or negative) gets back to the sender, then we may want the sender to **wait** for a reasonable amount of time before sending back the packet.

A much more important discussion is regarding **how much** time the sender should wait: it should be at least the following:

$$t_{\text{min}} = t_{\text{round trip}} + t_{\text{data processing at receiver's end}}$$

The time **can't be too short** (otherwise slow throughput between two end points would always result in a packet loss) but it **can't be too long** either (otherwise packets would reach the other end in a too long time). Today, each "**sender**" side **establishes** a **time** for which packets should be able to be sent: if after such time the sender doesn't receive anything from the receiver, it will **retransmit** the packet. It may happen that with particularly huge delays the sender might send a duplicate of the packet, but we know that with the `rdt2.2` protocol such issue doesn't represent a threat anymore.

The sender won't retransmit forever: via the implementation of a **countdown timer**, the sender for each packet starts a timer and takes appropriate actions whenever a reply arrives from the receiver end, such as stopping the timer.

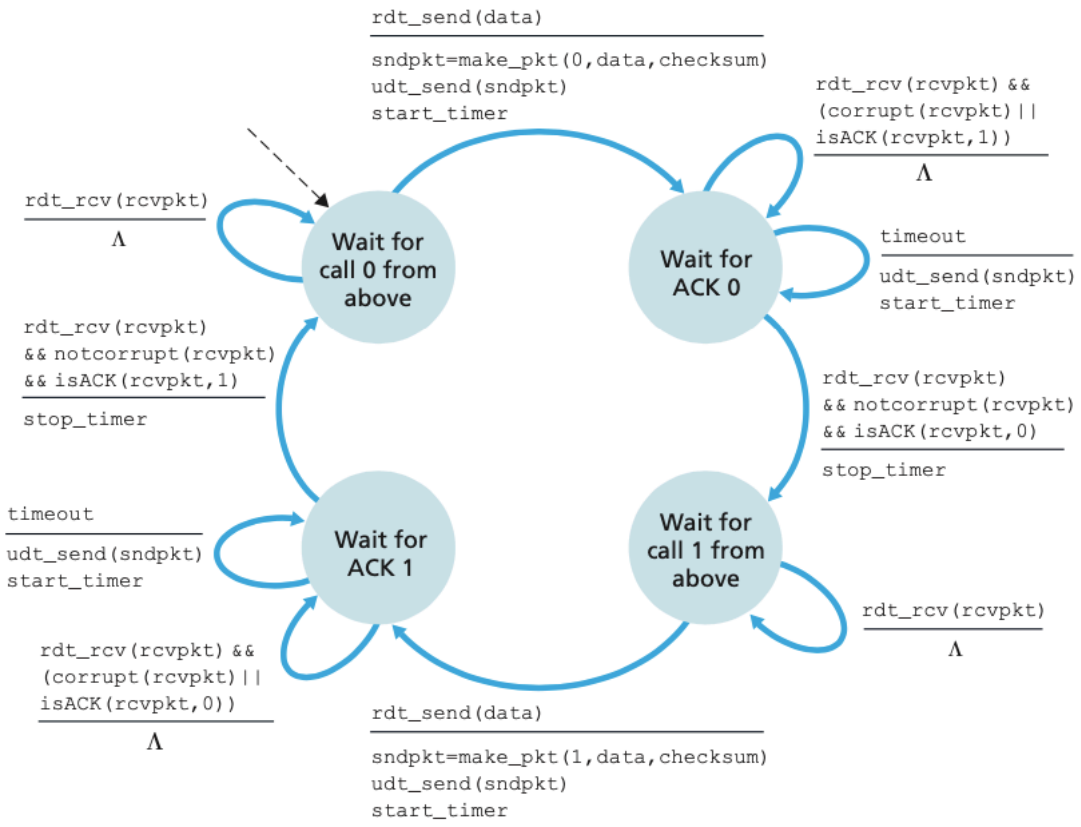



**Figure 8.11:** FSM of the sender side with the `rdt3.0` protocol

The sender side is very similar to the one of `rdt2.2`, while the receiver side basically remains the same as `rdt2.2`: the only major change is, in fact, the timeout mechanism on the sender side. This protocol is also said to be an **alternating-bit protocol**, and that's because the only sequence numbers that are allowed are 0 and 1.

## 8.4 Pipelined RDT Protocols

By itself, the `rdt3.0` protocol works fine, but it's not really performant: since it's a **stop-and-wait** protocol, we can send a new packet only after at least one round-trip (in the case where no packets get lost or there are no errors). For instance, let us make the following example: we want to send a packet from a source $A$ to a destination $B$. The link has a **RTT** of 30 milliseconds, and it sends the data through the link at the speed $R$ of 1 Gbps (so $10^9$ bits per second). We want to send a message made by multiple packets, where the size $L$ of a single packet is 1.000 bytes (thus 8.000 bits). Now, the time needed to transmit one packet from $A$ to $B$ is equal to

$$d_{\text{trans}} = \frac{L}{R} = \frac{8.000 \text{ bits / packet}}{10^9 \text{bits / sec}} = 8 \text{ microseconds}$$

This means that, if we start sending a packet at $t = 0$, then at $t = L/R = 8$ microseconds the last bit of the packet will be sent. We then wait for a RTT and, supposing that the ACK packet will have the same size and transmission time $d_{\text{trans}}$ of the original packet, then we also have to wait another $t = L/R$. In total, the sender is active for the following percent of the total time used during a connection (we'll call this number **utilization** $U$):

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30 + 0.008} = \frac{0.008}{30.008} = 0.00027 = 0,027\%$$

An utilization of the $0,027\%$ is really low, and it's not good for today's networks (consider that in our example we had a 1 Gbps bandwidth, which is somehow "wasted" on that little percentage). There is a solution to this problem: instead of operating in a stop-and-wait way, we can instead send multiple packets, as if we were filling a pipeline: in fact, this technique is called **pipelining** and allows for sending multiple packets. There are some consequences if we want to shift from a stop-and-wait to a pipelined architecture:

- the **sequence numbers range** must be **increased** to allow all the packets;
- **more packets** must be **stored** by the sender in a buffer when it waits for the ACK packets.

There are two approaches of the **pipelining architecture** that are used today: **Go-Back-N** (GBN) and **selective repeat**.

## 8.4.1 Go-Back-N (GBN)

For such protocol, there can't be more than $N$ unacknowledged packets in the pipeline. We denote with `base` the sequence number of the oldest unacknowledged packet, while with `nextseqnum` we denote the smallest (and not yet used by any packet) sequence number available. We then have, among $m$ sequence numbers, 4 possible intervals:

- `[0, base-1]`: it's the interval of all the sequence numbers that were used for already acknowledged packets;
- `[base, nextseqnum-1]`: it's the interval of all the sent but now yet acknowledged packets;
- `[nextseqnum, (base - 1) + N]`: it's the interval of usable sequence numbers that are not yet bound to any sent packet;
- `[base + N, m]`: it's the interval of not usable sequence numbers. In order to use a number of this interval, some unacknowledged packets must be acknowledged.

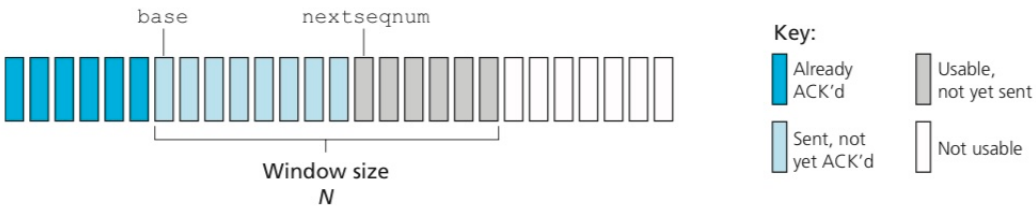



**Figure 8.12:** Sequence numbers in Go-Back-N from the sender side

The Go-Back-N protocol is also sometimes referred to as a **sliding-window protocol**, since the window of possible sequence numbers slides across all the possible values. Even $N$ is referred as **window size**. Since in the TCP protocol we have a $k$ bits field for storing the sequence number, we then allow for sequence numbers in the range $[0, 2^k - 1]$ with the Go-Back-N protocol.

In order to adapt the `rdt3.0` FSM to the Go-Back-N protocol, we need to make sure that the sender accounts for 3 main events:

- **Invocation from above**: whenever the application layer calls the GBN protocol, the sender must make sure that there are **enough** available **sequence numbers**: if not enough space is available, then the data would be sent back to the application layer (realistically, they will be stored in a buffer and sent whenever there are enough sequence numbers);
- **Receiving ACKs**: in such protocol, whenever the sender side receives the ACK packet relative to a sequence number $n$, it considers all the numbers up to $n$ to be **arrived**, and **validates all** such **ACKs**;
- **Timeouts**: with this protocol, if there is a timeout with a packet, **all** the **packets before** and up to the timed out packet will be **sent again**. This might be a problem if, of $k$ packets, all the packets up to the $k-1^{\text{th}}$ were sent: this would only create unnecessary traffic on the network. In some implementations of GBN, the timer is related to the oldest unacknowledged sent packet.

The receiver side is very simple as well: when it receives a packet $n$, if such packet is in order and has no issues, the GBN protocol sends all the data and the packets before packet $n$ (with packet $n$ included) to the application layer and sends an ACK packet to the sender side with the highest sequence number (in our case, $n$). This is possible because each packet is sent sequentially, and thus it's fine to use a **cumulative acknowledgment mechanism**.

GBN **discards out-of-order packets**, and that's because it has to deliver all the packets in order to the application layer. Even saving the packet in the buffer may not be enough: if packets $n$ gets lost, but packet $n+1$ arrives, then even if we just asked to the sender to resent packet $n$ we would inevitably get again packet $n+1$. The issue would be that eventually, in a second transmission, packet $n+1$ could be corrupted and would need a retransmission.

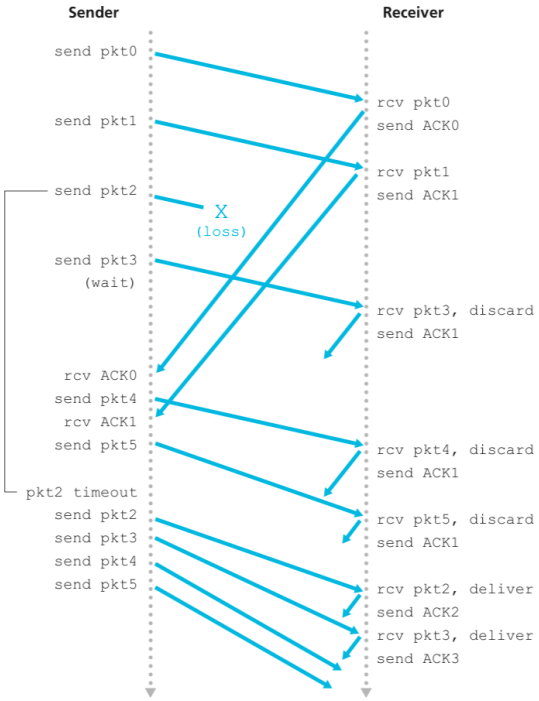



**Figure 8.13:** Example of a possible Go-Back-N implementation

## 8.4.2 Selective Repeat (SR)

The problem with GBN is that if we have a **large window size** and a large **bandwidth delay**, then GBN may end up retransmitting a **large number of packets**, and that would fill quickly the pipeline with unnecessary transmissions. Selective repeat approaches this problem by asking to the sender to **send back only** the **needed packets**, and this means that the receiver and the sender individually acknowledge each single packet.

SR still uses a window size $N$ of sequence numbers, but the receiver side acts a bit differently: it acknowledges and validates any packet that it receives, **no matter** what the **order** is. Whenever a packet is lost, then any other out-of-order packet (which is any validated packet with a sequence number greater than the one of the lost packet) is stored in a buffer: the receiver then asks to the sender to send the missing packet back and, whenever it will be validated, it will be sent together with the other packets.

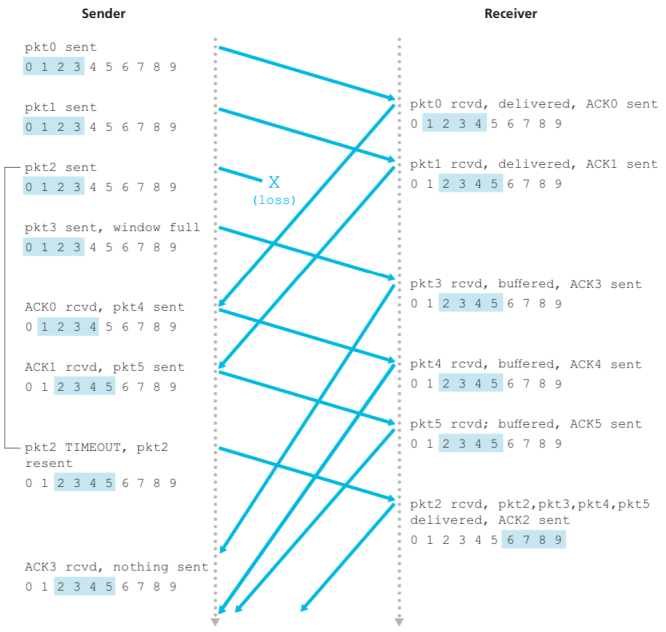



**Figure 8.14:** Example of a possible Selective Repeat implementation

It's important, if the receiver receives a packet that it's outside its window, that an ACK packet is **generated** and sent back to the sender: if this won't happen, then the sender might try forever to send back the packet, and will **never move** its **window**. This allows us to understand that the sender and receiver windows will not always be equal, but will be in most of the case **asynchronous**.

The window sizes can't even be too large: suppose that we have a window size of 3 and 4 possible sequence numbers (0, 1, 2 and 3): when packets 0, 1 and 2 get sent, the receiver sends the relative ACK packets; if for instance the three ACKs were to get lost, then the sender would send again packet 0, 1 and 2, but now the receiver's window is at 3, 0 and 1. The receiver would then consider the newly received packets 0 and 1 as **new** packets, not as the old packets that were sent back: this is a problem.

From this we can infer that there is a maximum size for the window:

$$N \leq \frac{\text{number of sequential numbers}}{2}$$

These approaches regarding `rdt` are used in the TCP protocol, but of course they do scale on the whole network. If such protocols were used on a single wire, then it would be perfect as it is, but in the reality, TCP is used on a network. The network gives the possibility in case to **reorder packets**, such that the receiver side would work as a buffer that collect packets which will "eventually" arrive in the future. We also have to **ensure** that a **sequential number** can't be used again until the sender is sure that the packet arrived at the other end. Moreover, packets won't live forever: in high-speed networks, usually packets sent via the TCP protocol have a **three minutes lifetime**.

## 8.5 Transmission Control Protocol (TCP)

As we said in the previous sections, TCP is a connection-oriented protocol: this means that before exchanging data via a TCP connection, an **handshake** must be first done be-

tween the two ends. The connection established by the TCP protocol is a logical connection: only the sender and the receiver are aware of this connection, but the intermediate links are not. TCP provides a **full-duplex service**: data can be sent in both directions at the same time; TCP is, moreover, only a **point-to-point** protocol: it's only between two end systems. This means that **multicasting** (sending of data from one host to multiple hosts) is **not allowed**.

How is a connection established in TCP? It all starts with a **three-way handshake**: whenever the sender wants to start a connection, it first connects with the receiver by sending a TCP segment; after that, the receiver will reply with another segment and finally the sender will reply one last time with the payload. Once the payload is passed from the application layer to the transport layer, it will be moved into the **send buffer**; the TCP protocol will then proceed to send the data to the receiver that will store the data inside a **receiver buffer** and then pass it to the application layer.

The maximum amount of data that can be grabbed from the sending side is called **M**aximum **S**egment **S**ize (**MSS**), which is usually determined by the largest link-layer frame that can be sent (which is called **M**aximum **T**ransmission **U**nit). The MTU must take account of the length of the TCP segment and the TCP and IP headers.

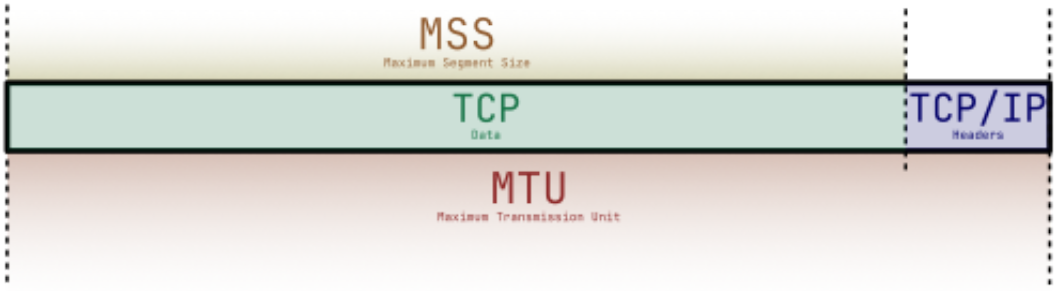



**Figure 8.15:** Visual representation of a packet sent by the TCP protocol

Each TCP segment has a TCP header, which contains various data:

- two 32-bits wide fields, one for the **sequence number** and the second for the **acknowledgment number**,
- a 16-bits wide field for the **receive window**;
- a 4-bits wide field for the **header length**;
- an optional field with an undefined width for the **options** (such as for negotiating the size of the MSS)
- a 6-bits wide field for the flags. There are thus 6 flags:
  - **ACK**: indicates if the packet is acknowledged or not;
  - **RST**, **SYN** and **FIN**: used for setting up and closing the connection;
  - **PSH**: used to tell to the TCP protocol that the data should be passed immediately to the application layer whenever they arrive;
  - **URG**: tells the protocol that the data in the payload is "urgent".

Now, how do sequence numbers and acknowledgment packets work? Since the connection with TCP is **full-duplex**, each side has its own separate window of sequence numbers. Let us make an example: host $A$ and $B$ are exchanging data, and the initial sequence number for $A$ is 8000, while for $B$ it's 12000. $A$ sends an handshake packet to $B$ with `SEQ = 8000` and, whenever $B$ will respond to $A$, it will have to use `SEQ = 12000`.

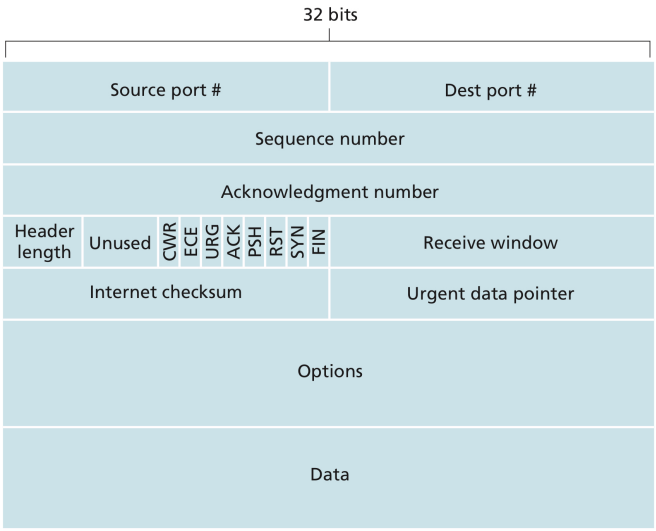



**Figure 8.16:** Visual representation of the TCP header and its fields

Differently from the `rdt3.0` sequence number mechanism, with TCP the **sequence numbers** refer to the **number** of the **first byte** in the **data field**. For instance, if the first sequence number is 700, and a segment carries 850 bytes of data, then the sequence number of the following segment will be $700 + 850 = 1550$. The acknowledgment number is the **next byte number expected** from the receiver by the sender. So for instance, if host $A$ sends to host $B$ 550 bytes of data, then the sequence number for such segment will be, supposing that the initial sequence number is 21, 21 itself. Whenever host $B$ receives all the data from host $A$, it will reply with an ACK segment that will have, as acknowledgment number, $21 + 550$, which is the last in-order received byte.

If for instance host $A$ sent three segments of 500, 200 and 300 bytes each, but for instance the second segment ($\texttt{segment}_1$) got lost, then host $B$ will receive the following (still assuming that the initial sequence number is 21):

- $\texttt{segment}_0$: `SEQ` $= 21$;
- $\texttt{segment}_2$: `SEQ` $= 21 + 500 + 200 = 721$;

Host $B$, noticing that all the bytes from 501 to 699 got lost, will send as ACK number $500 + 21 = 521$, telling host $A$ that it received only up to byte 521. TCP does it because it uses the **cumulative acknowledgment system**: the receiver side acknowledges up to the last received data byte. If a byte is missing, then TCP will acknowledge up to the **last received byte**.

There is a technique called **piggybacking**, which is used sometimes with TCP. Suppose that we're using a computer's terminal via SSH: since SSH uses TCP, we also have the usual three-way handshake. Whenever we type a letter on the keyboard, the letter is sent to the other computer. SSH, whenever we type a key on the keyboard, it echoes back what the other computer understood. This means that it not only has to acknowledge the character that we sent, but it also sends some data back. Instead of sending two separate payloads, it sends only one with both the echoed key and the acknowledgment for the packet sent by the client.

How are connections closed with the TCP protocol? Normally, a connection is persistent, even when there is no more sending of data, so there must be a way to end such connections. The **FIN** flag is used precisely for this scope. By the TCP protocol, a connection is said to be **closed** when **both** the **sender** and the **receiver** use the **FIN flag** in their messages. But what happens if only one end sends the FIN flag? Then we would have an **half-closed connection**: supposing that only the sender sent the FIN flag, then by sending it the receiver knows that the sender it's not sending anymore data via that channel, but it's still open to receive data. The connection will be closed only when also the receiver will send the FIN flag.

### 8.5.1 RTT Estimation and Timeout

Since also TCP uses the concept of timeout, how do we choose it? It can't be fixed, since the delays on the network are variable, so it must be variable. It surely can't be smaller than the RTT, since otherwise each packet would result in a timeout, and it should be somewhat bigger and not precisely equal to the RTT. But now another problem arises: how do we calculate the RTT? There is an algorithm for that:

> Let us denote with $\texttt{rtt}_i$ the **time between** the **sending** of packet $i$ and the **arrival** of its relative **ACK packet**, while with $\texttt{RTT}_i$ we denote the **estimate** of the average RTT after packet $i$. We can make an **e**xponentially **w**eighted **m**oving **a**verage (**EWMA**), which is given by the formula
>
> $$\texttt{RTT}_i = \underbrace{\alpha}_{\alpha = 0.875} \cdot \texttt{RTT}_{i-1} + (1-\alpha) \cdot \texttt{rtt}_i$$
>
> Whenever using the EWMA, we always consider that $\texttt{RTT}_0 = 0$. This average is said to be **exponential** because the weight of a given sample decays incredibly fast.
>
> What happens when we have to retransmit a packet? By **Karn's algorithm**, we don't update the average with the retransmitted packet's RTT.

So, as we said with `rdt3.0`, whenever a segment is sent, a timer is started. Whenever the timer reaches the `RTT` value given by the EWMA, then a timeout occurs, and the segment is sent again. The time waited from the timeout timer isn't precisely the `RTT` given by the EWMA, but it's a bit higher: the RTF standard suggests to make the timeout equal to the `RTT` multiplied by two. This is to allow some extra time. Such time is called **retransmission timeout interval**, and is denotes as $\texttt{RTO}_i$.

$$\texttt{SuggestedRTO}_i = 2 \cdot \texttt{RTT}_{i-1}$$

Sometimes, `RTO` deviates too much, and causes too much frequent timeouts. This is why a new and improved value of `RTO` is employed, and it's given by the Jacobson's algorithm:

$$\texttt{RTO}_i = \texttt{RTT}_{i-1} + 4 \cdot \underbrace{\texttt{MDEV}_{i-1}}_{\text{Mean deviation}}$$

Why do we use the mean deviation and not the standard deviation? Simply because it's cheaper to compute. The standard deviation looks like the following:

$$\texttt{STDDEV} = \sqrt{\sum_{i=1}^{n} \texttt{rtt}_i - \text{avg}(\texttt{rtt})} \quad \text{where } n \text{ stands for the number of sent packets}$$

The **mean deviation** (also denoted as DevRTT) looks instead like the following:

$$\texttt{MDEV}_i = (1-\rho)\cdot\texttt{MDEV}_{i-1} + \rho\cdot|\texttt{rtt}_i - \texttt{RTT}_{i-1}|$$

Here we have another EWMA, where $\rho$ is equal to 0.25. With the MDEV, the RTO is now much more accurate. Timeouts still occur, but they are less likely to happen.

We said that RTT suggests that the timeout interval (RTO) should be equal to the RTT multiplied by two, but whenever there is a large variation in the estimated RTT, that might not be enough: that's why the timeout interval in such cases it's equal to

$$\texttt{RTO}_i = \underbrace{\texttt{RTT}_i}_{\text{Estimated RTT}} + \underbrace{4\cdot\texttt{DevRTT}}_{\text{Safety margin}}$$

The problem with the timer is that, if there is a high congestion on the network, then the RTO might not be enough and could cause unnecessary timeouts. Because of Karn's algorithm though, the values of the RTT and of the MDEV can't be modified. The solution for this problem is represented by the **exponential backoff**: upon a timeout, the segment is retransmitted, and the following is recomputed:

$$\texttt{RTO}_i = 2\cdot\texttt{RTO}_{i-1}$$

By only modifying the RTO, we don't change the RTT. For the exponential backoff, the timeout interval grows exponentially for each retransmission. This is a limited form of congestion control.

TCP also employs a technique called **fast retransmit**: if the sender receives **3 ACK packets** for the **same segment**, then it **retransmits** the packets without waiting for the timeout to occur. The packet that will be retransmitted is the **unacknowledged** one with the **lowest sequence number**. This technique is not motivated by the fact that a timeout will occur, but rather is a form of NAK.

## 8.5.2 Flow Control

Until now, we always assumed that both the receiver and the sender had the same speed regarding the passing the data from one layer to the other. But what happens if the **network layer** is **faster** than the **application layer**? Since the sockets act on some intermediary buffers, the network layer would **overcrowd** the receiver's **socket buffer**. In order to avoid this, the **receiver** side should **communicate** to the **sender** side that it can digest only a certain amount of data per time period.

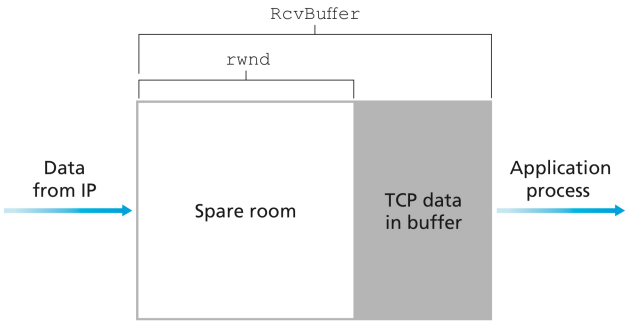



**Figure 8.17:** Representation of a receiver's buffer

In the TCP header, there is a field (called **receive window**, also written as rwnd) that tells the sender how much free space there is in the receiver's buffer, where the space is

expressed in bytes. The size of the buffer (`RcvBuffer`) is usually around 4096 bytes, even though some OSs autoadjust this value. At the receiver side, it must be always checked that the following holds:

$$\texttt{LastByteRcvd} - \texttt{LastByteRead} \leq \texttt{RcvBuffer}$$

We can also compute the value of `rwnd` with the following:

$$\texttt{rwnd} = \texttt{RcvBuffer} - (\texttt{LastByteRcvd} - \texttt{LastByteRead})$$

### 8.5.3 Connection Management and Establishment

TCP is said to provide a connection-oriented service because of the **handshake** that it instantiates between the two ends before starting to send data from one host to the other. The handshake has various properties, such as the **agreement** to **establish a connection** and the transmission of various **connection parameters**, such as the starting sequence number.

By the three handshake system, the sender first sends a request for a connection, (first handshake), then the receiver acknowledges the request (second handshake) and finally the sender acknowledges the receiver's acknowledge by sending in the meanwhile some data (third handshake). The sender is said to have an **active open** connection, while the receiver has a **passive open** connection.

How is the initial sequence number determined? By the RFC standard, it should change over time (it should, by the standard, be a 32-bits counter that increments each $4\mu s$. It completes a round in around $4.5h$). Such number is transmitted only when the `SYN` flag is active on the message. Since the first sequence number is used for the handshake, the first data bytes use the initial sequence number plus 1.

Whenever one host wants to end the connection, it has to send a message with the `FIN` flag active. Mind that the connection will still exist until **both sides** decide to **close the connection**. If only one side closes the connection, it will **still listen** for messages from the other side, but it won't be able to send other messages. Such situation is called **half-closed connection**. Each `FIN` message **must be acknowledged**, and in case the `FIN` message can be simultaneous (since the connection is **full-duplex**).

## 8.6 Congestion Control

We've seen what happens and how to tackle the situation where one host sends too much data to another host, but what happens when too many sources are sending too much data too fast so that the network can't handle the traffic? This phenomena is called **congestion** and can result in either **long queues** at the routers ports or in **packet loss** because of the routers buffers being overcrowded. Mind, this is a different scenario from the flow control problem.

Let us tackle this problem by considering, step by step, which problems we are actually facing and what do they cause:

1) Let us imagine, for this first scenario, that we have **one router** with **infinite buffers** and that the links that connect the two servers to the same router have

both a limited capacity $R$. We also have two hosts on both the sender and receiver sides: the two hosts on one side share the same link, with both of them having a transmission rate equal to $R/2$. Let us denote with $\lambda_{in}$ the transmission rate at the sender and with $\lambda_{out}$ the throughput at the receiver.

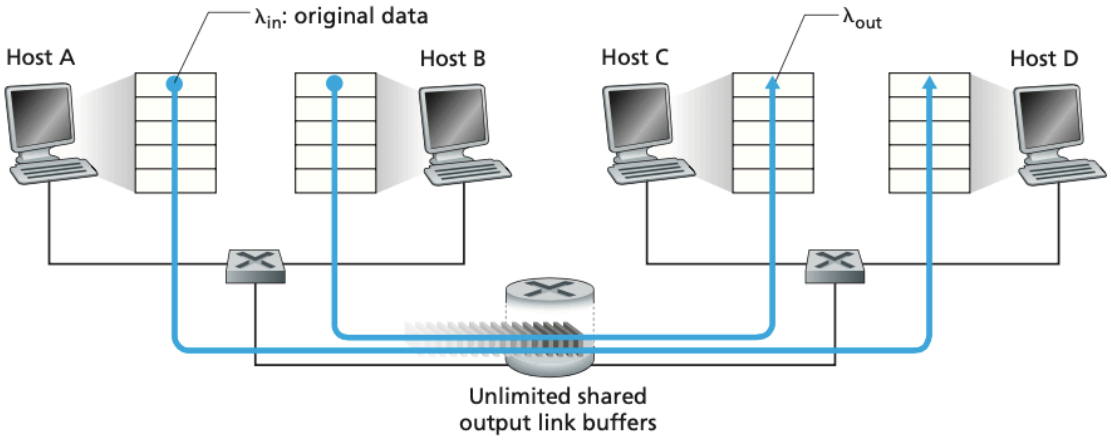



**Figure 8.18:** Scenario 1 and all its involved parts

If $\lambda_{in} \leq R/2$, then all the data can be passed at rate $\lambda_{in}$ and will arrive at the receiver with a rate $\lambda_{out} = \lambda_{in}$. If $\lambda_{in}$ becomes bigger than $R/2$, then the transmission rate $\lambda_{out}$ will be fixed to $R/2$, but there will be **delays**, since the sender is sending too much data and the link is not capient enough for sending everything.

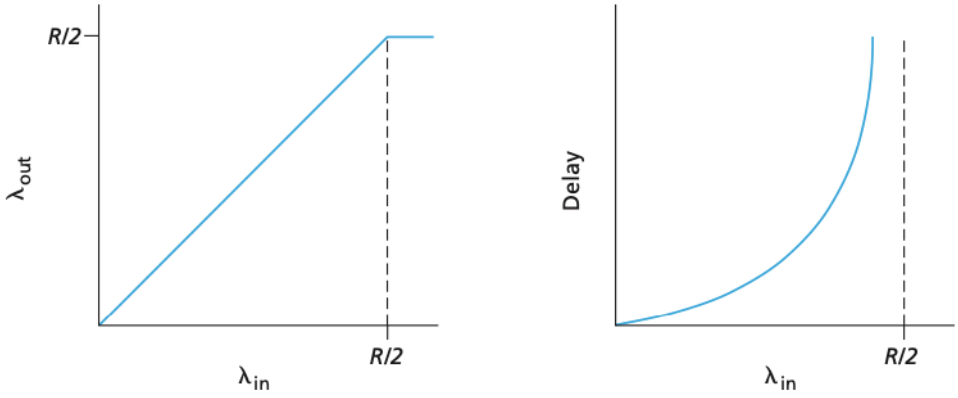



**Figure 8.19:** Scenario 1 throughput and delay

2) For this second scenario, let us imagine that there is **one router** but the buffers have **limited capacity**. Because of the limited capacity, retransmission might be necessary.

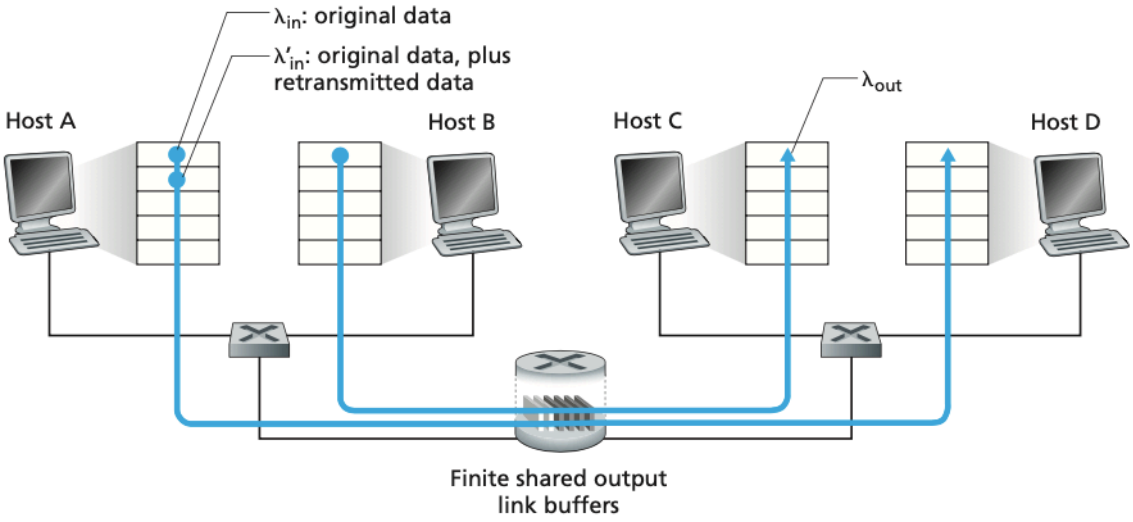
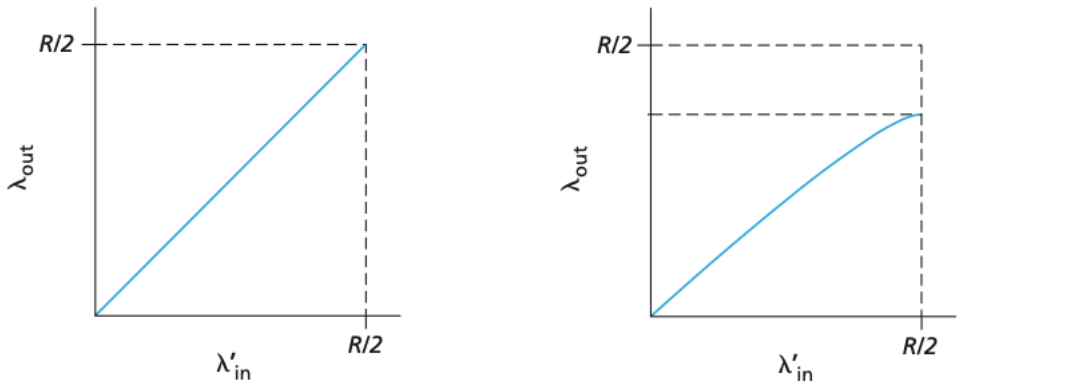



**Figure 8.20:** Scenario 2 and all its involved parts

Now, the application layer has the same transmission rate both at the receiver and at the sender side ($\lambda_{in} = \lambda_{out}$), but at the sender side, the transmission rate of the transport layer is **different** and **greater** than the one of the application layer, and that's because there is a retransmission. The application layer isn't affected because of the intermediate buffer between the two layers. We denote is as $\lambda'_{in} \geq \lambda_{in}$, where $\lambda'$ is related to the transport layer, while $\lambda$ to the application layer. Now, retransmission wouldn't be a problem with empty buffers, but realistically, buffers could be full, and some packets could be lost. This leads to a reduction of the final throughput, which has now to account for retransmissions. When $\lambda'_{in}$ reaches $R/2$, then some packets are for sure some retransmissions. It could also happen that a premature retransmission occurs, leading to an ulterior reduction of the final throughput, which has now to account for not only the needed retransmissions, but also for the un-needed ones.




Ideal condition with empty buffers

Realistic condition with necessary and non necessary retransmissions

**Figure 8.21:** Scenario 2 throughput analysis

3) For this last scenario, we consider a network where there are multiple hosts and multiple routers with finite buffers. Let us suppose that host $A$ sends too many packets and overcrowds the queue in router $R1$: any other packet incoming from host $D$ will never go through router $R1$, and will get lost.

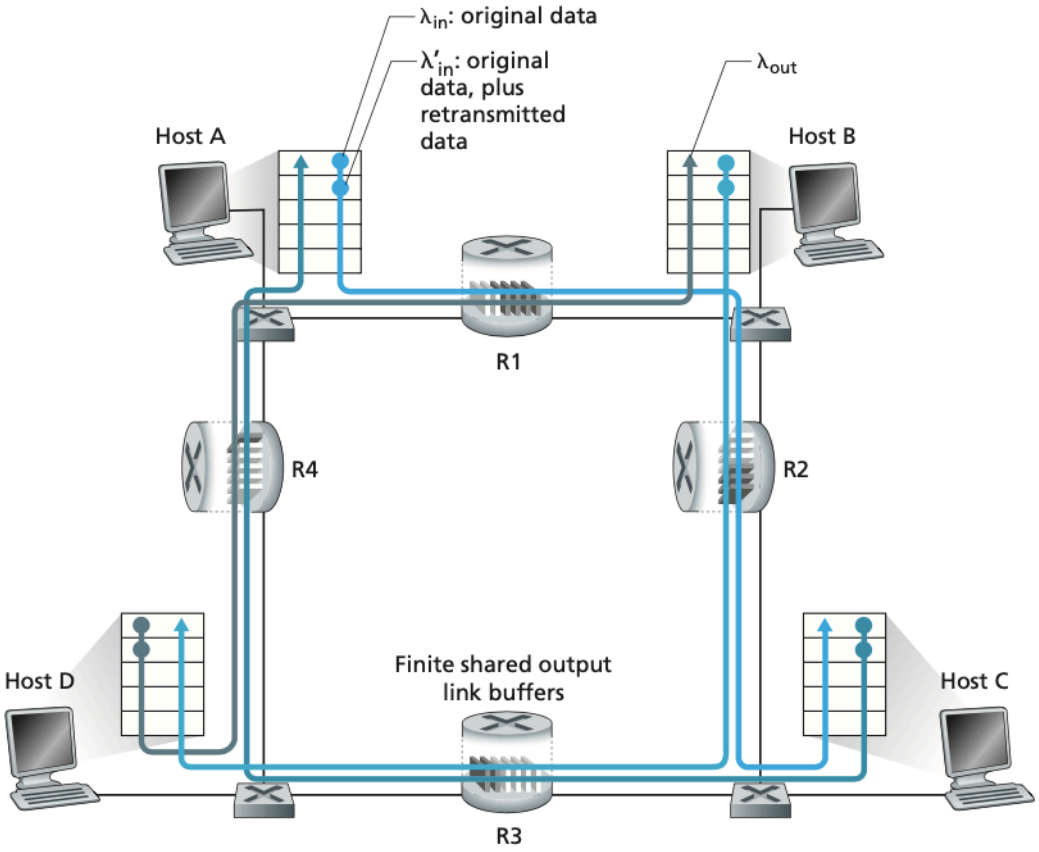



**Figure 8.22:** Scenario 3 and all its involved parts

We can think about this scenario in an asymptotic way: the more $\lambda'_{in}$ approaches $R/2$, the more the packet loss. At a certain point, when the router's buffers become too overcrowded, we don't simply lose some packets: we lose all the incoming packets. And this happens at a router like $R1$ the same way it happens to another router such as $R2$. The throughput will ultimately tend to 0 for all the hosts.

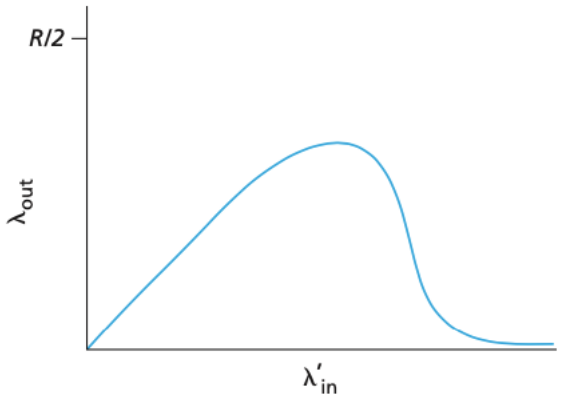



**Figure 8.23:** Scenario 3 throughput analysis

In these previous 3 scenarios we saw how congestion can be formed, and depending on it different approaches can be taken in order to reduce such congestion. There are 2 ways that allow to reduce congestion, and they are the **end-to-end congestion control** and the **network-assisted congestion control**.

The **end-to-end congestion control** mechanism deduces the congestion just by looking at the loss and delay of the packets: it has no feedback from the network at all. This is the approach used by TCP. For the **network-assisted congestion control**, the routers provide to the sending/receiving hosts direct feedback regarding the congestion level through the forwarded packets. The said routers could indicate the congestion rate or limit the transmission rate.

There are some issues with the end-to-end congestion control mechanism: how can the sender **limit the rate** and **detect congestion**? What algorithm should be used to determine the **rate limit**?

1) How can the sender limit the rate? With the use of a variable called **congestion window**, which is denoted as `cwnd`, the sender knows how much data should be sent in order to avoid congestion. `cwnd` is a variable handled by the sender that is automatically adapted based on the congestion detected by the receiver. The **sender window size** is always equal to

   $$\min(\texttt{rwnd}, \texttt{cwnd})$$

   This is because even if `cwnd` allows to transmit a huge quantity of data per time unit, it shouldn't overcrowd the receiver's buffer. At each ACK by the receiver, the sender updates its window, which is always bounded from above by $^{\text{window size}}/_{\texttt{RTT}}$;

2) How can the sender detect congestion? Congestion happens whenever a **packet is lost** and a retransmission is needed. The sender must retransmit a packet either if a **duplicate ACK** is received or if a **timeout** occurs. Duplicate ACKs are considered as a **weak** congestion indicator, while timeouts are considered as **strong** congestion indicators. So if an ACK packet arrives within a reasonable amount of time, that means that the sending rate and window can be **increased**, otherwise if a duplicate ACK arrives or a timeout occurs, then the rate and the window should be **decreased**. TCP is said to be **self-clocking** because it reacts to the receiver ACKs to determine the congestion window;

3) Which algorithm should be used to determine the sending rate? A possible approach could be to increase the rate for each correctly received ACK and decrease the rate in case a packet got lost. An algorithm called **A**dditive **I**ncrease **M**ltiplicative **D**ecrease (**AIMD**) determines that for each positive ACK the sending rate increases by one and gets halved for each lost packet. The AIMD algorithm is said to have a sawtooth behaviour because of its particular form.

The AIMD algorithm also has different versions and approaches depending on the type of loss: if a triple duplicate ACK arrives, for a version of the TCP protocol called **TCP Reno**, the sender window is cut in half; if a timeout occurs instead, the sender window is cut to the value of the MSS (this is implemented in two versions of TCP: **Tahoe** and **Reno**).

There are various methods that are used by the TCP protocol that allow for congestion control, and multiple versions of the TCP protocol were developed depending on which methods and techniques were adopted.

### 8.6.1 TCP Tahoe and Reno: Slow Start, Congestion Avoidance and Fast Recovery mode

For TCP slow start, the starting value of `cwnd` (when a connection is firstly established) is equal to 1 MSS. For each positive ACK received, the value of `cwnd` is doubled. The initial rate of the value is indeed low, but it builds up exponentially fast.

The value of `cwnd` can increase but not always exponentially: there is a threshold over which `cwnd` starts growing linearly. This threshold is denoted as `ssthresh`, and it's equal to the previous value of `cwnd` over 2. Whenever `cwnd` goes over `ssthresh`, then it starts to grow linearly. This is done to avoid congestion, and because of that the phase of linear grow is called **Congestion Avoidance**.

If a loss event occurs though, then `cwnd` starts again from 1, and `ssthresh` gets updated with value $\frac{\texttt{cwnd}}{2}$. This slow start approach is equal for both the two versions of TCP Reno and Tahoe. The difference between these two versions is that TCP Reno (the newer) threats duplicate ACKs differently from timeouts, while TCP Tahoe (the older) treats them equally.

The **Fast Recovery** mode was incorporated in the TCP protocol in a later version, called TCP Reno. This is to distinguish between a packet lost because of a triple duplicate ACK or because of a timeout. How does it work though? Whenever a **triple duplicate ACK** is received, then the TCP protocol enters in **Fast Recovery** mode: the `ssthresh` becomes again $\frac{\texttt{cwnd}}{2}$ and `cwnd` gets updated to be equal to `ssthresh` +3 MSS.

Now, for each duplicate ACK packet that is received relative to the lost segment that made TCP enter the Fast Recovery mode, then the value of `cwnd` gets **increased** by **one unit** of MSS. Whenever an **ACK** packet relative to the **previously lost packet** is received, then the TCP protocol exits the Fast Recovery mode and enters in **Congestion Avoidance** by setting `cwnd = ssthresh`. If a **timeout** occurs while TCP is in Fast Recovery mode, then the TCP starts again from scratch in **Slow Start**: `cwnd` becomes equal to 1 MSS and `ssthresh` becomes equal to $\frac{\texttt{cwnd}}{2}$.

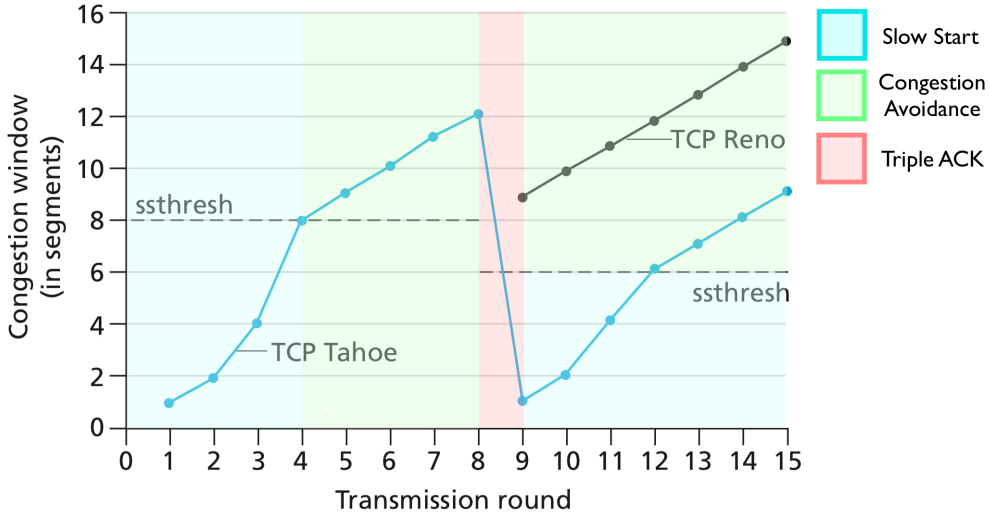



**Figure 8.24:** Differences between TCP Tahoe and Reno when a triple duplicate ACK arrives

### 8.6.2 TCP Throughput and Fairness

In order to determine the throughput, we assume a general case of the sending rate. We can **scrap** the **Slow Start phase** from the analysis, since it grows exponentially fast and **quickly recovers up** to the average rate. Let us denote with $w$ the **window size** in bytes and with $W$ the **value** of $w$ when a **loss occurs**: at any time the throughput is equal to $\frac{w}{\text{RTT}}$, but the value of $w$ will always be between $\frac{W}{2}$ and $W$. Thus we have the following

$$\frac{1}{2} \cdot \frac{W}{\text{RTT}} \leq \text{Throughput} \leq \frac{W}{\text{RTT}}$$

The average throughput will be then precisely in between the two ends:

$$\text{average throughput} = \frac{3 \cdot W}{4 \cdot \text{RTT}}$$

Is **TCP fair** among all the connections? Supposing that a link with capacity $R$ must handle $k$ TCP connections, then each connection should have a **maximum rate** of $\frac{R}{k}$. We can make an example with two competing TCP connections: we assume that both connections have the **same MSS** and **RTT** and both connections are being used to **transmit** a **large file**. Ideally, the throughput of both connections should be equal to $R$. Both connections will start with a **small sending window size**, which will linearly grow up until $\frac{R}{2}$. If the sending window size becomes too large and goes over $\frac{R}{2}$, then some loss might occur and both connection will shrink the sending window size. In general, **TCP is fair** under the **assumptions** that both connections have the **same RTT** and there is a **fixed number** of **connections** in the **Congestion Avoidance state**.

# Chapter 9
## Network Layer  Data plane

Whenever it comes down to routing a connection from an endpoint to another, we have to refer to the network layer. It is made of mainly two planes: the **data plane** and the **control plane**.

The network layer's role is simple: to redirect connections from router to router to the right endpoint. The two most important functions are:

1) **forwarding**: whenever a packet arrives to a router, then the router is responsible to **route** the **packet** to the next router, in order to make the packet reach its destination. This function is implemented at the hardware level;

2) **routing**: the network layer is responsible for routing all the packets through the network, via the use of various routing algorithms. Such algorithms are variations of the **Dijkstra's algorithm**. This function is implemented at the software level.

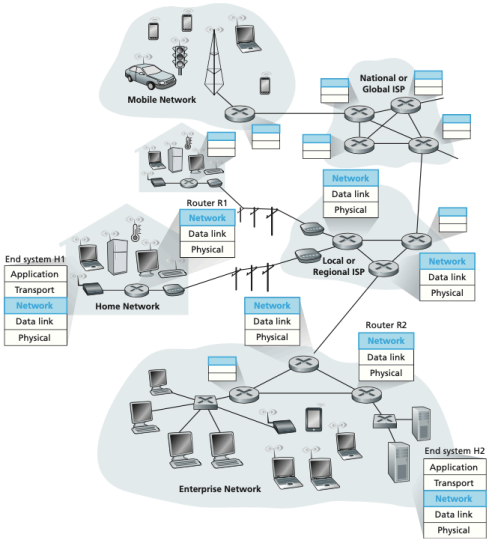



**Figure 9.1:** Example of a network

In order to implement routing, **forwarding tables** are needed. Such tables allow the router to know where to **internally** redirect packets depending on the header.

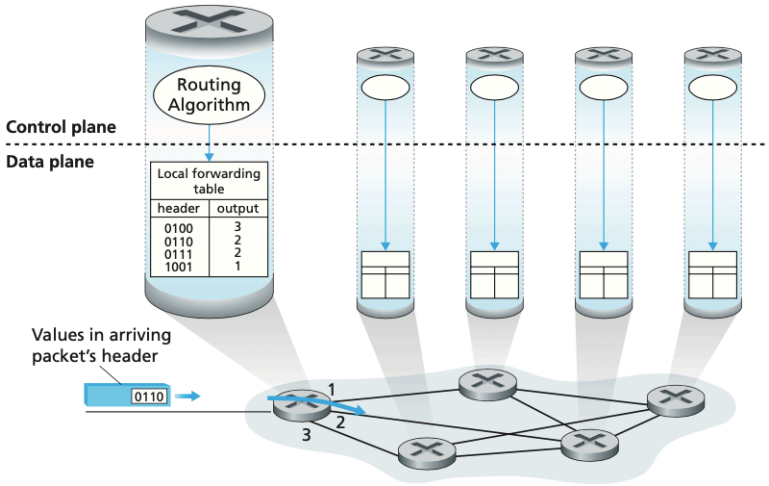



**Figure 9.2:** Visual representation on how **forwarding tables** and **routing** works

The network layer is divided into two planes: the **data** plane and the **control** plane. The **data** plane is in charge of managing all the per-router functions, and determines how a router should handle incoming packets (so how to redirect them internally within the router itself); the **control** plane instead acts on a bigger scale: it determines how a datagram should traverse the network in order to go from a source $A$ to a destination $B$. There are two control plane approaches: via **traditional routing algorithms** (which are implemented in the routers) or via **S**oftware-**D**efined **N**etworking (**SDN**, implemented in remote servers):

- **per-router control plane**: each router runs a routing algorithm with its own forwarding tables and forwarding functions;
- **Software-Defined Networking** (**SDN**): for this approach, a separate and remote controller distributes the forwarding tables to each router.

Which services would the network layer implement? We would want, in the best possible scenario, for these services to be implemented:

- **guaranteed delivery with bounded delay**: we would want a guaranteed delivery of all the datagrams, and preferably within a certain (short) time window;
- **in-order delivery**: for this service all packets should arrive to the receiver in the same order on which they were sent;
- **guaranteed minimum bandwidth**: for this service, all the packets delivered at a specified bit rate that are under a specific bandwidth are guaranteed to be delivered;
- **security**: this service would encrypt all the sent datagrams and decrypt them once the datagrams reach their destination.

The Internet's network layer provides only one service, which is called **best-effort service**: such service does not have any guarantee regarding the delivery of the datagrams, nor does it have any quality of life services. But why is it implemented this way? First, because it's a **simple mechanism**, which allowed for the quick and rapid deployment of the Internet over the world; second, a sufficient **provision** and deployment of the **bandwidth** allows for most of the real-time applications to be good enough most of the times; thirdly, **replicated application-layer distributed services** help the people to connect to various services from all over the world.

Such reasons might seem simple, but if we then give a look at the performance of the Internet stack, then we can realize how the best-effort methodology actually works really well.

## 9.1 Routers Architecture

We said that routers are very important in the network layer, but how do they work? How are they made? There are 4 main components:

- **input ports**: they perform several functions, such as performing link layer functions. Through these ports also **lookup functions** are performed, that allow the router to know where to redirect the incoming packet. Any **control packet** (packets with information regarding the forwarding tables) get forwarded directly to the **routing processor**;

- **switching fabrics**: the circuitry that connects the input ports to the output ports;
- **output ports**: these ports store the packets incoming from the switching fabrics and **send** such **packets** on the outgoing link. When the traffic is bidirectional, the output port is paired with an input port on the same line card;
- **routing processor**: it performs control-plane operations and, in traditional routers, it performs routing protocols, maintaining the routing tables and computing the forwarding tables.

Whenever a packet comes into an input port, these three functions are performed, with this exact order:

1) **physical layer**: the bits get received from the physical link;
2) **link layer**: some link layer functions are here performed;
3) **decentralized switching**: by using the header fields values, the port must use the forwarding table stored into the input port memory to route the incoming packets ("*match plus action*"). The goal is to forward them as quickly as possible. In case too many packets arrive at the same time, after the output port gets determined, they get stored in a queue.

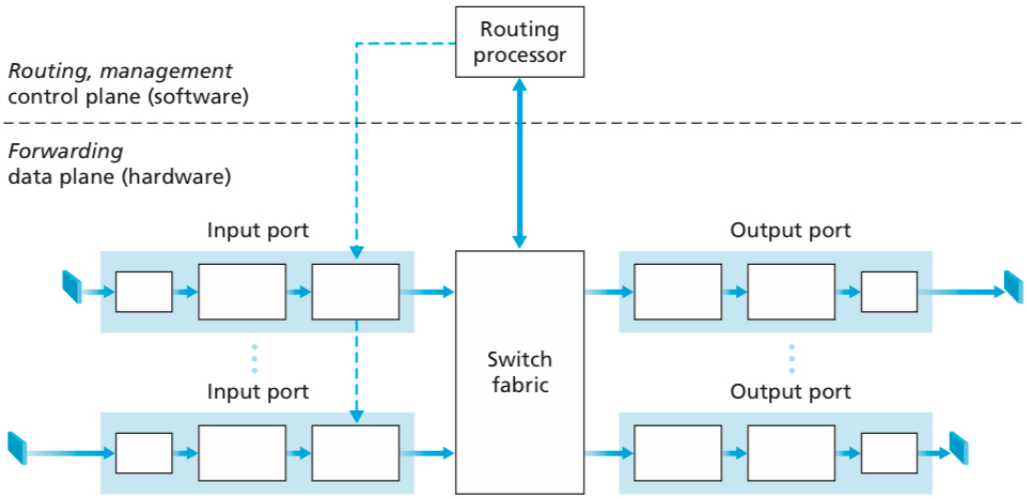



**Figure 9.3:** Architecture of a router

There are two types of forwarding techniques:

- **destination-based forwarding**: the forwarding is based only on the destination IP address (this is the traditional way;
- **generalized forwarding**: the forwarding is based on some header fields values.

## 9.1.1 Destination-Based Forwarding

As we said earlier, the forwarding is done at the input port, after the operations of the physical and link layers. Here, the router uses a forwarding table (which is either generated by the routing processor or received by a SDN controller) to redirect the packet to the right output port. Suppose for instance to have a table like this:

| Destination Address | Output Port |
|---|---|
| ... | ... |
| 11001000 00010111 00010000 00000000 | 1 |
| 11001000 00010111 00010000 00000001 | 1 |
| 11001000 00010111 00010000 00000010 | 1 |
| 11001000 00010111 00010000 00000011 | 2 |
| ... | ... |

Now, it's clearly not possible to store all the possible addresses into the router: it would take otherwise an enormous time to update all the entries, be the table computed by the routing processor or received by a SDN. We can though work by prefixes: if we **group** all the nearby addresses with the **same output port**, then we can save up some space. The forwarding table would look then something like this:

| Destination Address | Output Port |
|---|---|
| ... | ... |
| 11001000 00010111 00010*** ******** | 0 |
| 11001000 00010111 00011000 ******** | 1 |
| 11001000 00010111 00011*** ******** | 2 |
| Otherwise | 3 |

When a packet must be mapped to a port, then the router tries to match with the longest prefix. So for instance the address `11001000 00010111 00010000 00110100` would be sent to port `0`, since the prefix matches up to bit 21; the address `11001000 00010111 00011000 10110110` is sent to port `1`. Why wouldn't it be mapped to port 2? Because the matching prefix for port `1` is longer than the one for port 2.

There is a reason why we perform the longest prefix matching, but it will be explained later. Everything in the router must happen as quickly as possible, so also searching into the table the right output port must be fast operation. At the hardware level, usually embedded on-chip DRAM and SRAM are used, but in some routers **T**ernary **C**ontent **A**ccess **M**emories (**TCAM**s) are used (such kind of memory allows for the retrieval of the output port from the table in just one clock cycle, regardless of the table size). After having determined the output port, the packets get directed to the switching fabric, and in case that the switching fabric blocks them because of other packets being currently sent from the interested output port, the packets get temporarily stored in a queue.

## 9.1.2 Switching Fabrics

The switching fabric is a high-speed circuitry that must redirect packets from an input port to the relative output port. A switching fabric transfers data from one port to the other at a **switching rate**. There are three ways a switching fabric could be made:

- **Switching via Memory**: the first generation of routers used this kind of fabric, where the routers were just **simple computers**, and the redirection task was assigned to the CPU. Simply put, whenever a packet arrived at an input port, an interrupt signal would be sent to the CPU, which would copy the packet into the memory, extract the headers, look up for the address on the table and put the packet in the output port's buffer. The memory would be connected via a **single bus**, which means that only **one packet** at a time could be **forwarded**. If the bandwidth of the memory bus was $B$, then the final throughput would be less than $B/2$ (since it's $B$ for writing, $B$ for reading and some time to compute all the intermediary data);

- **Switching via a bus**: for this method, the packet would be sent from the input port to the output port via a **shared bus**. The speed of the forwarding operations would depend on the bus bandwidth (this is called **bus contention**);
- **Switching via an interconnection network**: this kind of connection was used back in the days with multiprocessor systems. A crossbar switch is a switch that connects $N$ input ports to $N$ output ports via $2N$ busses: the busses form a sort of "grid", which allows to select which data should be picked. If two packets from two different ports need to be forwarded to different output ports then that's possible: we say that crossbar switches allow for parallelism, and are thus called **non-blocking**. There is only one exception: if two packets from two different input ports must be forwarded to the same output port, then one of the two packets must wait before getting forwarded. This solution can also scale pretty easily: multiple planes can be done of this fabric, so that a better speedup can be obtained.

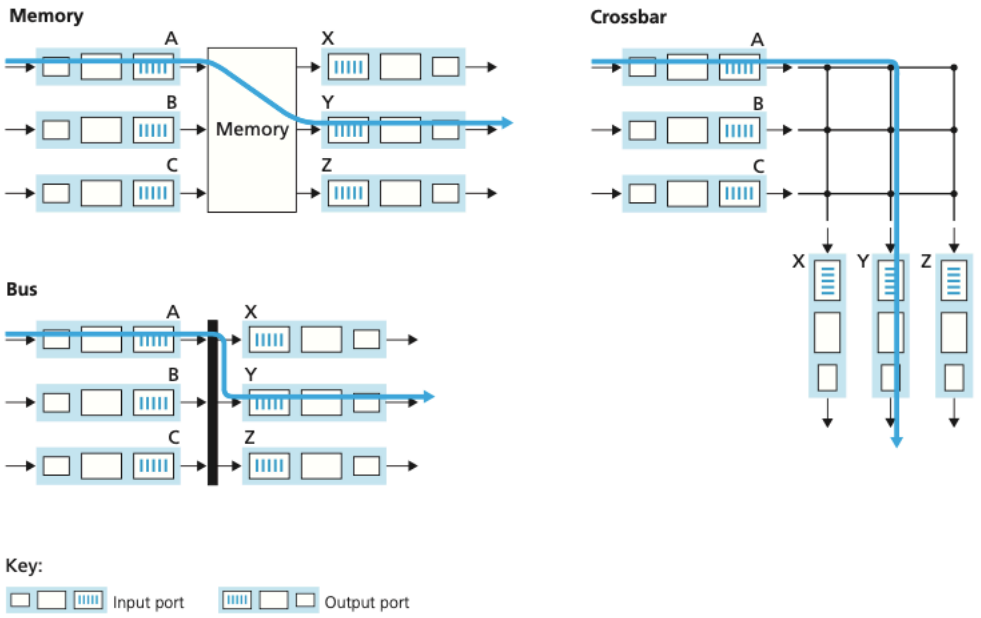



**Figure 9.4:** Different types of switching fabric

If the switching fabric is not fast enough, queues might grow at the input ports with all the incoming packets. This kind of blocking, where a datagram has to wait and prevents other datagrams to get sent, is called **head-of-the-line blocking** (**HOL blocking**). That can happen both at the input side and at the output side: there is a buffer also at the output ports in the case where some datagram might be using the output port, and if too many datagrams stack on top of the queue, then not only we get HOL blocking, but the router won't be able to store any more packets, leading to some **packet loss**. The problem then is the following: what should be done with the packet that could be lost? Do we just lose it? Or is it better to clear the queue from some packets and then send it?

In most of the cases, it's most advantageous to drop a packet before the queue is full. Such thing can be done via some packet dropping policies (called **a**ctive **q**ueue **m**anagement (**AQM**) algorithms). One of the most implemented algorithms is the **R**andom **E**arly **D**etection algorithm (**RED**). In general, such AQM algorithms act either on the **tail** of the queue or on the **priority** of the packets.

Let us consider for instance a priority scheduling algorithm: there would be multiple

queues, based on the different priorities. The output port would then pick from the queues the elements with the highest priorities first, and then the ones with lowest priority. Within a priority queue, packets are served in a FCFS logic.

Packets can also be scheduled in another way: all the packets would be schedules based on some criteria (for instance some header fields) and then a round robin algorithm would cyclically select one packet from all the queues in order to send it. Another way is to have a **w**eighted **f**air **q**ueuing (WFQ) algorithm, which is a generalization of the round robin algorithm: for such algorithm, each class has a weight, which gets updated on each cycle. The port then serves cyclically all the queues. This way there is a minimum bandwidth guarantee.

It's important to develop scheduling algorithms that allow for **network neutrality**. Such neutrality can be done under various aspects, such as the technical aspect (how should an ISP manage its resources? All the scheduling algorithms and the management of the buffers are the mechanisms), the social-economic aspect (there should be competition and innovation, and also free speech) and the legal aspect (the network should allow only for the circulation of legal contents). Many nations worldwide have different ideas regarding what is allowed on the network and whatnot. The Open Internet has a line of conduct, which can be explained with three key points:

- **no blocking**: the Internet should not block lawful content that is subject to reasonable network management;
- **no throttling**: there should be no impairment of lawful Internet traffic that makes lawful content circulate on the network;
- **no paid prioritization**: the Internet should not engage in paid prioritization.

## 9.2 Internet Protocol (IP)

The Internet Protocol is one of the most important parts in the Internet, since it allows for directing packets to specific destinations. The IP datagram format is made of various elements, but some of them are central to the functioning of the protocol:

- **Version number**: a 4 bits space where the version of the protocol is stated. Nowadays, there are 2 mostly used versions: **IPv4** and **IPv6**;
- **Header length**: since the dimension of the IP datagram is variable (because it can have a variable number of options), the length is needed in order to know where the payload starts. When an IP datagram has no options, it's usually 20 bytes wide;

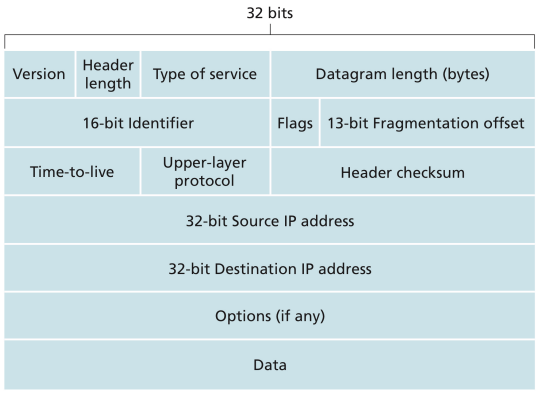



**Figure 9.5:** Visual representation of the IPv4 datagram

- **Identifier, flags and fragmentation offset**: these data spaces are used to determine the IP fragmentation. Only supported in IPv4, not in IPv6;
- **Type Of Service** (**TOS**): it allows for different types of IP datagrams, such as the ones for real-time applications;
- **Datagram length**: is the length of both header and data stored into the datagram. The length can be expressed in a 16-bits field, which means that the biggest datagram that could theoretically be sent is 65.535 bytes. In practice though, datagrams usually don't go over the 1.500 bytes;
- **Time To Live** (**TTL**): it's a field included so that datagrams won't circulate forever inside the network. Such number is decremented by 1 for each router where the datagram passes. Whenever the TTL reaches 0, then the router that will receive the datagram with TTL 0 will have to drop it;
- **Protocol**: indicates, via a number, the transport layer protocol used. For instance, 6 indicates TCP, while 17 indicates UDP;
- **Header checksum**: it's the checksum of the datagram header. The checksum is computed by taking the header and making the one's complement sum of the first 2 bytes with the next 2 bytes. The value is then stored in this field;
- **Source and destination IP addresses**: it contains the source and destination IP addresses, and sometimes the destination IP addresses is inserted upon consultancy of a DNS server;
- **Options**: it's a field of variable length used to list various options that could be relevant for the sending of the data;
- **Data** (or **payload**): finally, the core content of the packet. The payload could either be the content of a transport-layer segment or a ICMP message.

### 9.2.1 IP Addressing

How do IP addresses work? How are they assigned to each machine? By theirself, IPv4 addresses are 32-bits **identifiers**, and they each identify an interface.

**Definition:** Interface

An **interface** is a connection between a **device** and a **physical link**. Usually, routers have many interfaces, while hosts have one or two interfaces (where one is the wired interface and the second is the wireless interface)

Between a router and its connected hosts, a **subnet** is formed. We can notice the subnet from the left-most numbers of the IP address: notice how all the addresses start with the same prefix `233.1.X.X` and only the last two numbers change. There is a pattern: the 3$^{rd}$ number denotes all the devices that are connected to a port of the router (so all the devices inside the same subnet), while the last number denotes the device inside the formed subnet. But what is a subnet? Here follows a more standard definition of what a subnet is:

**Definition:** Subnet

A **subnet** (or **IP network**) is a group of devices connected between each other **without** passing through an intervening router

A common notation when defining subnets is the following:

`XXX.XXX.XXX.XXX/YY`

where `/YY` stands for the number of bits, starting from the left, that are used to identify a subnet. For instance, considering **Figure 9.6**, we have three subnets: `233.1.1.0/24`, `233.1.2.0/24` and `233.1.3.0/24`. The `0` as last number indicates **all the devices** inside the subnet. The IP address relative to a subnet also works outside the subnet itself: if `233.1.3.2/24` wants to communicate with `233.1.1.2/24`, then it can do so by using the subnet IP address.

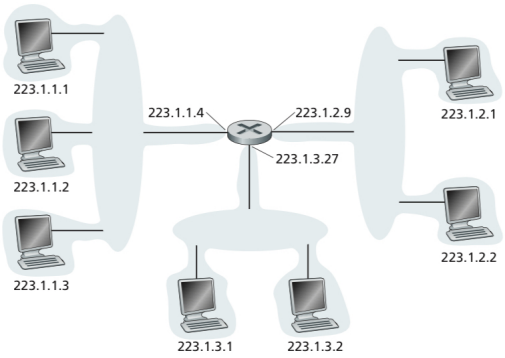



**Figure 9.6:** Example of a network with 3 subnets

There is a strategy used to address all the devices on the Internet, and it's called **C**lassless **I**nter**D**omain **R**outing (**CIDR**). For such strategy, we consider the `XXX.XXX.XXX.XXX/YY` notation, and from such address we can distinguish two parts: the **prefix** (or **network prefix**) and the **host part**. The prefix is given by the `YY` left-most bits of the address. For instance then, let us have the address `200.23.16.0/23`: the partitioning is as follows:

`200.23.16.0/23` $\Longrightarrow$ `11001000 00010111 0001000`**0** `00000000`

Network prefix | Host part

The CIDR strategy is useful because when a datagram from another subnet has to be sent to an host of a different subnet, then only the prefix part of the address is considered (so the routers only consider the first `YY` bits): this drastically reduces the sizes of the forwarding tables, since it's not necessary to have a full list of all the IP addresses. The last $32 - \texttt{YY}$ bits are then related to the host, and are necessary to distinguish the devices **within** the subnet. Before CIDR, the possible values of `YY` that could be used to define the IP addresses were only 8, 16 and 24: such scheme was called **classful addressing**, since IP address were divided in classed depending on the value used (if the value was 8 then the IP address was an A-class address, if it was 16 then we had a B-class address and lastly, with 24 we had a C-class address).

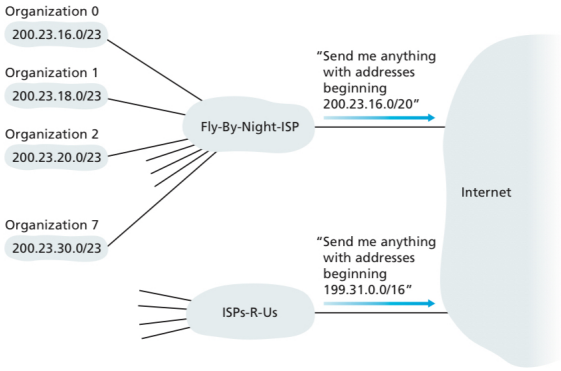



**Figure 9.7:** Example of hierarchical aggregation

Since the introduction of the Internet, most of the organizations would have a given set

of IP addresses and a custom network prefix. Each organization would refer to a specific ISP, so that the external incoming traffic would arrive by passing the following steps:

$$\text{Internet} \rightarrow \text{ISP} \rightarrow \text{Organization subnet} \rightarrow \text{Host}$$

If an organization would want to change ISP, then the new ISP would simply add the organization IP address to route all the incoming traffic. But how can a network admin or an host get its own IP address?

- Regarding a **network**: it would have to contact its managing ISP, which would have to define the network's subnet address. Then, once that happens, the network is free to handle all the host IP addresses by itself. The range of IP addresses given to the organization is indeed a subnet of the ranges inside the ISP subnet. An ISP, in order to get possess of some IP addresses, must contact the **I**nternet **C**orporation for **A**ssigned **N**ames and **N**umbers (**ICANN**);
- Regarding an **host**: it can either be hard-coded in the system by a system administrator (for instance, UNIX OSs allow to change it in the config file `/etc/rc.config`) or given by the **D**ynamic **H**ost **C**onfiguration **P**rotocol (**DHCP**), which determines the IP address from a remote server.

How does the DHCP protocol work? It basically is a client-server protocol, with the goal of dynamically assigning an IP address to all the requesting hosts. The DHCP protocol can be configured such that an host gets either a **temporary IP address** each time it connects to a specific network or a **static IP address** for whenever it joins such network. The DHCP protocol is widely employed in cases such as residential or public networks. In the best case scenario, each subnet would have its own DHCP server configured, but if that is not possible, a **DHCP relay agent** (usually a router) can contact a known DHCP server for the requesting host.

The protocol has 4 steps:

1) **DHCP discovery**: via a broadcast message, the requesting host tries to contact an eventual DHCP server. The broadcast message is sent within a UDP packet on port 67, and it's sent to the IP address `255.255.255.255`, which is the broadcast address, while the source IP address is `0.0.0.0` (since the host doesn't have a local IP address yet). If no DHCP server is present on the subnet, then a relay agent will contact a known DHCP server;
2) **DHCP offer**: whenever a DHCP server gets the message, it makes an offer with an IP address to the host. The offer message includes an **IP address**,

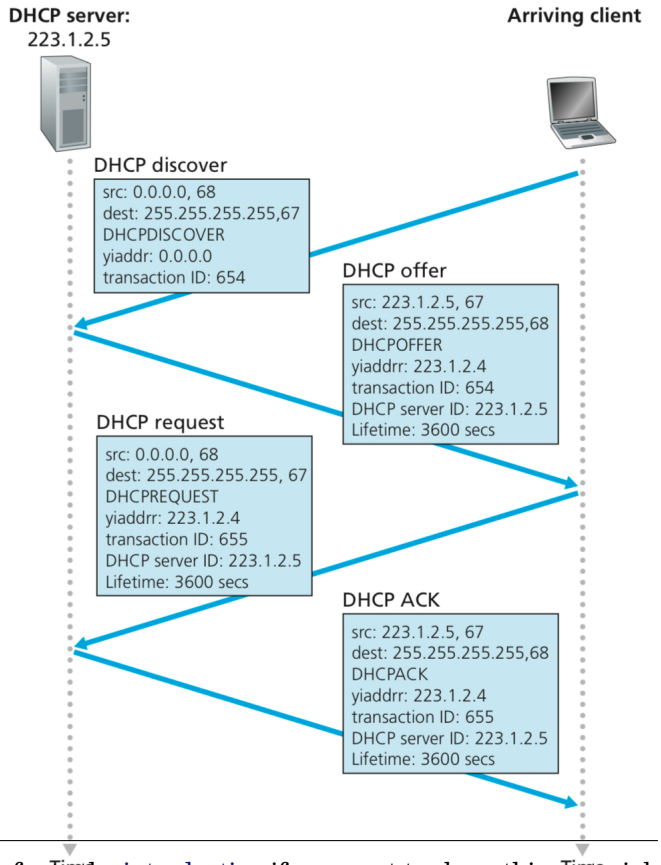



**Figure 9.8:** The 4 DHCP steps

the **lifetime** for the address (which usually ranges from several hours to several days) and the **transaction ID** (a numeric ID which is useful for the host to recognise the DHCP offer, especially in the case where multiple DHCP servers are available);

3) **DHCP request**: the host accepts the sent request from the DHCP server by replying back with the configuration parameters offered by the DHCP server;

4) **DHCP ACK**: the DHCP server acknowledges the transaction and replies with an ACK message.

The DHCP protocol does not only provide the IP address, but can also provide the address for the first-hop router, the name and IP address of a DNS server and the network mask for the subnet where the host lies in.

### 9.2.2 Network Address Translation (NAT)

In situations such as local networks, where multiple devices of all sort are connected to the Internet, it's not important (and nowadays possible) to have an IP address dedicated to each device. Since we can't anymore generate new IPv4 addresses for new devices, how should we handle the introduction of a new device inside a network? It's done via the **N**etwork **A**ddress **T**ranslation (**NAT**) approach. In order to use NAT, the router must be enabled to do that.

NAT works in the following way: the router acts as a **portal** between the outside world and the local network. Each device inside the local network gets a new **private IP address**, with one of the possible prefixes: `10/8`, `172.16/12` or `192.168/16`. Whenever a host has to make a connection with the outer world, the router registers in a **NAT translation table** the IP address of the requesting device and uses instead a public IP address to make the request. When the response from the server reaches the router, it then redirects the response to the requesting host. This is useful in local or public networks. Usually, whenever a new device joins the local network, the new private IP address is assigned via DHCP.

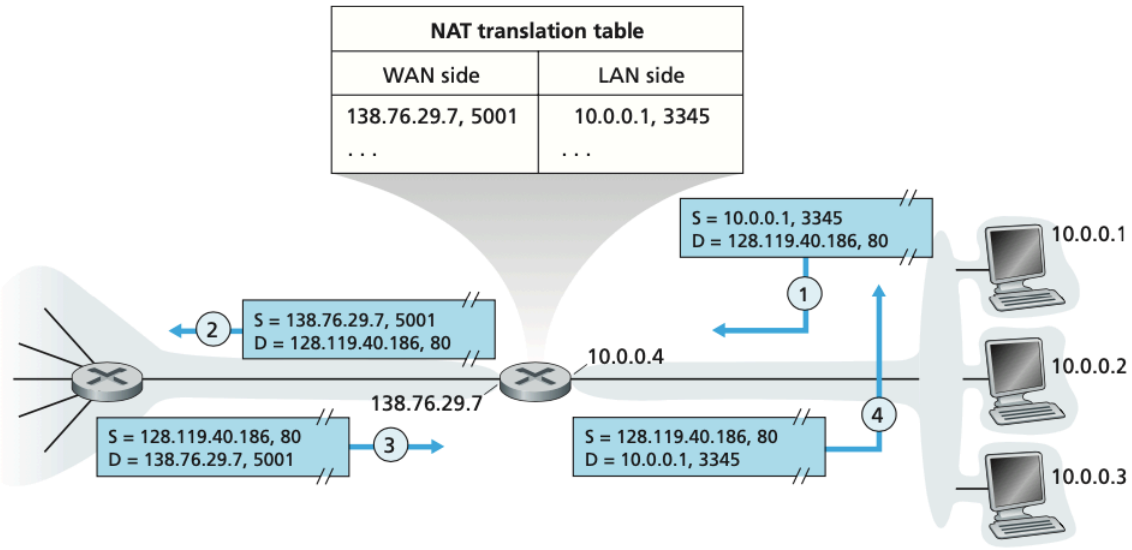


**Figure 9.9:** Example of a local network using NAT to connect to the Internet

But what if two hosts from the same local network try to access to the same server simultaneously? NAT solves this problem by assigning a **different public port number** for each host whenever a new connection is established. Of course, only the public port changes, since when a packet comes back on a certain port, then the router knows that every packet on such port is destined to a specific host.

NAT is useful for a lot of other reasons rather than just saving IP addresses: only **one IP address** is needed for an **entire subnet**, since all the other devices in the subnet will have a private address; the **hosts can change IP address** without having to notify the whole world; the **ISP can be changed** without having to change anything inside the local network; since the **devices** in a local network **can't be addresses** from the **outside world**, then this adds a layer of security.

NAT has also been very **controversial**: routers should only process **up to layer 3** (the network layer), and this is violated when replacing the **port numbers**: **ports should**, moreover, be used only to **identify processes**, **not hosts**; the address shortage should be solved, in case, with the **transition to IPv6 addresses** (where there are $2^{128}$ addresses, which roughly approximates to $3,4 \cdot 10^{38}$ addresses); NAT violates, moreover, the **end-to-end agreement**, since there is an intermediary "listener" in between the two ends. Regardless of these controversial and philosophical takes, NAT is here to stay, since it's already widely used in home and institutional contexts.

### 9.2.3 IP Fragmentation

Not all the links can carry the same size of data: some of them can carry only up to a certain amount of data, so how do we deal with bigger packets? We know from the TCP section that links have a known **MTU** (**M**aximum **T**ransmission **U**nit), and this is the key to our solution: we can fragment the packets into **fragments** and then send them all together. The reassembly process is left to the end system, since routers have one heavy task already.

The reassembly process is possible because of the **identification**, **flags** and **fragmentation offset** fields in the IPv4 datagram header: whenever a datagram is created, it has a **fixed identification number**, which is used to determine, in the case of a fragmented packet, which fragments must be put together and which not. Whenever a datagram must be fragmented, the identification number is set to the same value for all the fragments.

Since IP is an unreliable protocol, for each datagram there is a flag bit (called **M**ore **F**ragments, **MF**) which is set to 1 in the case of a fragmented packet. Such flag bit is set to 0 only for the last fragment. If the datagram doesn't need fragmentation, then the **D**on't **F**ragment (**DF**) flag bit would be set to 1. On the offset field the protocol places a value that indicates where the data inside the fragment is located (in other words, the position of the fragment). The value is computed in the following way:

$$\text{Offset} = \frac{\text{Length of the data field in a fragment}}{8} \cdot (\text{Number of fragments sent} - 1)$$

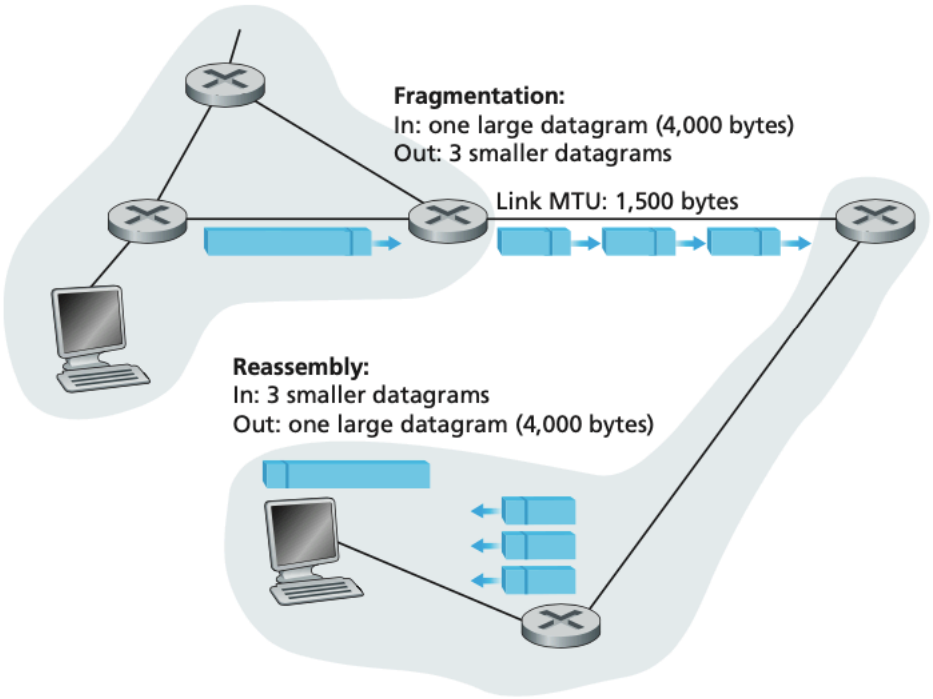



**Figure 9.10:** Visual representation of fragmentation and reassembly

### 9.2.4 IPv6 Addresses

In the 1990's, the risk of ending all the possible IPv4 addresses started to concern the **I**nternet **E**ngineering **T**ask **F**orce (**IETF**), and so the development of a new version of IP addresses begun. IPv6 is the version that was then created, and it came with various tweaks and modifications with respect to the previous version, IPv4:

- **Fixed-length header**: a new 40-bytes header was adopted, which allows for faster computations in terms of header analysis;
- **Greater address length**: with a 128-bits wide address, IPv6 can allow up to $2^{128}$ addresses, which is roughly $3,4 \cdot 10^{38}$ addresses;
- **Flow labelling**: there is not yet a precise definition of what a **flow** is, but the core concept of this function is to label packets belonging to different flows depending on the final use of the data. Such flow can be indicated in a custom field in the header;
- **No need for fragmentation from the routers**: instead of making the routers responsible for fragmentation, now this is handled **exclusively by the hosts**, and this allows for a major speed up with respect to the traffic speed. In the case where a fragmentation is needed, the router will reply to the sender with an ICMP error message such as "Packet Too Big", which will then provide to fragment the packet;
- **Removal of the checksum at the network layer**: since both TCP and the link layer perform checks on the packet for eventual corruption errors, the checksum function was removed since it was **redundant**, and this allows for a faster travel of the packet, since less computations are needed;
- **Removal of the Options field**: the option field was removed. If there is the need to specify an option, the IPv6 "Next Header" field can **point to the next header** where the option will be set. By doing so, it's possible to have a fixed-length IPv6 header.

Is it possible to transition to a full IPv6 Internet? Kind of. The biggest problem is that **IPv4 routers can't handle IPv6**, while for instance IPv6 routers are backwards compatible. There **can't be a flag day**, where the whole Internet stops for transitioning from IPv4 to IPv6: the last time that it occurred it was 35 years ago and, even if the Internet was smaller, it still was a problem.

Nowadays, a technique called **tunnelling** is used: the IPv6 datagram, whenever it has to go through an IPv4 router, gets **encapsulated** within an **IPv4 datagram** and then passed to the IPv4 router; the source and destination IP addresses in the IPv4 datagram aren't the ones of the original sender and receiver, but are respectively the ones of the **IPv6 router** at the **beginning** and at the **end** of the IPv4 routers "tunnel". Once the IPv4 datagram reaches the destination IPv6 router, then the router by looking at the protocol version (whenever the tunneling technique is used, the protocol version is 41), it understands that the tunneling technique was used, and thus proceeds to send only the IPv6 datagram contained in the payload of the IPv4 datagram. From that point on, the source and destination addresses are again the original ones.

Today, the IPv6 adoption is increasing really fast: even if only a small part of devices is actually using such protocol, it's quickly gaining adoption on all kind of devices. This teaches us though that it's really hard to change a protocol if such protocol is at the foundation of the Internet.

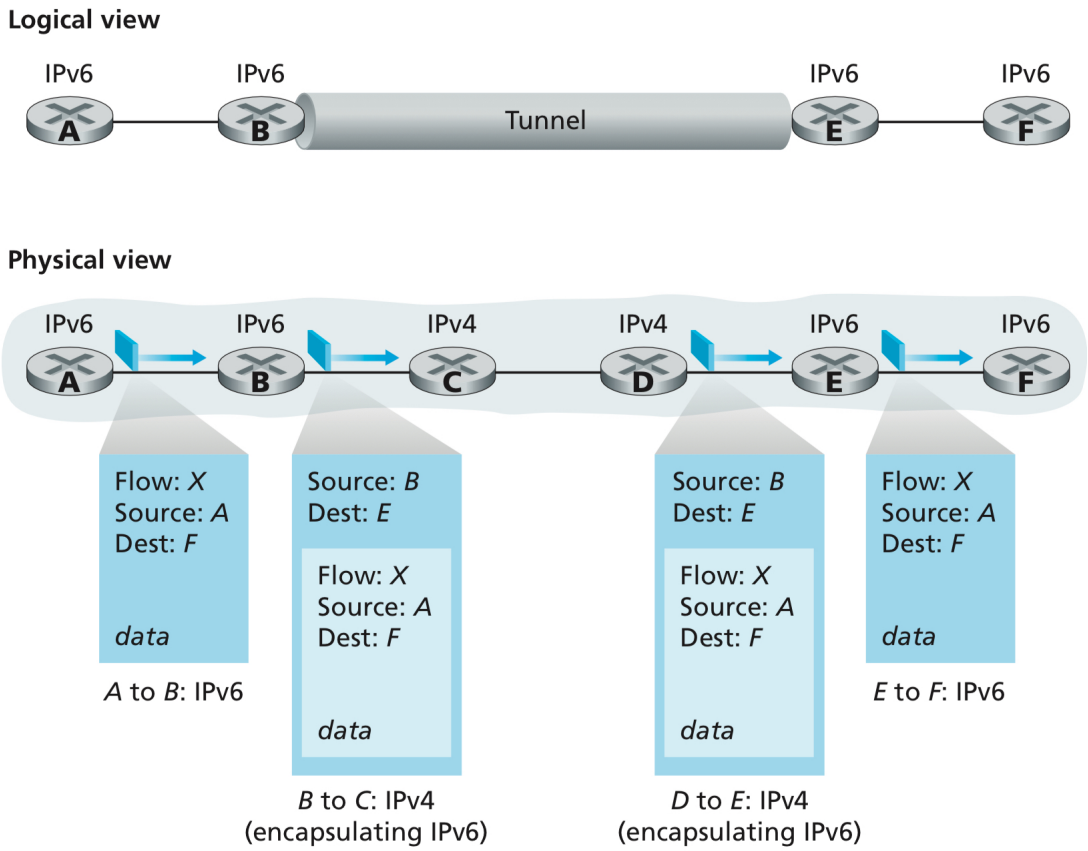



**Figure 9.11:** Visual representation of IPv6 tunnelling

# Chapter 10
## Network Layer Control plane

In the previous chapter we said that the network layer is divided into two planes: the **data** plane and the **control** plane. The control plane is responsible for the **managing** of the **routes** that the various packets must follow.

There are two approaches for structuring the control plane: the **per-router control** (which is a more traditional approach) and the **logically centralized control** (a software defined networking approach). For the **per-router control**, each router implements a routing algorithm and interacts with the other routers in the network, while for the **logically centralized control** a remote controller makes the forwarding tables and installs them among all the routers.

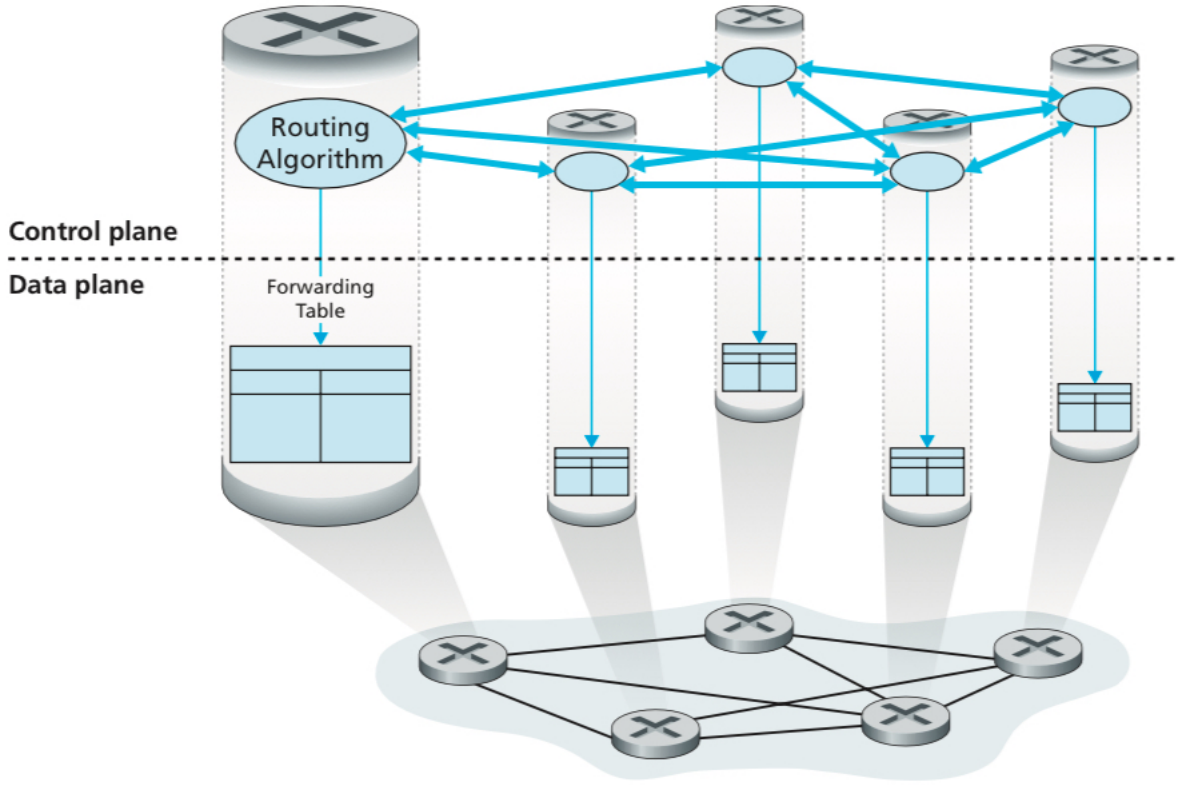


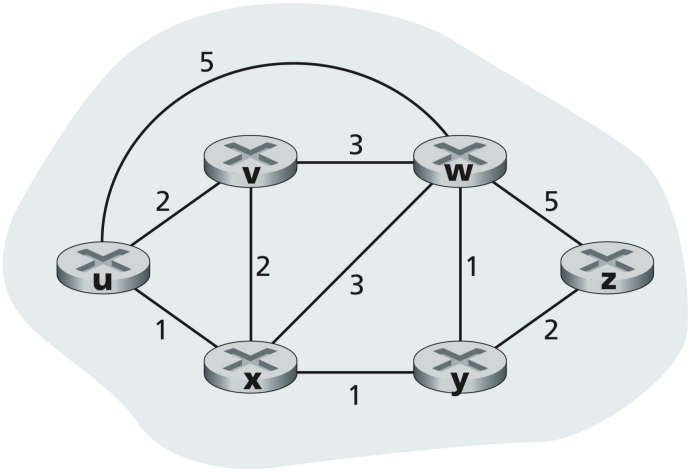

**Figure 10.1:** The network layer control plane

## 10.1 Routing Protocols

The goal of the routing protocols is to determine the most **optimal** paths to direct a packet from a source to its destination. By **path** we mean a **sequence** of traversed

routers. An **optimal path** is the path that has the **lowest cost**, that is the **fastest** among all the other paths and that is the **least congested**.

We can see the network as an **undirected graph** $G(N, E)$, where $N$ is the set of routers (the nodes) and $E$ is the set of links (the edges). Each edge has a cost that is defined by the network operator. Such cost could be for instance the bandwidth of the link. Consider the following example:




| Link | Cost |
|---|---|
| $c_{u,w}$ | 5 |
| $c_{v,w}$ | 3 |
| $c_{x,z}$ | $\infty$ |

**Figure 10.2:** Example of a network described as a graph with a little cost table

The cost between each router, if there exists a link that connects the two routers, is expresses as a normal numerical value, while if the link doesn't exist it is denoted as $\infty$.

Routing algorithms are classified depending on the following categories:

- how fast do routers change their forwarding tables? If they change it slowly over time, then they have a **static** routing algorithm, while if they change periodically based on some events (such as the increasing/decreasing of the cost of an edge) then they are said to use **dynamic** algorithms;
- do the routers act in a centralized or decentralized way? If they act in a centralized way, they are said to be **global** algorithms (an example of global algorithms are the **link state** algorithms), while if they act in a decentralized way they are said to be **decentralized** (the **distance vectors** algorithms are an example of decentralized algorithms).

## 10.1.1 Link State Algorithms

Link state algorithms are a group of **centralized** routing algorithms. A centralized routing algorithm knows the cost of **all the links** on the network, mostly thanks to some **link-state broadcasting protocols**: for such protocols, each router broadcasts the state of the various links connected to such routers.

Link-state routing protocols are based on the Dijkstra's algorithm, which allows to compute the path on a graph with the lowest cost. For the Dijkstra's algorithm (which is an iterative algorithm), after the $k^{\text{th}}$ iteration of the algorithm then the paths with the

lowest costs are known to $k$ nodes.

Before describing the algorithm, here is some of the notation used:

- $D(v)$: denotes the **current estimate** of the cost of the path from the source to the node $v$;
- $p(v)$: denotes the **previous node**, **neighbour** of $v$, along the current least-cost path from the source to the node $v$;
- $N'$: denotes the subset of nodes for which the least-cost **path** is **definitively known**.

The algorithm is the following:

```
// Initialization:
N‘ = {u};
for all nodes v {
    if v is a neighbour of u {
        D(v) = c(u, v);
    } else {
        D(v) = infinity;
    }
}

// In loop:
loop {
    find w not in N‘ such that D(w) is a minimum;
    add w to N‘;
    update D(v) for each neighbour v of w and not in N‘:
        D(v) = min(D(v), (D(w) + c(w, v)));
    /* the new cost to v is either the old cost to v or
     * the known least-path cost to w + the cost from w * to v
     */
}
```

After the algorithm executes, we can construct a forwarding table based on the shortest path. For instance, if we consider the previous network, after the execution of the algorithm, we get the following results:

| **Step** | $N'$ | $D(v)$, $p(v)$ | $D(w)$, $p(w)$ | $D(x)$, $p(x)$ | $D(y)$, $p(y)$ | $D(z)$, $p(z)$ |
|---|---|---|---|---|---|---|
| 0 | $u$ | 2, $u$ | 5, $u$ | 1, $u$ | $\infty$ | $\infty$ |
| 1 | $ux$ | 2, $u$ | 4, $u$ | | 2, $x$ | $\infty$ |
| 2 | $uxy$ | 2, $u$ | 3, $u$ | | | 4, $y$ |
| 3 | $uxyv$ | | 3, $u$ | | | 4, $y$ |
| 4 | $uxyvw$ | | | | | 4, $y$ |
| 5 | $uxyvwz$ | | | | | |

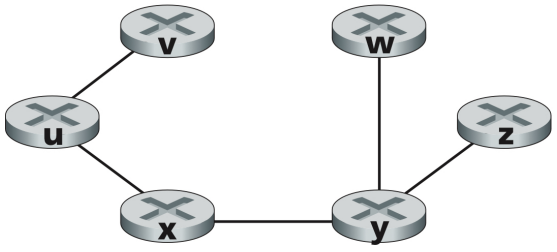



**Figure 10.3:** Shortest path from router $u$ to all the other routers

The Dijkstra's algorithm, given $n$ nodes, at each iteration must check all the nodes $w \notin N'$. It makes each time $\frac{n \cdot (n+1)}{2}$ comparisons, and takes $O(n^2)$ to map all the nodes in the network. With some other efficient implementations, it can take $O(n \cdot \log(n))$ time.

After computing its link states, each router must broadcast such states to other $n$ routers in the network, and the broadcast algorithms usually take $O(n)$ time to broadcast a message. In the worst case scenario, it could take $O(n^2)$ time to broadcast all the messages.

What happens when the link's cost depends on some factor such as the traffic volume? Some **route oscillations** could happen. Whenever a cost of a link varies though, also the least-cost path could vary: the path would have to be recomputed again then, and it would take some time.

## 10.1.2 Distance Vector Algorithms

The distance vector algorithms are **iterative**, **asynchronous** and **distributed**. They are based on the **Bellman-Ford** equation, which expresses an important relationship between the costs of the least-cost paths: consider $d_x(y)$ as the cost of the least cost path from $x$ to $y$, then:

$$d_x(y) = \min_v \{c(x, v) + d_v(y)\}$$

Readily, the cost of the least-cost path from $x$ to $y$ is equal to the **minimum** taken over **all** the **neighbours** $v$ of $x$ of the **sum** between the **direct cost** of the **link** from $x$ to a neighbour $v$ and the **estimated least-cost path** from $v$ to $y$.

The key idea of this algorithm is, from time to time, that each router **sends** to its **neighbours** its own **distance vector estimate**. When $x$ receives a new estimate from any neighbour, it **updates** its own estimate by using the Bellman-Ford equation. For this algorithm, at each time $t$, the information of a router $a$ will propagate to the routers that are on the next hop. For instance, at $t = 0$ the router $a$ computes its paths; at $t = 1$ it sends this information to all the neighbours, so all the routers that are 1 hop distant; at $t = 2$ those routers will propagate such information to the ones that are 2 hops distant from $a$, and so on and so forth. We can say that, at time $t$, the **information** got **propagated** to all the routers that are $t - t_0$ (where $t_0$ is the initial time at which a router started calculating its least-cost paths) **hops away** from the **source router**.

What happens if a link's cost **changes**? The algorithm will rerun to calculate and propagate the information, but **depending** on the **change** (if it went from an higher to a lower cost or vice versa) the **consequences change** as well. In the case where the cost when from an higher one to a lower one, the information will spread **quickly** among the routers (such event is called "**good news travel fast**").

Let us examine the following case: there are 3 routers $x$, $y$ and $z$, and each router knows the least-cost path to each other router. Suppose that the link's cost from $y$ to $x$ **increases** from 4 to 60. Router $y$ will see that router $z$ can go to $x$ with a total cost of 5, but $y$ doesn't know that it has to pass through itself. Router $y$ sets then that its shortest path is the path of $z$ plus the cost of the link from $y$ to $z$. Router $z$ then, by seeing that the path through $y$ is less costly than through the direct link with $x$, will update its path by setting the least-cost path equal to the path of $y$ plus the cost of the link between them, and so on and so forth...

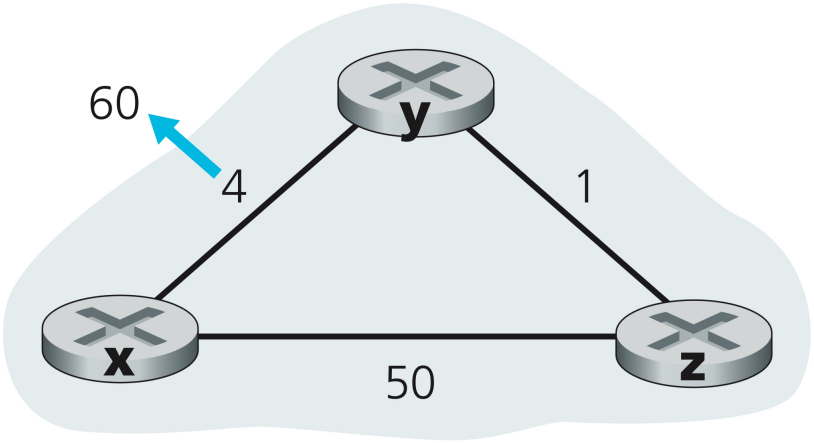



**Figure 10.4:** Example of a link changing its cost

From this previous example we can say two things: one is that **bad news travel slowly** (eventually, after a while, the two routers would settle on some fixed values) and that routers might loop the packets between them while they're updating their tables, creating **routing loops**. In order to avoid this occurrence, we can use a technique called **poisoned reverse**: for such technique, whenever a router $A$ establishes a path that goes through a router $B$ to get to router $C$, then router $A$ will tell router $B$ a white lie, which is that its cost to $C$ is equal to $\infty$. This is used also when a link changes its cost from a lower to an higher one. So how would the previous example become with poisoned reverse?

1) $y$ detects a change on the link between $x$ and $y$ from 4 to 60: it tells to $z$ that there has been a change;
2) $z$ understands that $y$ would probably instantiate a routing loop if it used $z$'s path: it uses the poisoned reverse technique and notifies $y$ that its path has a cost of $\infty$;
3) $y$ sets its path's cost to 60 (since the minimum between 60 and $\infty$ is 60) and notifies $z$;
4) $z$ sets its shortest path's cost to 50 and notifies $y$;
5) $y$ finally sets its least-cost path to the path of $z$ plus the link between $y$ and $z$ (so 51).

## 10.2 Intra and Inter-AS Routing

So far our idea of network was very **general**: all the routers are identical between each other and there weren't too many routers and possible destinations. In reality, there are billions of destinations: a routing table **can't store billions of entries**, since it would take too many space, so how can this be managed?

The Internet has a particular approach: it is structured as a network of networks, and each sub-network is administrated differently and **autonomously**. More precisely, what we called sub-networks are actually called **autonomous systems** (**AS**) or **domains**, so aggregates of routers managed by the same administrative entity. There are two types of ASs:

- **Intra-AS** (or **Intra-domain**): it denotes the routing done within the same AS (so within a network). All routers, in order to belong to the same AS, must run the same **domain protocol** (different networks can still use the same protocols though). For each AS, there is at least one **gateway router**, so a router that sits at the edge of the AS and that communicates directly with the gateway router of the other AS(s);
- **Inter-AS** (or **Inter-domain**): it denotes the routing among the various ASs. The gateway routers not only perform inter-domain routing, but also intra-domain routing.

This division between domains is used for various reasons: firstly, it allows to have **in-and-out policies**, so the admins of a domain can establish policies regarding which type of traffic goes in and out of the domain, and can easily manage the internal traffic within the domain; secondly, it's **easily scalable**: if the domain wants to expand, it can do so by just reconfiguring the internal structure and updating the other networks once it's needed (for instance to set the DNS entries); lastly, it allows for a better focus on **performance**: in the intra-domain routing the focus becomes the performance over the policy, while in the inter-domain routing there is a major focus on the policy.

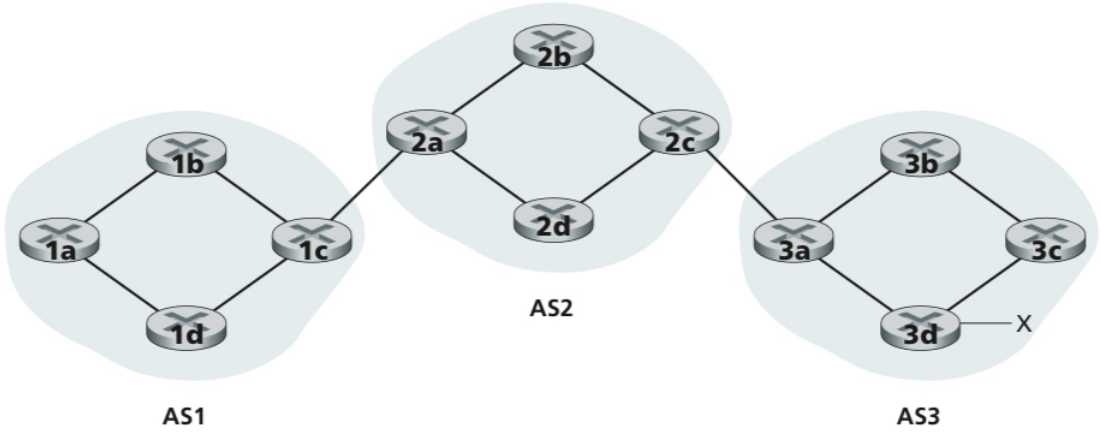



**Figure 10.5:** Example of interconnected ASs

The forwarding tables stored inside the routers must then consider the **results** of both the **intra-AS** and **inter-AS routing algorithms**. We already saw a bunch of intra-AS routing algorithms, but what should inter-AS algorithms do? They must, within an AS, **learn** which **destinations** are **reachable** from **which AS** and should **propagate** this reachability information to **all** the **routers within** its **AS**.

## 10.2.1 Intra-AS Routing Algorithms

Some examples of intra-AS routing protocols (based either on the Dijkstra's algorithm or on the distance vector algorithm) are the following:

- **R**outing **I**nformation **P**rotocol (**RIP**): it's a **distance vector-type** of algorithm, that made the routers exchange their shortest paths **each 30 seconds**. The available **edge costs** were in the range [1, 16], where 16 was the same as $\infty$; the routers would listen for **updates** on the **UDP port 520**: such updates could contain **at most 25 entries**.

  Whenever a router would start up, it would ask to its neighbours for their tables,

and it would use them to compute its own tables; once it completes to compute the tables, it would send them to its neighbours. The refresh of the tables would happen either **periodically** each 30 seconds or on the **change** of an edge's cost. There are some problems with this algorithm: it **doesn't take long to reach infinity** (in this case, 16), and it could **take minutes to stabilize** the network in the case where a router stopped working. For this algorithm, each route has a **timeout limit** of **180s** (so 3 minutes): if for 6 periods there are no updates regarding this route, then its **cost** is marked as $\infty$. Nowadays this protocol isn't widely used anymore;

- **E**nhanced **I**nterior **G**ateway **R**outing **P**rotocol (**EIGRP**): it's a distance vector-type of algorithm which was firstly developed by Cisco and then made public and open source in 2013;

- **O**pen **S**hortest **P**ath **F**irst (**OSPF**): it's a link state-type of routing algorithm, which is very similar to another famous protocol called **IS-IS**. It's an **open** algorithm and is a **classic link state** algorithm: a router **continuously floods** the other routers with its **link-state advertisements over** the **IP protocol** (rather than TCP/UDP). The cost has **different metrics**, such as bandwidth and delay. The particular side of this algorithm is that **each router** has the **full topology** of all the network, and uses the Dijkstra's algorithm to compute the forwarding table. In order to prevent malicious intrusions, the OSPF algorithm requires all the messages to be authenticated.

  The OSPF algorithm is usually employed in a **two-levels structure**: a **backbone area** and multiple **local areas**, all at the same level. The backbone area is the most "external" area of the AS, which contains the gateway routers, while the local areas are more internal. The backbone area contains some **backbone routers**, which run the OSPF algorithm only at the backbone area level. For this structure, the link state advertisements would be sent only either in the backbone area or in the local area to which the sending router belongs to.

  This algorithm has some interesting features, compared to the other ones: it **uses Dijkstra's algorithm** to construct the forwarding tables on each router, and it uses some **other additional features** while computing the tables; it allows for **equal-cost multipaths**, which is useful when managing the traffic; it **supports** for **hierarchy within** a **single AS**, so that each AS could be administrated differently depending on the needs; it **supports** for **multicast routing** via the **use** of the **MOSPF algorithm**, a version of OSPF modified to support multicast routing; finally, OSPF carries its **broadcast messages** with the **IP protocol**.

### 10.2.2 Inter-AS Routing Algorithm

While for the intra-AS routing we have the OSPF algorithm, for the inter-AS routing we have a protocol that is considered as the *de facto* inter-domain routing protocol: it's the **B**order **G**ateway **P**rotocol (**BGP**). Such protocol allows subnets to **advertise** the Internet about their **existence**, providing information such as how to reach them.

The BGP protocol allows each AS to obtain the subnet reachability information from the neighbour ASs via the **eBGP** connections, but it also allows to propagate the reachability

information within an AS via the **iBGP** connections. The protocol also allows to establish the **best routes** to the other subnets depending on the **reachability information** and on some **policies**.

For this protocol, the iBGP connections run within the **ASs**, while the eBGP ones run within the **gateway routers**. Such routers have **both connections** active simultaneously, since they are not only the means to communicate between ASs, but they are also part of their own AS, and must then be informed of every change within their AS.

The protocol sends messages over a **semi-permanent TCP connection**, and the forwarding tables uses, instead of the actual IP addresses, the **CIDR prefixes**, where each prefix denotes a **subnet**. Each entry in the forwarding tables will be in the form $(x, I)$, where $x$ is the **IP address** and $I$ is the **interface number** of one of the router's interfaces.

Let us consider the following example, where we want to build a path to the router $x$:

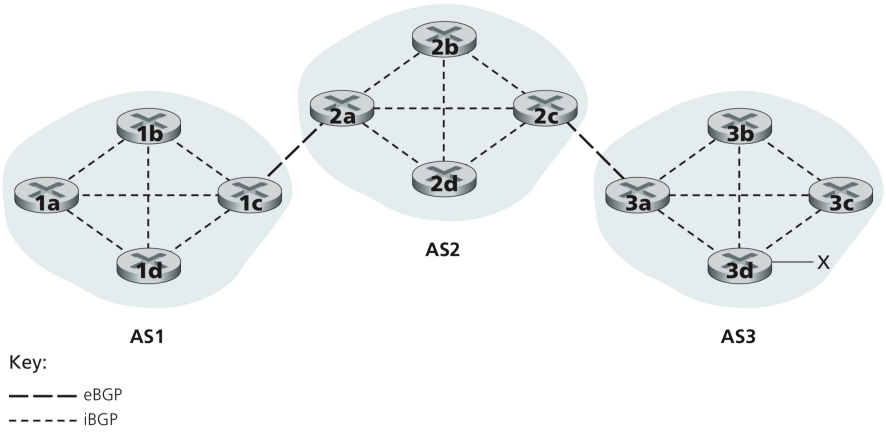



**Figure 10.6:** Example of three ASs communicating between each other

In the AS3, a **path** to $x$ is established. Now AS3 has to advertise all the other ASs about the existence of $x$. Router $3a$ will tell AS2 that $x$ exists via a message in the form `"AS3 x"`. When router $3a$ will contact the AS2 with the previous message, it promises that all the traffic directed to $x$ will be handled by AS3 and will be forwarded to $x$. AS2, whenever it will receive such message, will advertise AS1 by propagating the message to router $2a$, which will contact router $1c$ with a message of the type `"AS2 AS3 x"`.

Each of these messages is called **route**, and is made of two elements: the **destination prefix** and some **attributes**. In particular, there are two important attributes: **AS-PATH** (which is a list of all the ASs through which the route passed) and **NEXT-HOP** (which indicates the internal router in the next hop AS, so in the previous example routers $3a$ and $2a$ will be indicated in NEXT-HOP).

Let us imagine that there was a direct connection between router $1d$ and $3d$. AS3 will propagate, via the gateway routers $3a$ and $3d$, to both AS1 and AS2. AS1, based on its policies, will select one of the following two paths and will propagate it to the internal routers:

`2a; AS2 AS3; x` or `3d; AS3; x`

The BGP protocol uses 4 types of messages:

- `OPEN`: it opens a BGP connection over TCP and authenticates the two ends;
- `UPDATE`: it's used to advertise new paths or to withdraw older paths;
- `KEEPALIVE`: it keeps alive the connection whenever there are no updates, and it's also used to acknowledge the `OPEN` message;
- `NOTIFICATION`: reports errors contained in the previous message, and it's also used to close the connection.

The BGP protocol provides also some routing algorithms between routers. One of these algorithms is the **Hot Potato Routing**, which is one of the most simple ones: for this algorithm, whenever a packet has to go **from one AS to another**, it will **choose** to go through the gateway router whose **path to it** has the **lowest cost**. This algorithm doesn't care about the cost of the inter-domain links though.

## 10.3 ISP Policies

In an ideal network, all ISPs and all ASs would cooperate to transfer the packets as fast as possible, but in the real world, this is not something that always happens. It could be that, for any reason, an ISP decides not to let some traffic flow within its AS. For instance, consider the following example: there are three major ISPs and 3 customer networks, where $w$, $x$ and $y$ have a contract with the two ISPs $A$ and $C$, but not with $B$.

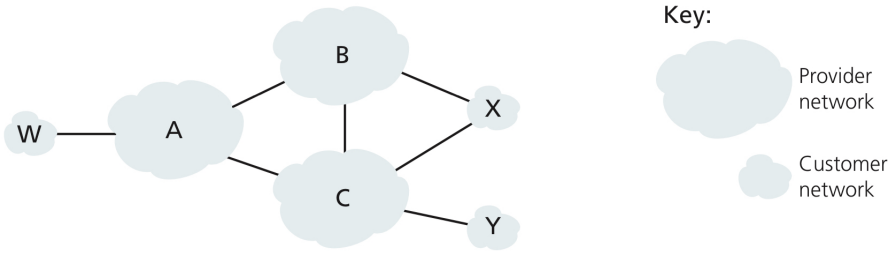



**Figure 10.7:** Example of routing policies

Suppose that $W$ is a newly opened network, so it needs to notify all the other networks about its existence. $w$ will send $A$ a message to notify the ISP about its presence, and then $A$ will notify both $B$ and $C$ with a message such as `"A, w"`. Now, ISP $B$ doesn't want to let the traffic for $w$ pass through its network, and once such message arrives it simply decides not to propagate it. $C$ will learn about the path $Aw$ and will propagate the path $CAw$, but it will never know about the existence of the path $BAw$, so it will never propagate $CBAw$. As a little side note, while $w$, $x$ and $y$ are called **customers** and $A$, $B$ and $C$ are called **provider networks**, $x$ is said to be **dual-homed**, since it's attached to two provider networks.

Lastly, how does a router select a route whenever multiple routes are available? There are multiple ways:

1) depending about a **local preference value attribute**, such as per policy decisions;
2) depending on the shortest **AS-PATH**;
3) depending on the closest **NEXT-HOP** router (as per the **Hot Potato Algorithm**);
4) depending on additional criteria.

# Chapter 11
## Link Layer

The link layer is the 4th layer of the Internet stack, and it has the task of **sending** the datagrams. It also provides some **error detection** and **correction** (together with the transport and application layer) and gives the possibility of broadcasting messages.

For the link layer, there is no difference between hosts and routers: they are all called **nodes**. In the same way, the communication channel between two nodes is called **link**, no matter which type of link it is. The link layer encapsulates each datagram in a **frame**.

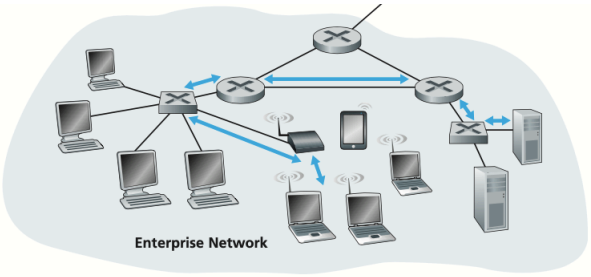



**Figure 11.1:** Example of links between nodes

Each link protocol does not provide the same services (so for instance one protocol may be more reliable than others regarding the completion of the data that got sent). The services provided by the link layer are:

- **Framing**: is the action of transforming a datagram into a frame. The frame contains various headers related to the link layer, which may change from protocol to protocol;
- **Link access**: in order to specify how the frame should be sent across a link, a **M**edium **A**ccess **C**ontrol (**MAC**) protocol is used; such protocol specifies such rules. This is useful whenever there is a shared resource that asks for some objects: the MAC address specifies where the packet should go and from whom it was sent;
- **Reliable delivery**: autoexplicative, the link layer should provide some reliability regarding the bits sent via the link. Such service is useful for instance with wireless links, which are more prone to errors;
- **Error detection and correction**: it may happen that corrupted bits might be sent via the links; in order to detect them, the link layer checks for errors in the frame and, if present, asks to the receiving node to perform a message check once the frame arrives on the other side.

The link layer is implemented in the **network interface cards** (sometimes called **NIC**s), which are present in every node; they are attached to the host system busses, which allows the system to use the received data. The **implementation** is a **mix** of **software**, **hardware** and **firmware**.

# 11.1 Error Detection

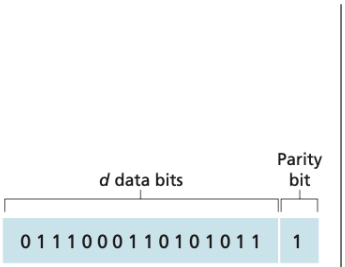
Error detection (also called **E**rror **D**etection and **C**orrection, **EDC**) is a service that is implemented at the link layer. It might not always work (some wrong bits might still be sent since they went undetected). It works in this way: the datagram gets framed and a EDC header is added to the frame. The frame gets then sent through the link and, whenever it reaches the other node, a check is performed: if all the bits are fine and correct, then the frame gets decomposed into the datagram, while if some bits are not correct, then the link layer can take some actions. Having larger EDC headers allow for a major security, since they can carry more information.

There are various checks that can be done, and they are the **parity checking**, the **Internet checksum** and the **C**yclic **R**edundancy **C**heck (**CRC**).

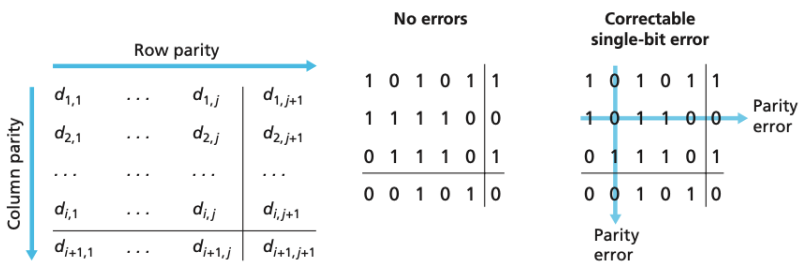
## 11.1.1 Parity Checking

There are two types of parity checking: either **single bit parity** or **two-dimensional bit parity**:

- **Single bit parity**: considering $d$ data bits, we also send one additional parity bit. Such bit will be 1 if the **number of 1's** in the $d$ **data bits + the parity bit** is **even**, while it will be 0 otherwise. Such check can also be done by checking if the number of bits is even. If there is though an **even number** of **erroneous bits** that was sent (with a check for the even number of bits), then the **error** would be **undetected**. Since erroneous bits are usually sent in clusters, the single bit parity check is not enough and may fall down because of some errors. Single parity check can moreover **detect** but **not correct** errors. This is why single bit parity check is not used, but is instead used an advanced version of parity bit;
- **Two-dimensional parity**: the $d$ bits of data are "disposed" in a two dimensional table: for each row and column, a single bit parity check is done. After determining all the parity checks, a further parity check is done between all the parity bits. With this check, we can detect and correct errors, since an error is propagated to both the parity bit of the column and the one of the row.




Single bit parity check




Two-dimensional parity check

The ability of checking by itself eventual errors from an host end is called **forward error correction** (**FEC**). This allows for a faster error connection, because it could be fixed locally, rather than having to send a `NAK` signal and wait for the server to resend the whole message.

## 11.1.2 Internet Checksums

This has already been covered in a previous chapter, but in general, checksums work in the following way:

- from the **sender** side: the content of the UDP segment is treated as a sequence of 16-bit integers; each integer is then summed up by using the 1's complements addition and loaded into the UDP checksum field;
- from the **receiver** side: the checksum is computed again from the receiver and, if it's different from the checksum, the received asks to the sender to send the payload again, otherwise the payload is sent successfully.

## 11.1.3 Cyclic Redundancy Check (CRC)

Another important error detection method is the **C**yclic **R**edundancy **C**heck (**CRC**). Such method produces CRC codes which are also called **polynomial codes**, and that's because such codes can be seen as a polynomial of 0's and 1's. But how does CRC work?

1) A sending host wants to send to a receiving host a message $D$ with $d$ bits of data;
2) Upon connection, both sending and receiving hosts agree on a bit pattern called **generator**; such generator is expressed as an $r+1$ bits number, and is denoted as $G$. The most significant bit of the generator has to be a 1;
3) The agreement is based on the following logic: having $d$ bits of data and $r$ additional bits (that are decided by the sender and are appended at the $D$ bits. The additional bits are denoted as $R$), we must choose a generator $G$ for which $d+r$ is **exactly divisible** by $G$, and we do so by using the **modulo-2 arithmetic**. If the **reminder** of the division is **nonzero**, the receiver knows that some bits are **corrupted**, otherwise the message isn't corrupted.

In modulo-2 arithmetic, additions have **no carries** and subtractions have **no borrows**. This means that doing an addition between two numbers is **identical** to do a subtraction. Both addition and subtraction are equal to the `XOR` of the two numbers:

```
1011  +      1011  -      1011  XOR
0101  =      0101  =      0101  =
----         ----         ----
1110         1110         1110
```

Regarding multiplication and division, they are the same as in the base-2 arithmetic, only that any required addition or subtraction is made without carries or borrows. As per the standard base-2 multiplication, any multiplication by $2^k$ shifts the bits to the left by $k$ places. In general, with CRC, we have to perform the following formula:

$$\underbrace{D}_{\text{Data bits}} \cdot \overbrace{2^r}^{r \text{ is the size of } R} \quad \texttt{XOR} \quad \underbrace{R}_{\text{CRC bits}} = nG$$

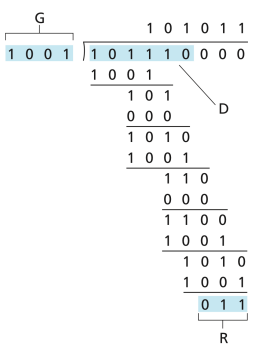



**Figure 11.2:** Example of CRC

That is, we have to find a sequence $R$ of bits such that $G$ divides $D \cdot 2^r$ XOR $R$ without any remainder. If we XOR $R$ to both sides we get

$$D \cdot 2^r = nG \text{ XOR } R$$

We can recover $R$ in this way: we know that $R$ is the remainder between $D \cdot 2^r$ and $G$, thus

$$R = \text{remainder}\left(\frac{D \cdot 2^r}{G}\right)$$

## 11.2 Multiple Access Protocols

There are two types of links: **point-to-point** links (such as the Ethernet connection between the Ethernet switch and a node, or like the dial-up access) or **broadcast** links (via either a shared wire or a shared medium. Some examples are the 4G/5G connections or the 802.11 wireless LAN).

Before starting to analyze the various link layer protocols, let's consider a problem: what happens if, on a single shared broadcast channel, two or more simultaneous nodes transmit on that same channel? The signals would **collide** eventually, and an **interference** would generate. How do we manage this? Via some **multiple access protocol**. This protocol provides a distributed algorithm that determines how various nodes should share a channel, so when they should or should not transmit. The communications relative to when a node could use the channel or not happen on the same channel, so there is no out-of-band channel for such communications.

An ideal protocol would work in the following way: given multiple access channels (**MAC**s), ideally:

- when one node wants to transmit, it can do so with bandwidth $R$;
- when $M$ nodes want to transmit, they can do so at rate $M/R$;
- this would be a fully decentralized algorithm: there are no special nodes which coordinate the transmission and there is no synchronization on clocks or slots, it's a very simple algorithm.

There are three types of multiple access protocols, each divided into its own class:

- **channel partitioning protocol**: for such protocols, the channel is divided into **different pieces**, depending on the **time**, the **frequency** or a **code**, and each piece is allocated to a node **exclusively**;
- **random access**: the channel is not divided, so collisions may happen, but there is a way to recover from collisions;
- **"taking turns"**: nodes take turns to send over a channel, but nodes with larger files that must be sent can take longer turns.

### 11.2.1 Channel Partitioning Protocols: TDMA and FDMA

At the beginning, when there was a general overview of the network, the concept of time and frequency division was introduced. Here are some more details regarding these two partitioning protocols:

- **Time Division Multiple Access** (**TDMA**): the channel is **divided** into multiple **time slots**. Let us suppose that a channel is partitioned in 6 slots, and that 3 nodes want to connect to the router via this partitioned link. Each node will have its **own time slot** assigned. If a slot isn't assigned, then it remains **masterless** and in **idle**, and **won't be merged** for instance with another slot. **Each time slot is equal** to the others, and all the nodes have their own fixed packet transmission time (or slot length) on which they can transmit;
- **Frequency Division Multiple Access** (**FDMA**): the channel spectrum is divided into **frequency bands**, and each frequency band is assigned to a node. As per TDMA, each **masterless frequency** will remain **idle**.

### 11.2.2 Random Access Protocols: Slotted and Pure ALOHA, CSMA/CD

For the class of random access protocols, the way a node transmits is described by the name of the class: at "random". a node transmits at full rate $R$, and there is **no** a priori **coordination** among nodes. If there are two nodes transmitting at the same time, a collision will occur. The random access MAC protocol gives specifications for **detecting** and **recovering** from collisions (for instance via delayed transmissions). There are various protocols belonging to this class, and some of the most famous are the Pure ALOHA, Slotted ALOHA, CSMA, CSMA/CD and CSMA/CA protocols.

In order to explain the **Slotted ALOHA** protocol, we'll make some initial assumptions:

- all frames have the same size;
- the time is divided into equal size slots, which are enough to send one frame;
- the nodes start to transmit only at the beginning of a slot;
- if 2 or more nodes transmit at the same time slot, then all nodes will detect a collision.

The protocol is really simple, whenever a node receives a new frame to send, it will **wait until** the **next time slot** to transmit and, if no other nodes are transmitting then there is **no collision** and the frame can be **sent successfully**, otherwise **if a collision occurs** then the frame will be **retransmitted** with a probability $p$ until it will be delivered successfully.

The **good sides** of this algorithm is that if there is a **single active node**, then it can send frames at **full rate**; moreover, it's a **decentralized algorithm**, and all the nodes must only be in sync for the time slots; lastly, it's a very **simple** algorithm. The **negative sides** are instead that **collisions** may occur, and that would waste some slots; there could also be some **idle slots**, which is not ideal; the nodes could moreover be able to **detect a collision** in a **time window** which is **shorter** than the actual **time slot**. Lastly,

they heavily depend on a **clock synchronization**.

About the **efficiency**, let us suppose that $N$ nodes must send a great number of frames, and that each node transmits with a probability $p$:

- the probability that a node transmits successfully in a time slot is $p \cdot (1-p)^{N-1}$;
- the probability that **any** node transmits successfully is then $Np \cdot (1-p)^{N-1}$;
- the maximum efficiency can be given by a $p'$ such that it maximizes $Np \cdot (1-p)^{N-1}$;
- in general, the maximum efficiency for a number of nodes which tends to infinity is equal to

  $$\lim_{N\to\infty} Np' \cdot (1-p')^{N-1} = \frac{1}{e} = 0.37 = 37\%$$

  This means that, at best, the link will be utilized for useful transmissions for the 37% of the time.

The **Pure ALOHA** protocol is **simpler** than the Slotted ALOHA, since it **doesn't require** any **time slotting**, and as a consequence **no synchronization** is needed. When a frame arrives at a node, it will be sent immediately. Although the algorithm is simpler, the **probability of collisions is way higher**: a frame sent at a time $t_0$ could overlap with other frames sent at $[t_0 - 1,\ t_0 + 1]$ even for one single bit, making the whole frame invalid and in need for a retransmission.

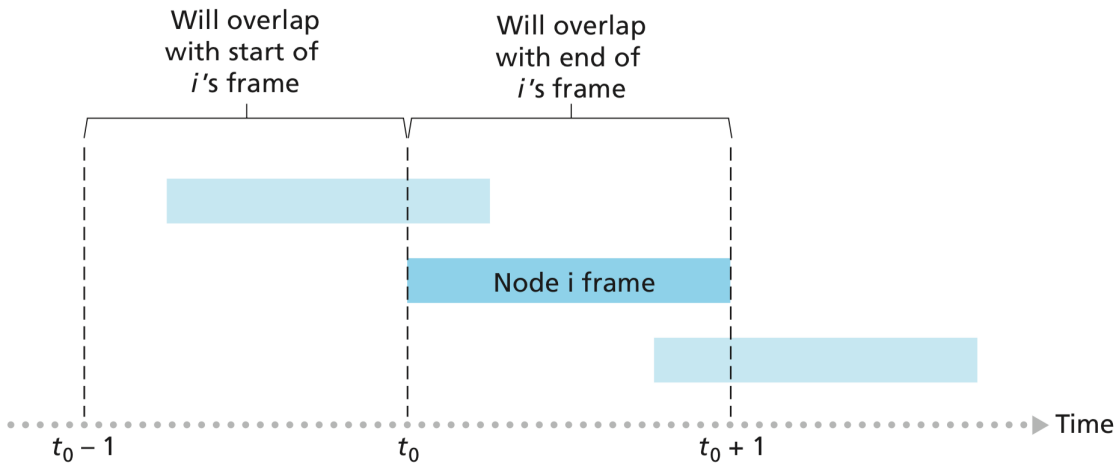



**Figure 11.3:** Example of Pure ALOHA

The probability of success with the Pure ALOHA protocol is given by the probability of successful transmission for one node times the probability that at the times $[t_0 - 1,\ t_0]$ and $[t_0,\ t_0 + 1]$:

$$p(\text{success}) = \underbrace{p}_{\text{a node transmits}} \cdot \overbrace{(1-p)^{N-1}}^{\text{no other node transmits at } [t_0-1,\, t_0]} \cdot \underbrace{(1-p)^{N-1}}_{\text{no other node transmits at } [t_0,\, t_0+1]}$$

As before, if we suppose that $N$ grows to infinity, we get the following:

$$\lim_{N\to\infty} p \cdot (1-p)^{2N-1} = \frac{1}{2e} = 0.18 = 18\%$$

> A version of the ALOHA protocol uses a technique called **backoff**: depending on a random value $k$, the protocol can retransmit the packet up to the $k^{\text{th}}$ time. If after the $k^{\text{th}}$ time the protocol wasn't able to retransmit, it will abort the connection.

We can define also some more concepts, such as the **collision duration**, which is the window of time on which two or more frames overlap. We call **vulnerability time** the time on which a collision occurs. In the Pure ALOHA protocol it is equal to $2T_{fr}$, where $T_{fr}$ is the transmission time for a frame, while on the Slotted ALOHA protocol it is equal to $T_{fr}$. In Pure ALOHA it is equal to $2T_{fr}$ because we have to account for any frame that could arrive in time $[t_0 - 1,\ t_0]$ and $[t_0,\ t_0 + 1]$, while in Slotted ALOHA the vulnerable time is equal to the time slice size, which is, by our initial assumption, equal to the frame transmission time $T_{fr}$.

A solution to the problem that the ALOHA protocol doesn't listen for any possible collision was implemented in another protocol, the **C**arrier **S**ense **M**ultiple **A**ccess (**CSMA**) protocol. A simple CSMA protocol **listens** for any possible collision **before sending** and, **if no collision** is detected, it then proceeds to **send the frame**. Another version of the CSMA protocol, the **CSMA/CD** protocol (where CD stands for **C**ollision **D**etection), can **detect collisions** within a **short period of time** and automatically **aborts** all the **colliding transmissions**, maximizing the throughput of the link. This is easy within a wired link, but it becomes harder with a wireless link.

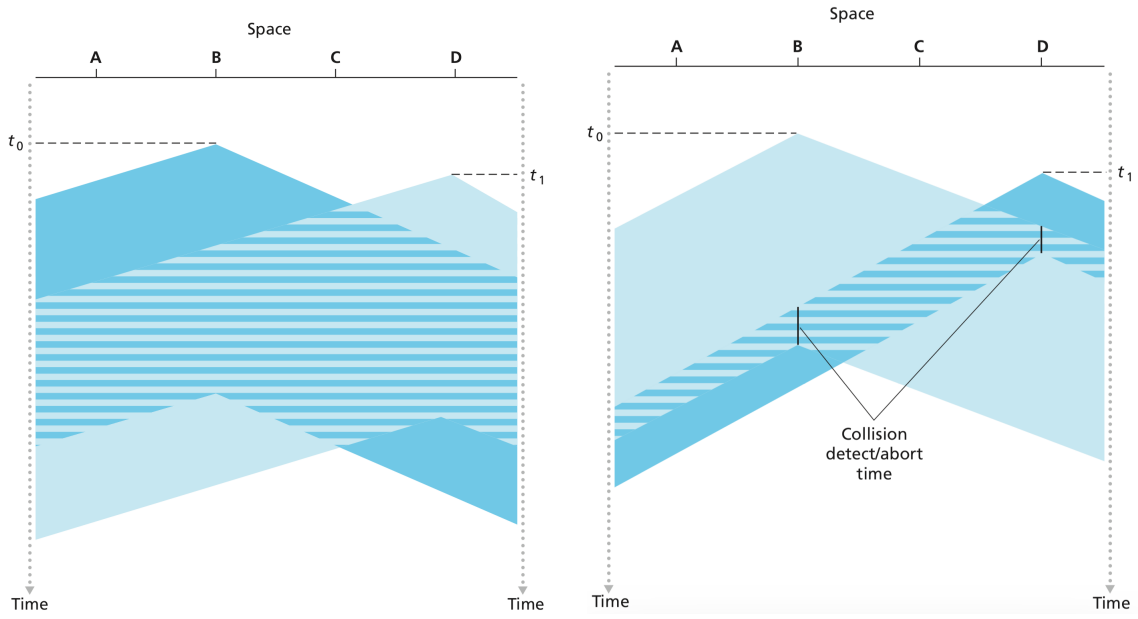



**Figure 11.4:** On the left, an example of the CSMA protocol; on the right, an example of the CSMA/CD protocol

This is how the CSMA/CD protocol works:

1) A node $A$ wants to send a packet on the link: seeing that the link isn't being used at the moment, $A$ starts to send the packet at $t_0$. Once it starts, the packet starts to propagate to all the other nodes. The **vulnerable time** for all the other nodes is **equal** to the **propagation time** needed for the packet to **reach the hosts**;

2) At a time $t_1$ node $B$ wants to send a frame, with $t_1$ being **within** the **vulnerable time**, to another node. Seeing that the node isn't being used (it doesn't know yet that $A$ is sending something too) it starts to send its frame;

3) At time $t_2$ the frame sent from $A$ propagated to $B$: in that moment $B$ realizes that a **collision occurred**, and **sends** a **jamming signal**, which specifies that a collision occurred. Meanwhile though, $B$ has already sent for $t_2 - t_1$ time its frame, which could've possibly **corrupted** $A$'s frame;

4) Once the jamming signal reaches $A$ at time $t_3$, $A$ has to **abort** the **sending** of the frame and **will have**, at a later time, to **retransmit all the frame**.

The minimum frame size that allows all the nodes to detect a collision is given by the **number of bits** that **can be transmitted** in $2T_p$ (readily, twice the propagation time). This is enough to let the other nodes understand whenever a collision happens.

The Ethernet protocol uses the CSMA/CD algorithm, with a different implementation of the backoff time: if a node has to send a frame for the $n^{\text{th}}$ time because of a collision, it chooses a value $k \in [0, 2^n - 1]$ of retransmissions. This means that **the more collisions** a node experiences, the **more** $k$ can be taken from a **wide set of numbers**. After choosing $k$, the Ethernet protocol waits the time needed to send $k \cdot 512$b. The value of $n$ though is **capped to 10** (always for the Ethernet protocol). This type of backoff is called **binary (exponential) backoff**.

### 11.2.3 "Taking turns" Protocol

**Channel partitioning protocols** can **share channels efficiently** when there is a **high load**, but they are pretty **inefficient at low loads**; **random access protocols** are instead **good** when there are **few nodes transmitting**, but on **high loads** they are pretty **inefficient**. The "taking turns" protocols try to take the best of the two types of algorithms.

For such kind of protocols, there is a **master** node which invites the other nodes to transmit in turn, like in a round robin algorithm. **All** the **nodes** communicate on a **common link** with the master, and **only one node** at a time can **transmit**. There are some concerns regarding this kind of techniques: not only there is a **single point of failure**, which is the master, but there can also be a **polling** of the link (a node could take more time because of the need to send larger files) and a **high latency** in the case there a node uses the link for too much time.

Another way to implement the "taking turns" protocol is to have a **token passing protocol**: **there isn't a master** anymore, and the sender is identified by a **token** that gets **passed** between each node in a **fixed order**. If a node doesn't have to send anything, it will **pass** the token. This is an **highly efficient** and **decentralized** protocol, but it has some flaws: for instance, if a **node fails**, then the whole **channel** could **crash**; if instead a node takes too much time to send its files, it could **block the entire channel**.

## 11.3 LANs and ARP Protocol

It's common to hear the term **LAN** in computer networking, but what is it precisely? It stands for **L**ocal **A**rea **N**etwork, and it denotes a smaller network that connects various

devices which are physically located one nearby the other. We saw in the network layer that in order to distinguish each device, an IP address is used, which (considering the IPv4 protocol) is made of 32 bits. At the link layer there are the **MAC** (or **LAN** or **physical** or **Ethernet**) **addresses**, which are used locally to distinguish the various **physical interfaces** of a router, a switch or a host. Each MAC address is made of 48 bits, and it's hard coded in the hardware (it's rarely settable via some software); it is shown usually as a set of six hexadecimal numbers, with each number having two ciphers:

`C3:9F:4A:62:49:CD`

**Each interface** on a LAN will have then a **locally unique IP address** and a **unique MAC address**. Why do we need both addresses? Suppose that a computer has both a wireless and wired network interface, then via the sole IP address it won't be possible to distinguish the wired or wireless interface. Via the MAC address though, one could refer to only the wired or the wireless interface.

The MAC addresses are **managed** by the **IEEE**, which sell the addresses in blocks. The difference between MAC addresses and IP addresses is that MAC addresses are **portable**: if an interface gets moved from one LAN to another, the MAC address won't change, but the IP address will.

While to retrieve a destination IP address we just need to consult a DNS, what do we do with MAC addresses? There has to be a way to determine the MAC address from the destination IP address. There is a protocol that allows each IP node (be it a router or an host) to have a **table** with the various **IP addresses** as **keys** and **MAC addresses** as the **values**: such table is called **ARP table**, because it's part of the **A**ddress **R**esolution **P**rotocol (**ARP**).

The entries in a table are stored in the following way:

`<IP_address; MAC_address; TTL>`

ARP tables entries also include a **TTL**, which usually has the value of 20 minutes; after such **20 minutes** the entry will be discarded. Here is an explanation on how the ARP protocol works in detail:

1) Suppose that an host $A$ wants to send something to a host $B$, but it has only the destination IP address, and it needs $B$'s MAC address. In order to find it, it broadcasts a message on the whole LAN at the MAC address `FF-FF-FF-FF-FF-FF`, so that all the nodes will receive that query;

2) The query sent by $A$ is in a format similar to the following:

   ```
   ...
   Source MAC: 49-BD-D2-C7-56-2A;
   Source IP: 222.222.222.222;
   Destination IP: 222.222.222.223;
   Destination MAC: <empty>;
   ...
   ```

   Host $B$ replies to host $A$ with its MAC address; host $A$ will save $B$'s reply within its ARP table and, after the TTL, it will discard it.

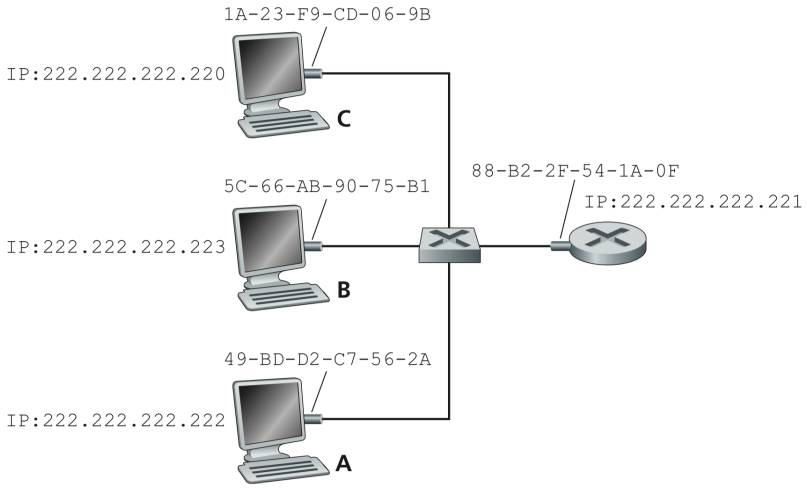



**Figure 11.5:** Example of a LAN with different nodes

Now let us suppose that host $A$ still wants to send a frame to host $B$, but now host $B$ is located in **another subnet**. Both $A$'s and $B$'s subnets are connected via a router $R$. For this example, we assume that $A$ knows the IP address of $B$ and $R$'s IP and MAC address. The sending process would be as follows:

1) $A$ prepares a packet, where in the datagram it places $B$'s IP address and in the frame it places the MAC address of the router's interface connected to $A$'s subnet. In the case where $A$ didn't have $R$'s MAC address, it could make an ARP query to retrieve it. $A$ sends then the packet to $R$;

2) $R$ receives $A$'s packet, it extracts the datagram and, once it determines the output interface, it looks up for $B$'s MAC address and, once it encapsulates the datagram back in a frame with the correct MAC address, it sends the message to $B$.

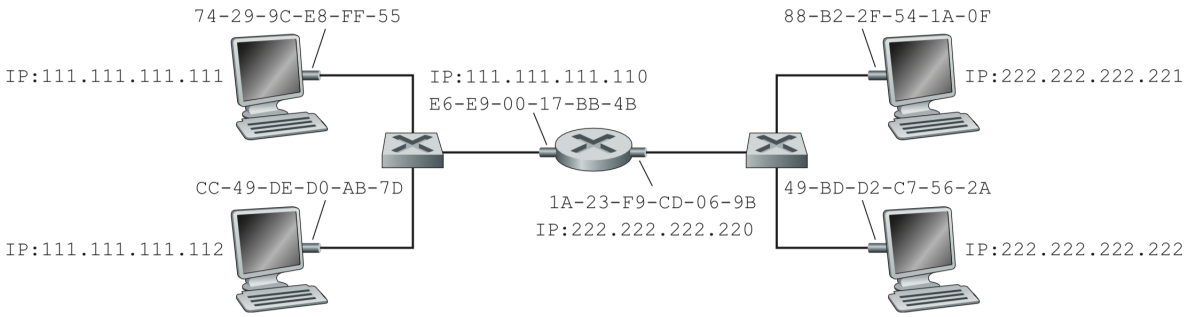



**Figure 11.6:** Example of two subnets and of two hosts exchanging data

ARP queries can be either **broadcast** or **unicast**: **broadcast** queries are made to all the devices in the LAN, while **unicast** queries are made directly to a specific host.

## 11.4 Ethernet

In the previous decades, the **IEEE 802** project was established by the IEEE. Such project contains various specifications regarding LANs and **P**ersonal **A**rea **N**etworks (**PAN**s). Among all the specifications, the **IEEE 802.3** specification is one of the most used. Such specification is also called **Ethernet**.

In the early 90's, Ethernet's physical topology saw all the various hosts connected via a single shared bus, which usually was a coaxial cable, and all nodes could collide between each other. Nowadays, a **switched** topology is adopted, and for this topology there is an active link-layer **switch** in the "center", between all the hosts. In the past, **hubs** were used instead of switches. An hub is a circuitry that takes as input a signal and re-sends it on another physical interface; by doing so, the strength of the signal would be boosted.

The Ethernet standard has its own type of frame header, which is made of two major parts: a **physical-layer header** and the **frame**. The physical layer header is added by the physical layer and contains a preamble and a **S**tart **F**rame **D**elimiter (**SFD**). The preamble is made of 7 bytes, and all the bytes are equal to `10101010`. These bytes are used to **synchronize** the receiver's clock to the sender's clock. The SFD is instead equal to `10101011`, and it's a **flag** used to determine when the frame starts.

The frame is made of the destination and source MAC addresses fields (6 bytes each), a **type** field (which specifies the type of protocol used at a higher layer; it could be the IP protocol or the ARP protocol. The field takes 2 bytes), the **data** field (which goes from a minimum of 46 bytes up to a maximum of 1500 bytes) and the **CRC** field, with a 4 bytes size.

In the Ethernet frame, the MAC addresses are sent in reversed order. Consider for instance the following address:

$$\texttt{47-20-1B-2E-08-EE}$$

In binary it would be:

$$\overbrace{01000111}^{47}-\overbrace{00100000}^{20}-\overbrace{00011011}^{1B}-\overbrace{00101110}^{2E}-\overbrace{00001000}^{08}-\overbrace{11101110}^{EE}$$

The order on which the addresses would be transmitted is the following:

$$11100010-00000100-11011000-01110100-00010000-01110111$$

Notice how the flipped address maintains the same order of the 6 numbers but it reversed each number singularly. But why is this done? Such thing is done in order to differentiate between **unicast**, **multicast** or **broadcast** messages:

- in a **unicast** transmission, all the nodes in a LAN will receive the frame, but only one node (the one with the corresponding MAC address) will keep the frame; the other nodes will discard such frame;
- in a **multicast** transmission, all the nodes in the LAN will receive the frame, but only the nodes in the group underlined by the MAC address will keep the frame, while the others will discard it;
- in a **broadcast** transmissions, all the nodes will receive and keep the frame, except for the sender.

> This types of messages can be recognized depending on the second cipher of the first digit: that's why the numbers get reversed, so that it's easier to analyze such cipher:

- if the cipher is **even** then the address is **unicast**;
- if the cipher is **odd** then the address is **multicast**;
- if all the ciphers are `F` then the address is **broadcast**.

The Ethernet protocol is a **connectionless** and **unreliable** protocol though: it's connectionless because it establishes **no handshake** between the network interfaces, and it's unreliable because there is **no acknowledgment system** between the two network interfaces. Moreover, if any data at a layer lower than the transport one gets lost, then it **won't be recovered** (that is, unless the transport protocol asks again for retransmission). The Ethernet protocol uses the **unslotted CSMA/CD** protocol with **binary backoff**.

Today the Ethernet protocol is still widely used because of its popularity, and there are **different versions** of it depending on the available sending rate on a link and on the type of link (for instance if a coaxial cable is used rather than a fiber optic cable).

## 11.5 Hubs and Switches

As we said previously, two common elements in LANs are **hubs** and **switches**. We recall that an hub is just a physical layer device that acts on the physical bits: whenever it receives a bit, it **repeats** it by boosting the signal to **all the outgoing interfaces**. An Ethernet network with a **starting hub** is called **broadcast LAN**; for such configuration though if two frames arrive at the hub from two different nodes at the same time, **collision** occurs.

**Switches** are link layer devices that store and forward Ethernet frames depending on the destination and/or source MAC address, it selects an output interface (or more than one) for such frame. It's a transparent device: the hosts are not aware of the presence of the switch on the LAN. Switches are, moreover, a plug-and-play device, and they don't need any configuration.

Hosts have a **dedicated** and **direct** connection to a switch, which is also capable of **buffering** the hosts' frames. The Ethernet protocol is used only at **each input port** of the switch, so we can have full-duplex connections and no collisions at the switch itself; rather, the **collision domain** is the **link itself** between a switch and an host. Moreover, an host $A$ can send data to an host $B$ and make the data pass through the switch while two other hosts $C$ and $D$ to the same. The only requirement is that each host must have its own direct link to the switch. Each switch has a **switch table**, where each entry stores the **MAC address** of the **host** at the other end of a link and the **switch port number** to which the host is connected; a **TTL** is also stored alongside the MAC address and the port number. It's conceptually **similar** to a **forwarding table**. This table gets **automatically compiled** by the switch as the traffic passed by the switch itself.

The switches also provide some filtering and forwarding functions:

- if the frame arriving at the switch contains a destination MAC address which is not in the switch table, the switch **floods** the network by sending the frame to all the

hosts. Whenever the host will send something through the switch, the switch will register the MAC address;

- if the frame arriving at the switch contains a **known destination MAC address** which is different from the source address, then the switch **forwards** it to the right port;
- if the frame arriving at the switch has a destination MAC address **equal** to the source MAC address (so the destination MAC address on the switch table corresponds to the incoming port number), then the switch **filters** and **drops** the packet.

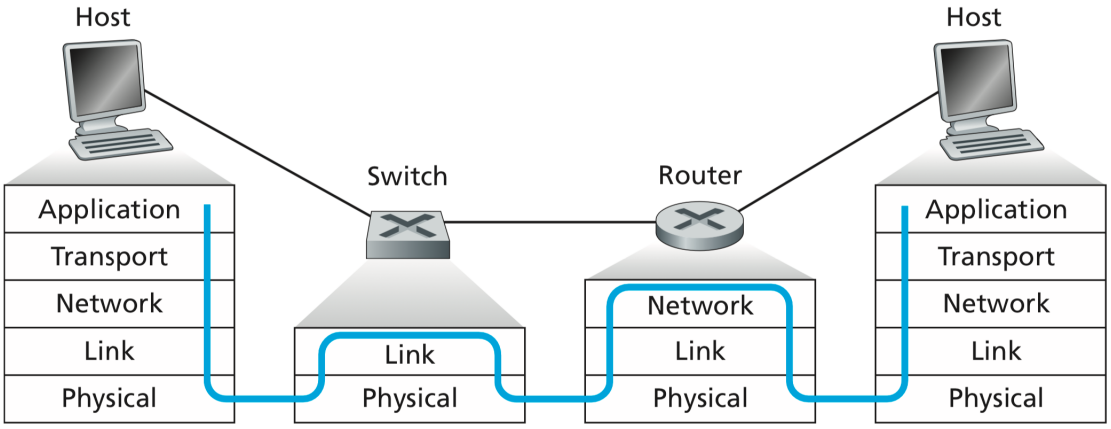



**Figure 11.7:** Difference between switches and routers

In the case of **multiple interconnected switches**, the logic is the same: each switch has its own switch table which gets completed over time thanks to the traffic passing by the switch itself. But what are the **differences** between routers and switches? Most of the services that they offer are the same: they both store and forward packets. The difference in the **store-and-forward** actions performed by the two types of devices is that **routers** store-and-forward depending on the **network layer** information such as the **IP addresses**, while **switches** act on the **link layer**, by looking at the **MAC addresses**. Both devices have some kind of "**forwarding tables**", but the ones used by the **routers** are **completed** with the use of some **routing algorithms**, while the ones used by **switches** are **completed** with **self-learning**.

Lastly, switches are **plug-and-play**, and have a **lower job-load**. They **must use** a **spanning tree structure** though, since they have no way to detect cycles, which could possibly flood the network. Routers on the other hand **can produce cycles** only if it's useful to redirect a packet on the least-cost path. Moreover, routers are usually **used on wider networks** and are **not plug-and-play**, since they require some prior configuration.