Management of physics datasets, mod. B

perina

July 2019

Contents

1 Campana's slides

1.1 Introduction

When dealing with physics experiments the likes of those performed at CERN, the events we are looking for have ridiculously low probabilities. The production of just ≈ 10 Higgs required 10^{11} billion collisions. Such a huge number of events obviously makes their management hard. The detector sees about 50 simultaneous collisions happening at a rate of 30 MHz. This is a huge data volume for the ATLAS detector to read out. Taking also into account that each collision produces data of about 1.5MB in size, this would correspond to 45 TBytes of data produced per second. This is a huge number of events to store, but also to process. Furthermore, most of these events are not really interesting for analysis. So we need a way to select which of all these 30MHz of events to keep. This is done through a sort of filtering process, that progressively reduces the amount of data, going from the detector to the publication. An event's lifetime goes as follows:

1. **Detector**;
2. **Trigger**;
3. **Data preparation, reconstruction and calibration**;
4. **Data analysis**;
5. **Publication**.

An additional, somewhat parallel step, is the one related to **simulations**.

Triggering is the first processing step of the event that is created at the center of the detector after the collision has occurred. If the trigger decides for the event to be accepted for further processing, the event moves to disk. The data get reconstructed and calibrated such that we can trust what we are measuring.

On parallel, we generate simulated data that describes what the standard model signals should look like on the detector, or how new physics signals should look like. We feed our detector data and our simulated data to analysis software that we develop, often using sophisticated analysis techniques. The results of these analyses is what makes our physics publications.

Triggering is essential in reducing the magnitude of the data stream coming from the detector. LHC collisions clock in at ≈40 MHz, and the triggering procedure allows us to get a more manageable ≈1 kHz. This triggering system is actually composed of two separate stages: a first layer, with selection logic encoded on fast electronics, provides a less than 0.5% acceptance. The data is then passed on to a large CPU farm, the High Level Trigger, which is running fast algorithms to take the final decision. In the end about 1 in 30'000 events are stored for further processing. The trigger helps the selection of interesting events, but also applies upper limits in the number of events we can select at each of its steps. After the detector+trigger system has completed its job, RAW data is produced. These will later be made available to analysis after adequate preparation.

An interesting aspect of this scheme is the time every step takes. Collisions occur every 25 ns. The hardware trigger takes 2.5 μs to process the event and decide whether to keep it or not. The HLT takes about 300 ms. This constitutes what we call the "**online**" processing, and it effectively happens in "real time". Anything beyond that processing step is happening "**offline**". In general, later steps take much more time than the ones at the beginning.

This is where high efficiency computing (HEP) steps in. The main purpose of computing facilities is to make RAW data into usable data, by means of reconstruction, calibration and Monte Carlo simulations. This whole offline part converges to the final step of actual data analysis.

1.1.1 Reconstruction

The event reconstruction turns RAW data into Analysis Object Data (AOD). (something about event reconstruction). The actual reconstruction involves **tracks** and **clusters**:

- **tracks**: the measurement of particles' trajectories inside a detector are subject to a degree of uncertainty. The smaller the errors, the more constrained the possible trajectories are. More points and smaller errors reduce the possible trajectories. In order to find them, least squares procedures are employed, so that uncertainties about the trajectories themselves can be retrieved, too;
- **clustering**: by clustering, we mean the procedure of reconstructing the amount of energy that gets deposited in the calorimeter by the particles that traverse it. The meaningful quantities are: the energy, the position of the deposit, and the direction of the incident particles. Typically, the shower produced by one or more particles extends over contiguous cells (the basic units of which a calorimeter is composed). There are various clustering algorithms that can group various cells based on the expected behavior of a shower pattern.

Combining these yields a so called **object** (which one could also call a particle): trajectories and energy measurement, grouped together.

1.1.2 Monte Carlo simulations

At this point, the data is ready for the actual analysis. However, at this level, Monte Carlo simulations come into play. This is because we only have one detector: how do we know that the effects it measures are due to its specifics and construction? Maybe, a different detector would have produced different results. And what about the detector's interactions with radiation?

Moreover, when we measure something through a detector, the measures we get are voltages, currents, times. Linking them to a specific particle set, however, is an interpretation. Simulating the detector itself can aid in correcting inefficiencies, inaccuracies and so on. Monte Carlo simulations are used as a way to get the expected values from theory, which can then be compared to the experimental values. They are also fundamental in order to prove that the detector's working is known to the experimentalists. The chain of Monte Carlo simulations works as follows:

- **Event generation** (from $< 1s$ to a few hours per event): simulating the actual physical process through which the particles are produced;

- **Detector simulation** (from 1 to 10 minutes per event): simulating the interactions of the particle with the materials that compose the detector;
- **Digitization** (from 5 to 60 s per event): translating the interactions into realistic signals;
- **Reconstruction**: analogously to what happens for real data, the events must be reconstructed.

This simple overview is enough to realize how computationally demanding the whole procedure is. There is 1 PB/s of data generated by the detectors, up to 60 PB/year of stored data. Some large experiments have managed data sets of more than 200 PB. This equates to one million cores working 24/7. This amount of processing and storage, if centralized, would place CERN amongst the top supercomputers in the world. However, the system is actually distributed.

1.1.3 The GRID model

The GRID model is a computing model that involves a distributed architecture of a large number of (loosely) coupled computation and storage centers, which are connected in order to solve a complex problem. GRID is organized in multiple tiers, levels with different sizes and different tasks.

- Tier-0: Located at CERN. Involves data recording, reconstruction and distribution;
- Tier-1: permanent storage, re-processing, analysis. Tier-1 sites are located in multiple research facilities around the world (10^1 as order of magnitude);
- Tier-2: these smaller sites (around 160 of them) are focused on simulations and end-user analysis.
- Tier-3: smaller institutes;
- Tier-4: single workstations.

In total, there are 170 sites in 42 countries, 750k CPU cores, 1 EB of storage (10^{18} byte), > 2 million jobs per day and 10-100 Gb links. An international collaboration for the distribution of LHC data exists in the Worldwide LHC Computing Grid (WLCG), which integrates different computing centers around the world under a single infrastructure accessible to all involved physicists.

The fact that the whole system is distributed makes so that there is a continuous data transfer at rates of 35GB/s across the WLCG.

The HL-LHC is expected to produce 1 EB of data by 2026. How can such a massive amount of data be managed? There is a need for a computing model: GRID itself is a computing model, a distributed one at it.

1.2 Data management components

Scientific computing is based on a distributed architecture. This type of architecture is based on three main pillars:

- **Computing power**;
- **Storage**;
- **Links (network)**.

These three are deeply interconnected, and are limited by budget. Any approach that aims at optimising the performance/cost of a model must optimize with all three aspects in mind. Having low CPU usage may be good, but not if it requires a lot of memory/network costs.

In general, the main problems which data management faces are the following:

- **Data reliability**: the data must be self consistent over time and protected from spoil;
- **Access control**: it is important to decide who can actually access the stored data. There is the need for an authentication protocol;
- **Data distribution**: there must be a way for data to be transferred to whoever may require them;
- **Data archives, history, long term preservation**: simply put, the data must not be modified as time passes. The way to achieve this is to resort to tapes, which make data reachable and readable;
- In general, a data management protocol must allow the **establishment of a workflow for data processing**, covering all steps that are needed from data collection to end analysis.

1.2.1 Storage models

The simplest version of a storage model is storing all data together in one place. Its advantages are uniformity (everything is done in the same way), simplicity, ease of maintenance and no need to move data. This means that performances and reliability can be quite good. However, this is far too simplistic for a real life situation.

Something similar happens on the commercial side of things. Google Drive, Dropbox etc. are simple cloud-based systems which provide the user with two main services: get and put (and also some additional ones, like sharing and so on). In case of need, additional services may also be purchased. Such a model works because it receives a lot of funding, since the companies are private: this is a commercial model. Each Petabyte on Dropbox costs 10 times its ”physical amount”: the additional price comes from all the services built around the storage itself. Needless to say, physics experiments needs something with a much better storage/cost ratio. Moreover, the requirements for the storage largely depend on what kind of physics we are dealing with.

For a statistical mechanics application, for instance: the starting point is a bunch of configuration files, each of small size (approximately 1MB), which are then fed to some computationally intensive procedure (typically, simulations). This is a CPU bound operation. When it comes to storage, we do not need something too massive, as the files are not straining for the storage. However, we must be sure that the configuration files are preserved. To do this, we may employ a large numebr of disks. We need to achieve

- redundancy;
- reliability.

For a high energy physics application, instead, the most demanding part is the storage one. We deal with large volumes of data, over which we need to perform a small amount of relatively fast operations: the operation is IO bound (input/output bound: a large fraction of time is spent on accessing data). In this case, what we need is actually a few copies with high performance. Ina sense, we are asking the dual version of the storage needed before.

This is where multiple **data pools** come into play. We need a system that is suitable for both requirements; to achieve such flexibility, the preferred way is to employ a number of specialised structures (pools). Indeed, the two building blocks to empower data processing are:

- Data pools with different quality of services (defined on the basis of the particular need of the process we are considering);
- Tools for data transfer between pools.

Data pools themselves have three main parameters, which can be conveniently placed as the vertices of a triangle. These are **performance**, **reliability** and **cost**. Only two of these can be achieved at a time. As examples, flashdrives, solid state disks (SSDs) and mirrored disk are fast and reliable, but expensive. On the contrary, tapes are extremely cheap and reliable, but lack speed (due in part to their sequential nature). Disks, instead, are cheap and fast, but unreliable. As a consequence, depending on the service one needs, an appropriate mix of these must be employed.

Actually, the triangular model is an approximation, as more categories can be evaluated as a metric for comparing different hardware choices. For instance: where is scalability?

1.2.2 Name servers and databases

The name server is the database which contains the catalogue of all data. In a sense, it acts like the index of the greater dataset. It is a simple lookup based, single key database application that can be implemented in various ways. However, it is the most complex and error prone component, while also being the most important. Losing one disk of the database is not tragic: data can be retrieved. However, losing the name server means losing all the information.

As a consequence, **reliability is critical**, and the name server must be accessible at all times. Furthermore, because of its role, the name server is potentially a bottleneck in the overall performance.

When a server is not able to provide all requested information, it enters a denial of service (DoS) state. A good name server is one which, when the maximum number of requests is reached, does not immediately enter DoS, but instead manages incoming requests and leaves them pending until they can be processed. On the other hand, for a bad name server, reaching the maximum means shutting down the whole system.

An improvement of performance can be attained by employing an **indexed lookup table** and returning the requested data in multiple batches (**paging in shards**). This is because, for requests that involve a large fraction of all entries in the database, a complete fulfillment of the request may result in having the server locked in that particular high weight task.

An example of a well designed name server system are the **uniform resource identifiers (URI)**: for example, the address

`http://csc.cern.ch/data/2012/School/page.htm`

can be decomposed in successive parts, following a sort of hierarchical reasoning.

At first, we define the protocol ('https'), then the host/domain ('csn.cern.ch'), then a particular volume ('data') and so on. This is an example of a **federated namespace**, which offers almost infinite horizontal scalability and high efficiency, but is weak to DNS failure: if the DNS fails, no information can be retrieved at all.

A similar problem arises in storage systems:

`storage://cern.ch/data/2012/School/page.htm`

where the database lookup is located at the file level, which is the last one (in this case, 'page.htm'). This allows a greater flexibility, but it impacts the performance. In general, the two extremes are:

- **file systems** (i.e., no database);
- **storage systems** with **lookup tables at the file level**.

and many possible compromises exist between the two.

1.2.3 Data access protocols

The starting point in data access protocols is **POSIX** (Portable Operating System Interface for Unix): this is a standard which (among other things) defines **file-level access**: opening, reading, writing, deleting and so on.

In posix, metadata is limited: there are users, permits but not much else. This, together with the fact that some operations (like ls) are not scalable, make posix a sub optimal choice for distributed data. However, there are some commands which perform as if they were posix but are suitable for distributed data.

The concept of distributed data is also linked to that of **efficiency**. Imagine we had to perform the sum of two number on disk. We would need to perform the following operations:

- open the file in the disk;
- allocate memory in the RAM in order to store the two numbers;
- move them in the CPU register and sum the two number s there;
- transfer the result back in the memory, and from there to the disk if necessary.

This simple example highlights an important concept, that is **data locality**. When data and computing power (i.e., CPUs) are co-located, the efficiency can be high. Whenever a site has idle CPUs (i.e., not enough data), idle or saturated networks, or an excess of data (which implies that there are not enough CPUs for analysis), then efficiency drops. Indeed,

$$efficiency = \frac{\texttt{time during which CPUs work}}{\texttt{total run time of the program}}$$

Then, **limiting the idle time of CPUs improves performance**. This is why analysis made with high efficiency requires the data to be pre-located to where the CPUs are available, or to allow peer-to-peer data transfers, so that sites with excess CPUs can process excess data. Both these approaches are used in HEP.

1.3 Reliability

Reliability is related to the **probability of losing data**. A possible definition is "the probability that a storage device will perform an arbitrarily large number of IO operations without data loss during a specified period of time".

Reliability can be referred to both **hardware** and **services**, but the fundamental one is hardware related.

For example, on a disk server with simple disks, reliability of the service is just the reliability if those disks. Data management involves finding ways to improve reliability, even if this means an increasing cost and/or additional hardware (e.g., disk mirroring and RAID).

Hardware reliability is based on tapes. These have a bad reputation in some use cases, as they are slow in random access mode (they are intrinsically sequential) and can have high latency when mounting processes and seeking data (rewinding and fast-forwarding are rather slow), are inefficient for small files and are comparable to disks when looking at their cost. However, they also have many advantages, like a fast sequential access mode (2x the sequential speed of disks), a much higher reliability (several orders of magnitude higher than disks), no need of energy for storing the data (unlike disks, which must be powered in order to store anything), a better physical volume per petabyte ratio, and are less prone to accidental erasing of large amounts of data in a small time. This means that, when not used in random access mode, tapes serve a precise role in the architecture.

When discussing **reliability and disks**, the **redundant array of inexpensive disks (RAID)** system plays a fundamental role. There are many types of RAIDs:

- RAID0: disk striping. Data is distributed between two or more disks, but with no redundancy. This increases the performance, but decreases reliability (so that it is not a proper RAID, as there is no redundancy).

- RAID1: disk mirroring. This is the opposite of RAID1, as the same data is simply copied on two disks, so that they are mirrored. If one particular piece of data is lost, then its copy on the other disk will prevent any actual loss.
- RAID5: parity information, distributed across all disks.
- RAID6: Reed-Solomon error correction (allowing up to 2 disks lost without actual loss of data)

Errors can be detected if there is a checksum: data should be consistent with the respective checksum (so that errors may be detected independently). By using **redundancy**, then, **errors can be corrected**.

Reed-Solomon error correction is based on oversampling a polynomial constructed from the data by a margin of n points. Since any set of m distinct points can univocally determine an $m-1$ degree polynomial, and since that there are $m+n$ points, then we can afford to lose up to n points without actual loss, as the original polynomial can still be retrieved from the remaining points. For different values of n, we have different implementations, as $n = 0$ equals no redundancy, $n = 1$ is the RAID5 architecture and $n = 2$ is the Reed-Solomon one.

With n corrupted values, the Reed-Solomon procedure can only detect an error, while for lower values of n it can also be corrected. However, if a checksum/hash is added on each point, then we can correct also n errors.

With RAID, reliability depends on various parameters, namely the hardware reliability, the type of RAID and the number of disks in the set.

1.3.1 RAID5 reliability

In RAID5, disks are regrouped in sets of the same size, with each disk having capacity c. If there are n of such disks, then the RAID5 capacity will be $c(n-1)$, so that the system is immune to the loss of one disk, but not of two.

Some calculations: disks mean time between failures (MTBF) is between 3×10^5 and 1.2×10^6 hours, replacement time of a failed disk is less than 4 hours. Then, the probability of a disk to fail within the next 4 hours (meaning that there are two failed disks at the same time, and so total loss of data) is

$$P_f = \frac{Hours}{MTBF} = \frac{4}{3 \times 10^5} = 1.3 \times 10^{-5}$$

Then, the probability of at least one disk failing in the next 4 hours inside a 15PB computer centre (that is, a centre with 15000 disks) is

$$P_{f_{15000}} = 1 - (1 - P_f)^{15000} = 0.18$$

In a 10-disks RAID5 system, the probability of one of the remaining disks failing within 4 hours is

$$P_{f_9} = 1 - (1 - P_f)^9 = 1.2 \times 10^{-4}$$

which is obviously smaller. However, we can not assume that the second failure is independent of the first one: after all, the two disks are in the same computer centre. We can then try and arbitrarily increase this number by two orders of magnitude just to take dependencies into account, so that we get

$$P_{f_{9corr}} = 1 - (1 - P_f)^{900} = 0.0119$$

Then, the probability of losing the data stored in the computer centre over the next 4 hours is

$$P_{loss} = P_{f_{9corr}} \times P_{f_{15000}} = 6.16 \times 10^{-4}$$

which may seem small. However, over a 10 year period, this becomes

$$P_{loss10y} = 1 - (1 - P_{loss})^{10 \times 365 \times 6} \approx 1$$

1.3.2 RAID6 reliability

What about RAID6? In this architecture, disks are regrouped in sets of arbitrary size. If each disk has capacity c and there are n disks per set, then the total capacity will be $c(n-2)$, meaning that the loss of up to 2 disks is allowed.

As before, the probability of a disk to fail within the next 4 hours is

$$P_f = \frac{Hours}{MTBF} = \frac{4}{3 \times 10^5} = 1.3 \times 10^{-5}$$

In a set of 10 disks, the probability of one of the remaining 9 disks failing is

$$P_{f9} = 1 - (1 - P_f)^{900} = 1.19 \times 10^{-3}$$

and the probability of one of the remaining 8 failing is

$$P_{f8} = 1 - (1 - P_f)^{800} = 1.06 \times 10^{-3}$$

Then, the probability of losing data in the next 4 hours is

$$P_{loss} = P_{f9} \times P_{f8} \times P_{f_{15000}} = 2.29 \times 10^{-5}$$

and over 10 years, this becomes

$$P_{loss10y} = 1 - (1 - P_{loss})^{10 \times 365 \times 6} = 0.394$$

which is better than its RAID5 counterpart.

1.3.3 Arbitrary reliability

Since RAID systems are disk based, it means that the smallest unit is just a disk. This is a lack of **granularity**: it would be a more flexible solution to have smaller sizes.

This can be achieved by using **file chunks** (or blocks): by splitting files in chunks of fixed size s, and grouping them in sets of m chunks, generating n additional chunks in a Reed-Solomon like fashion and scattering them **across independent storage units** will allow us to retrieve all the information in the m chunks, provided that up to n have been lost or corrupted (in a sense, chunks are treated as disks).

Acting this way, we can achieve **arbitrary reliability**, as the system is immune to the loss of n simultaneous failures from pools of $m + n$ storage chunks. This, however, results in a raw **storage space loss** of $\frac{n}{m+n}$. A higher m results in a smaller space lost, but at the cost of a higher data loss in the case of failure.

This is analogous to **gambling**: a big m may seem a good choice, until a failure actually happens, and a large portion of data gets lost. On the other hand, arbitrarily high reliability can be achieved by setting a small m, but at the cost of a lot of space lost.

The values of m and n can be tuned in the following way:

- starting from the MTBF provided by the hardware builder, decide the (somewhat arbitrary) replacement time of a failed component;
- choose m and n as to expect the wanted reliability;
- while running the system, constantly monitor and record the hardware failures and the replacement times, as to constantly tweak and adjust both MTBF and the average replacement time;
- recalculate m and n using these constantly adjusted values.

1.3.4 Chunks transfer: the BitTorrent protocol

If we want to use chunks, it is necessary to define how these chunks are transferred. There are many protocols, among which the most popular is BitTorrent.

For each chunk, an SHA1 hash (a 160 bit digest, i.e. the output of a cryptographic hash function, which maps any input to a fixed-size binary string in such a way that obtaining the input from the output is extremely hard) is created, and all digests are assembled in a single torrent file, which contains all relevant metadata information.

Torrents are published and registered, with a tracker that lists all clients currently sharing the torrent's chunks. In particular, they contain an announce section, which specifies the URL of the tracker, and an info section, containing names of the files, their length and the list of SHA1 digests. It is then up to the client to retrieve and reassemble the original file, while also checking its integrity.

In order to do so, a hash is associated to each chunk; this guarantees that errors can be corrected also in the case of a loss of n chunks (analogously to what happens when applying checksums in Reed-Solomon). The overhead of the checksum is negligible when compared to the actual chunk size, since a checksum is around a few hundred bytes and a chunk is various megabytes. Still, if we want high efficiency, the physical block size of the storage must fit both the chunk and its checksum. Having chunks the exact size of the physical block is an error, as it leaves no space for the checksum in the same block, and so the chunk will actually require 2 physical blocks instead of one.

1.3.5 A small recap of arbitrary reliability

- **Plain**: reliability of the service is just reliability of the hardware.
- **Replication**: reliable and with maximum performance; however, requires a lot of storage overhead, as data is doubled. For example, with 3 copies, there would be a 200% overhead.
- **Reed-Solomon, triple parity**: maximum reliability, minimum storage overhead. For instance, a 4+2 system can afford a loss of 2 (and so has a storage overhead of 50%); a 10+3 can lose 3 (and so has a storage overhead of 30%)
- **Low density parity check (LDPC)**: this approach uses a bipartite graph where the original data is one layer, and the other one is composed of the additional copies. ¡some other stuff about it maybe later¿
- **File checksums** and **block level checksums**

1.3.6 Availability vs. reliability

Reliability is linked to the probability of losing data. However, data can be not available even without it being actually lost: for example, the service may be down and the data temporarily unavailable. This is to say that **high reliability does not imply high availability and viceversa**. How can we guarantee both reliability and availability?

Take the following example: suppose T independent storage sets, and have the storage being configured to replicate files R times, with $R < T$. For example, we may write each file a total of $R = 3$ times across a system of 6 storage units. Any of those three may be used by a client when reading, and the load can be spread across the whole system, so that a higher throughput may be attained.

Now, suppose that a failure in one of the storage units is detected. The unit is automatically disabled. Read requests can still continue, as long as there is at least one copy of the requested file in the system. Also, file creation can still take place, as newly written data will be placed in one of the remaining 5 units. Delete requests will still work, and updates can continue, just by modifying the remaining copies or creating an additional copy on the fly on another unit.

Thus, even in case of hardware failure, service can still continue. The broken part can then be substituted by a brand new component, on which data can be copied starting from the replicas that were not damaged.

Some calculations: since there is redundancy, there must be multiple failures in order for data to be lost.

$$Prob(\texttt{Service fail}) = Prob(\texttt{failure of 1 and failure of 2})$$

If the two events were independent, then we would be able to factorize the joint probability and calculations would be easy. However, that is not always the case: we then need to reduce the dependencies as much as possible. A **distributed/dispersed storage** on different servers (or even better, on different data centers) can make different units **independent**. This greatly improves reliability, and allows the aforementioned factorization:

$$Prob(\texttt{Service fail}) = Prob(\texttt{failure of 1}) \times Prob(\texttt{failure of 2})$$

In the case of R replicas, then, the calculation is simply

$$Prob(\texttt{Service fail}) = Prob(\texttt{Single failure})^R$$

So, knowing the probability of a single failure, we can calculate the probability of data being unavailable by raising it to the R-th power.

When actually dealing with managing such a storage system, it is important to have **enough spare parts** to cope with both planned replacements and unexpected hardware failures. Furthermore, the system should not be dependent on timely repair, so that only asynchronous interventions are required (otherwise, there would be the need of having a constant presence of on-site technicians to face failures). In general, the responses that can be automated should be automated, and manual operations should be kept at a minimum. Also, reliability, availability and performance should be tuned as to meet the service agreements.

1.4 Access control and security

Data management is deeply linked to cryptography, as it enables control over data access.

1.4.1 Cryptography: symmetric and asymmetric encryptions

Cryptography solves the following needs:

- **confidentiality**: ensure that nobody can access the contents of the transferred information, even if the whole message was to be listened to;
- **integrity**: ensure that the message has not been modified during transmission;
- **authenticity**: we can verify that the entity we think we are talking to is actually who we think it is;
- **identity**: we can verify who the specific individual behind the entity is;
- **non-repudiation**: the individual can not deny his association with the entity he's behind.

There are two main types of encryption: **symmetric** and **asymmetric**.

- **symmetric encryption**: an input is encrypted by using a certain key, thus producing an encrypted version. If the recipient of the message knows the key, then he can decrypt the encoded message and get the information. This means that in symmetric encryption schemes, both the sender and the receiver (and no other individual!) must know the shared key. The key becomes a common secret between the two.

- **asymmetric encryption**: in this paradigm, the encryption and the decryption use different keys. There is no need for the sender and the receiver to share a common, secret key. Instead, each of them possess two keys, one of which is public (i.e., everyone knows it) while the other is private (only the owner knows it). It is important to stress that, in asymmetric encryption, the relation between the two keys (private and public) is unknown, and no knowledge of one key can be retrieved from the other, even if both the clear-text input and the encrypted text are available at the same time. Also, the two keys must be interchangeable: when a key pair (i.e., a public-private pair) is generated, any of the two can be used as public or private (at least, in theory).

It is important to stress that the two approaches differ also in what cracking them means.

For an asymmetric encryption scheme, cracking means solving a well defined associated mathematical problem, which is usually something like 'find x and y such that $x \times y = N$, with an extremely large N'.

In symmetric encryption, instead, cracking is performed by trying different decryptions and comparing the results with some supposed decryption. Since the message is not known, there is actually no 'ground truth', but we will only consider those decryptions which produce a meaningful result. If a decryption attempt of the code 'fdT57Kb.Ph' yields 'njcdipolz2', then we will be inclined to discard such attempt. In a way, there is the need of a sort of dictionary. In general, a guess of the original message will rely on supposed redundancies, pieces of message that, repeating themselves, give rise to possible weaknesses. From an encryption perspective, then, compression is important as it removes these redundancies, and leaves brute force attacks as the only possible way to crack the code.

Asymmetric encryption can provide a way to ensure **authenticity** and **confidentiality**. For authenticity, it is sufficient for the sender to encrypt some message with his private key. Then, the only valid decryption will be the one provided by the sender's public key. Then, authenticity is attained. Confidentiality, instead, is obtained by encrypting the message using the recipient's public key. This encrypted message is now decryptable only by the recipient himself, as he is the only one who owns the respective private key.

1.4.2 Cryptographic hash functions and digital signatures

An additional tool in cryptography are cryptographic hash functions (CHF). These are a family of functions which, when fed some input string, transform it into a fixed-size output string (called '**digest**'), which can be thought of as a short representation of the original message. They are made in such a way that any small modification in the original input results in massively different digests.

Among their requirements, hash function must be efficiently computable, and at the same time it should be impossible to find a (previously unknown) message that matches a given digest, and there should be no collisions (i.e., two different input strings producing the same digest).

In a sense, cryptographic hash functions are built in order to be easy to compute but extremely hard to invert, meaning that retrieving the input that produced a given digest should be virtually impossible.

The 'no collisions' requirement is actually not satisfiable. We are asking the function to be injective, but if every message (of arbitrary length) is transformed into a fixed-size digest, it can not possibly have this property (if the digest were 10 bits long, every possible 11 bit input should be mapped to the 2^{10} possible digests, which clearly invalidates any injectivity claim).

CHFs can be used in order to create a **digital signature**, and therefore allow **integrity**. Take for example a given message M produced by the user A. By using a CHF, A can generate a digest of M, then encrypt it using his/her private key. The now encrypted digest will serve as a digital signature. When M is sent by A, the receiver can generate a digest for M itself, using the same CHF used by A. Then, by using A's public key, the receiver can also decrypt the signature itself, thus recovering the original digest of M that A produced at the time of sending.

Comparing the two (the digest produced by hashing M and the one retrieved from decrypting the signature) allow the receiver to check the integrity of M.

If any modification was to be made on the message halfway through its transmission, then the two would almost certainly be different, while a perfect accordance would (almost certainly) imply that the message was not altered. Practically speaking, since **collisions** are **extremely rare**, digital signatures can be considered a **proof of integrity**.

An example of actual protocol is **SSL** (here presented in a simplified way). It ensures **confidentiality**, and, if digitally signed, it also ensures **integrity**. Also, depending on how the public keys are exchanged, it can ensure authenticity, identity and non-repudiation.

Suppose A is the sender and B is the receiver, and both have their own private and public key. A initially encrypts the message using his private key, $Priv_A$, and then encrypts once again using B's public key, Pub_B. The doubly-encrypted message is transmitted over the public network, and B receives it.

Now, since B is the only one who can 'remove' Pub_B, as he is the only owner of $Priv_B$, he can turn back the message to its $Priv_A$ encryption. By using Pub_A, then, he can retrieve the original message.

In real world applications, however, SSL uses a **hybrid encryption scheme**, i.e., a mix of symmetric and asymmetric encryption. At first, a symmetric key is randomly generated, and it will be used strictly for the current session. This key will be used to encrypt the clear text message.

Then, the symmetric key itself is encrypted, using the public keys of the recipients of the message. The symmetric-encrypted message will then be sent together with a set of 'digital envelopes', each corresponding to a particular recipient. Each recipient will then be able to take 'his' digital envelope and decrypt it using his/her private key, thus gaining access to the session key and decrypting the message.

Some final words about cryptography:

- Kerchoff's principle: the security of the encryption scheme must depend only on the secrecy of the key, and not on the secrecy of the algorithm;
- on the same note, the algorithms should actually be published, and proved to be hack-resistant for quite some time;
- even the most advanced designs have some degree of weakness, and attacks can be successful: this must be taken into consideration, as there is no perfectly safe option.

1.4.3 AAA: authentication, authorization, accounting

Cryptography solves the aforementioned problems. However, how can we safely share secret keys (for symmetric encryption) and public keys on the internet?

Take, for instance, the following situation: B creates a public/private pair of keys, and tells everybody that the public key he generated belongs to A. People will then send confidential information to A, encrypting with B's public key.

A will not be able to decrypt these information, as he does not have the required private key. B, on the other hand, will be perfectly able to decrypt the messages and access the confidential data. This is not due to a problem in asymmetric encryption itself, but rather a problem in the steps before encryption: nobody can guarantee that the keys B produced are actually his own.

There is the need of a **trusted third party** that can guarantee each other's identities. Some options are PKI (for asymmetric encryption) and Kerberos (for symmetric encryption).

1.4.4 PKI and certificates

If we wanted to exchange public keys with A, we would have one of two options: before using the key, meeting/calling A and check that the keys are right (for example, one could send a mail or make a telephone call, provided that he trusts these services), or we could also use a trusted third party and having it check if the keys are correct (here, certificates play a big role).

PKI (which stands for **public key infrastructure**) provides the technologies to enable practical **distribution of public keys by using certificates**. PKI is a group of solutions for key generation, key distribution, certificate generation, key revocation/validation and managing trust.

A **certificate** is a file which, in the simplest case, contains just a public key and information about the entity that is being certified to own that public key. Everything is then digitally signed by someone trusted (a friend or a certificate authority, or CA) and for which we already have a public key.

The idea is that, when verifying a certificate, we take the contents and generate the hash, we decrypt the signature using the CA public key and compare the two (analogously to what was described above). If the two correspond, then the certificate is verified.

A common type of certificate is the **X.509 certificate**. It contains the following information:

- X.500 subject: who is the owner;
- public key (or info about it);
- X.500 issuer (the entity which has signed);
- expiration date;
- additional arbitrary information;
- the CA digital signature (of the issuer of the certificate).

The idea is that, when **verifying** a certificate, we may end up **'walking the path' of certifications**, as an authority may be subordinate to another one issuing the certificate. This goes on until we reach the root or an explicitly trusted subordinate CA. The point is that trust 'is originated' at some point in the chain, and certificates will, sooner or later, reference that trusted entity.

Certificates can allow authentication, but not in an automatic way. Owning a certificate of individual A does not mean we are individual A - we are not necessarily authenticated. For authentication to take place, some sort of **challenge** must be issued. In particular, if we wanted to verify that a person pretending to be A is actually A, we need to ask that person something that only the true A would know: that is, A's private key.

Person B gets person A's certificate, and then verifies the digital signature (meaning that he can now trust that the public key really belongs to person A). B then generates some random phrase, and challenges A to encrypt it. If A is really A, then he owns the appropriate private key that matches the public one stored in the certificate.

B can then decrypt the message with that public key: if the decrypted phrase matches the staring one, then A is authenticated.

NB: each time a private key is used to encrypt anything, it is indirectly **exposed**. It is a good idea to limit this exposure: to do so, key hierarchies and proxy certificates are used.

A **proxy** consists of a new certificate and a private key. The key pair that is used for the proxy, i.e. the public key embedded in the certificate and the private key, may either be regenerated for each proxy or obtained by other means. The new certificate contains the owner's identity, modified slightly to indicate that it is a proxy. The new certificate is signed by the owner, rather than a CA.

The **less** a key is **exposed**, the **longer** it can stay **active**. Thus, the key from a 'central' authority (for instance, CERN Root CA) should be kept as protected as possible, and 'peripheric' keys should instead take the exposure. That is, the key of a single person should not be held in the same regard as the one from a central authority (i.e., there are hierarchical differences).

Certificates only contain public information and they are digitally signed, so there is no need of particular security: they hold no sensitive information and are protected from tampering by the digital signature anyways. The same is not true for private keys, however: they are confidential and must be stored in a particularly safe place (files protected by passwords, OS protected storage, smartcards).

Smartcards can be ranked on the basis of their security: a bad smartcard is not smart at all, only containing the certificate and the private key, with no security mechanism to protect the private key. A better smartcard would make the private key not readable, only allowing the encryption of random strings using the private key (as a way of challenging the knowledge of the private key itself). An even better smartcard would require the user to enter a PIN code before allowing any operation involving the private key, as to minimize unnecessary exposure of the private key.

The fact is that private keys do indeed become compromised with the passing of time. As a consequence, **certificates can be revoked**. This can be done by requiring the CA to sign a certificate revocation certificate; still, the owner of the certificate must let the world know that the revocation actually comes from him: a difficult task, for which certificate revocation lists (CRLs) are used. This is a non scalable process: the CRL must be kept up to date, and must be checked by the certificate validation process.

An alternative is using expiration dates. When the expiration date is reached, the CA issues a new certificate, choosing one of two approaches:

- creating a new certificate and keeping the same public key, so that the user can keep on using his old private key;
- forcing the certificate to have a different public key, and thus forcing the user to change his private key too.

The choice depends on the intended purpose of the certificate (which is written in the certificate).

The underlying theme in all of these discussions is **trust**. In the real world, trust is easily rankable: we trust a passport way more than a random membership card. When it comes to digital certificates, however, this is not as easy. For instance, there are many classes of certificates with different purposes: a class 2 certificate is designed for people publishing software as individuals, while a class 3 certificate is designed for companies which publish software (and a class 3 certificate offers greater assurance, but also requires a minimum financial stability level).

1.4.5 Kerberos: an alternative to PKI

Kerberos has the same goals of PKI, but achieves them in a different way, which has both advantages and disadvantages. Pros:

- Simpler to manage, keys are automatically managed, easier for users to understand;
- forwardable authentication is easier to implement.

Cons:

- Cross domain authentication and domain trust are more difficult to implement;
- offline authentication is difficult to implement.

Kerberos is built on **symmetric encryption** (and so is completely different from PKI already).

It is based on the existence of a central authority, known as the **Key Distribution Center (KDC)**.

Each user shares a secret key with the KDC, through which secure communication with the KDC is possible: two different users have no way of verifying each other's identities. The KDC is trusted by everyone: its job is to distribute a unique session key to a pair of users that want to establish a secure channel. This way, the two users can use a symmetric encryption paradigm.

An example of a Kerberos session could be the following: person A wants to communicate with person (or service) B. A can communicate safely with the KDC by using his master key, let us call it M_A, and can issue a communication request to B. The KDC then generates a unique, random key for A and B, let us call it K_{AB}. Two copies of K_{AB} are sent to A: one is encrypted using A's master key, M_A, while the other is sent with A's name encrypted with B's master key; this is known as the **'Kerberos ticket'**.

This ticket is actually a message to B: it notifies B that A wants to communicate. Since the value of K_{AB} is only known to A, B and the KDC, if the entity that wants to communicate with B proves knowledge of this key, then B can safely assume it is actually A.

Now, authentication **could** be done in the following way: A sends the ticket to B, who decrypts it and gets K_{AB}. Now, B can generate some random phrase and send it back to A. A will then encrypt the phrase using K_{AB} and send the encrypted phrase back to B, who can now verify if the encryption was done using the right K_{AB}; if that is the case, A is authenticated. The same process could then be repeated from A's perspective in order to authenticate B.

However, Kerberos does not work in this way: the challenge is not simply a random phrase, but it is instead based on current time. This is because of the possibility of a **reflection attack**:

1. a client (the victim) initiates a connection to the server (the attacker);
2. the attacker initiates a new connection to the victim;
3. now, the victim issues a challenge for the inbound (arriving) connection from the attacker;
4. the attacker can 'recycle' this same challenge back to the victim: this is because another connection is already open, and the attacker has still to issue a challenge
5. the victim computes the correct response to the challenge posed by the attacker. In a sense, the victim responded to its own challenge;
6. since the attacker now knows the correct answer to the challenge, he can use it to respond to the original challenge posed by the victim. The attacker has now access to the victim.

In order to prevent this, **Kerberos acts in the following way**:

- A sends the ticket to B (recall that the ticket contains A's name and K_{AB}, boh encrypted using B's master key, M_B);
- A has to prove that he knows K_{AB}. A's name and the **current time** are also sent, encrypted with K_{AB} (this is called **'authenticator'**);
- B takes the ticket, decrypts it and retrieves K_{AB}. Using it, it decrypts the authenticator and compares the name in the authenticator to the name in the ticket.
- if the time is correct, this provides a proof that the authenticator was indeed encrypted with K_{AB};
- B can also avoid attackers by rejecting authenticators with time already used or with the wrong time altogether.

The reason behind the encryption of time is that it serves as a way to prove the knowledge of the common secret, that is, the common key K_{AB}. This implies that the time synchronization is fundamental for Kerberos to work properly. If B wants to avoid replaying an earlier attempt, he needs to keep track of all previous times.

This approach, however, does not scale. The alternative is to use a time window: B only needs to remember the times in the past (for instance) 5 minutes. Then, comparing the encrypted time with the recorded times of the past 5 minutes, B can be sure that the time has not been used before: the request can be accepted. However, if the time falls out of this window, B can not be sure that the time he is receiving has not already been used. As a consequence, the request is not granted. However, this provides a hint of what B's time is.

Now that A has proved his identity to B, A wants B to prove his own identity. To do so, A can send a flag in his request. After B has authenticated A, he takes the timestamp sent by A, encrypts it with K_{AB}, and sends it back to A. Now, A can decrypt this and verify that it is the same timestamp A originally sent. By doing so, A has authenticated B, because only B could have decrypted the authenticator (as only B has the required master key, M_B. Recall that B receives both a ticket, encrypted with M_B, and an authenticator, which is encrypted with K_{AB}. Only B can actually retrieve K_{AB} by decrypting the ticket!). By sending the timestamp back, encrypted with K_{AB}, B has proved he can access the information inside the authenticator. The time is the only piece of information that is unique in A's message.

After all these steps, **A and B share a secure key** K_{AB} which they can use to communicate.

Actually, though, Kerberos is more complex, including an additional extra step. When A first logs in, he actually asks the KDC for a **ticket granting ticket**, or **TGT**. Stored inside the TGT is the session key, K_{AK}, to be used for communications between A and the KDC throughout the day. So, when A requests a ticket for B, he actually sends the KDC his TGT plus an authenticator. The KDC then sends back the session key K_{AB}, encrypted using K_{AK} (so the master key is not actually used). This introduces a sort of hierarchical system of keys:

- **short-term key**: the session key, a key shared among two entities for authentication purposes. Generated by the KDC, it is always sent on a secure channel, as the security of the whole Kerberos system depends on it: it is encrypted using the Ticket Granting Services (TGS) key.
- **medium-term key**: the TGS key, a secret key shared between entities and the KDC in order to obtain session keys. It is encrypted using the master key.
- **long-term key**: the master key, a secret key shared between each entity and the KDC. It must be known to both the entity and the KDC before the Kerberos protocol can take place. It is generated at the moment of the domain enrollment process and it is derived from the creator's password. It is sent through the secure channel.
- **the secure channel**: provided by the master key shared between the workstation and the KDC (in which case it is derived from the workstation's machine account password).

As already mentioned in the case of PKI keys, the keys with the longest lifetime are the ones which are less exposed. This makes the secure channel the most long term and the session key the shortest. <slide 133...>

1.4.6 Authorization

Authorization implements the **access control**, that is, the information describing what the end-user can actually do on computing resources. It is the association of a **right**, a **subject** and a **resource**:

- right: use, read, modify, delete, open, execute...
- subject: person, account, computer, printer, room...
- resource: file, computer, printer, room, information system...

Of course, an authorization can be **time dependent**; for example, it could be restricted to a window of some hours.

The authorization process is the verification that the user has all the required permissions to access a given resource.

In practice, each resource has a linked list called **Access Control List (ACL)**. This is part of the metadata regarding that resource, and it is composed of Access Control Entries (ACEs). Every

ACE contains the subject (person, account, computer, group...) and the right (use, read, and so on) that the subject has on the resource. The time of the right may also be present.

There are two main ways of using ACLs:

- only 'allow' permissions are possible;
- both 'allow' and 'deny' are possible.

In the second case, additional rules must be defined in order to avoid contradictions when dealing with groups of users (for instance, such rules may be that denials come first and ACEs are processed in the order they are defined).

Authorization must also take into account the **identification** of users. There are two main ways to do so:

- identifying users by 'login names' or by the 'subject' found in the certificate;
- identifying by a Globally Unique Identifier (GUID) or Virtual ID, such that 'login name' and 'subject' only become two attributes of the account.

The latter is to prefer, as it has much better performance and is more flexible. However,since GUID/Virtual ID are instance-specific, moving the data requires modifications to the information.

There is a fundamental problem about authorization: where should the ACL be stored? On a database or within the resource itself? And how should ACL be verified? It can be delegated to the OS: the ACL are set on the file system and the process accessing the resource impersonates the credential of the user. It can also be managed by the application itself: the permissions of the owner of the request must be verified by the Data Management software on every request.

Furthermore, ACL must be supported on every node of the file system: every folder! Then, **inheritance** must be handled properly via a set of flags. Also, calculating the resultant permissions in real time when a particular file is accessed is indeed possible, but requires low level code optimization. Otherwise, we must resort to compiling the resultant permissions (file permissions + inheritance permissions) with the file, which becomes inefficient when changing permissions.

In a sense, calculating permissions in real time allows for faster writing, while the other approach behaves better in reading. In real life, a mix is implemented.

But that's not all: what about groups of users, and roles inside a group? ACLs can be granted to aggregate groups of users. But then again, when should the group resolution be done? Either at run-time, when the resource is accessed (at the cost of additional database lookups: bad performance!), or at authentication time: that is, when the user logs in, an authentication token (which contains the login name and all the groups the user belongs to) is produced. The ACL is the compared to all the groups that particular user belongs to, and authorizations are computed accordingly. This is obviously better, but, in the case of a change in the groups memberships, it requires to log back in (that is, a re-authentication process) in order to be effective.

And what is the scope of the group? Is it something local to the storage element? Local to the site? Or global? Another important concept in authorization is **granularity**: how far does the protection of ACLs go? Is it restricted to pool, directories, files, parts of files? The best way to act is to defining the security model in a proper and consistent way:

- No access by default, to be overridden by 'allow permission' ACLs;
- World access (everywhere) by default, overridden by 'deny permission' ACLs;
- an arbitrary default access that can be overridden by either one of the 'allow' and 'deny' ACLs (this one requires some more attention, as it must be able to handle permission conflicts in the case of inheritance and groups).

1.4.7 Accounting: distributing responsibilities

By **accounting**, we mean a list of actions (and related metadata, like who, when, what, where) that enable **traceability** of all subsequent changes and transactions rollback (i.e., enabling Ctrl+Z). There may be multiple possible levels of accounting: simple logging (i.e., saving the history), logging and journal of all transactions (allowing to both know the history and perform rollbacks if necessary). A good accounting strategy can be a valid alternative to a strict authorization scheme: users get more power, but also receive a share of responsibilities.

1.5 Scalability

1.5.1 Cloud storage

By cloud storage, we mean storage (generally hosted by third parties on the internet) that offers a model of **virtual pools**. This makes cloud storage highly scalable and flexible; also, it makes paying 'a la carte' possible (only the needed/used services are actually paid for, everything is highly personalized, so to speak).

Cloud often employs simple interfaces to access storage, and does not employ the posix interface: this makes it possible to deploy scalable infrastructures (recall that posix, while otherwise good, does not allow scalability). Cloud storage can indeed be simpler than traditional storage: there is a single pool where all data go, so that the system is fast, reliable and outsourced. There is a unique quality of service, and it is economically interesting for small/medium data sizes.

However, it can become **simplistic**: a single pool with a **unique quality of service** is not suited to the requirements of scientific computations, as it can easily become expensive and/or inefficient.

Cloud storage employs **Distributed Hash Tables (DHTs)** in order to empower scalability. The storage (that is composed of multiple clusters/data centers) is federated under a single name space. Some keywords when discussing DHTs are hash tables (and distributed hash tables), hash algorithms and collisions.

- **Hash table**: a hash table is a data structure that uses a hash function to map keys (e.g., a person's name) to their respective values (e.g., the telephone number). A hash table implements an **associative memory**. A hash function is used to transform a key into the index of an array element where the respective value is stored. In a hash table, the cost for a lookup is independent of the table's dimension: this is a case of **perfect scalability**. Also insertions and deletions of key-value pairs can be performed at a constant cost, independent of the number of elements.

- **Hash algorithms** and **collisions**: the hash table allows the retrieval, insertion and deletion of elements in a size-independent fashion. This is because the maximum number of entries in a hash table is determined by the chosen hash function. For example, using CRC16 as a hash function allows the creation of hash tables with 2^{16} (65536) entries, while SHA1 allows tables with 2^{160} ($> 10^{48}$) entries.

  Whatever the size may be, hash functions are subject to **collisions**: a collision is the event of two (or more) distinct inputs (keys) being associated to a single output (value). Collisions must be handled appropriately.

  An important quantity when dealing with hash functions and hash tables is the **load factor** (also known as fill factor), which is defined as the ratio between the number of stored entries and the size of the table's array. As the load factor increases, so do the probability of collisions and their cost. However, keeping the load factor too low results in wasted memory, as large portions of the hash table are left unused. The two extremes for the load factor (0 in the case of wasted memory, 1 in the case of too many collisions) are to be avoided.

A way to do so is **resizing** the hash table: this requires a full or incremental rehash of all keys (for instance, rehashing may happen when the load factor goes over some pre defined threshold, say, 0.75. In this case, the table could for instance get doubled in size, and all hashes recalculated so that the entries spread over the entirety of the new hash table). Rehashing breaks scalability. A workaround is using a hash function that preserve the key hashes after the table gets resized. This is called **consistent hashing**. This becomes essential when dealing with distributed hash tables, since the load may be distributed among various servers, and adding/removing servers would otherwise require a complete rehashing of all stored data.

- **Distributed hash tables**: in order to hash among different servers, a keyspace partitioning scheme is used; it allows hash values to be distributed among the various servers. An **overlay network** is also established between all nodes in the network of servers, so that any one of them can route a certain data request (e.g., reading or writing some particular entry) to the server containing the right key. Once again, it must be stressed that the only viable option is consistent hashing, which allows servers to be added or removed and only affecting adjacent nodes.

  When implementing a distributed hash table, **reliability** and fault tolerance can be obtained by simply replicating values (or adding error correction information)on n adjacent nodes. In the case of widely spread nodes, the reliability can be calculated (as distances between data centers allow us to make the assumption of independence of failures). All of these features make distributed hash tables highly scalable, as data reading and writing are independent of the size of the hash table itself.

- **Keyspace partitioning**: two ways of performing partitioning are hash partitioning and range partitioning.

  Range partitioning requires a table through which objects are mapped to their storage instances, which makes adding instances easy (an example may be partitioning based on the date of the entry, so that temporally contiguous entries are stored in the same location).

  Hash partitioning requires all data to be moved when adding a new instance, unless consistent hashing is used (another possible workaround is starting with a large number of storage instances that are deployed on a few servers. When data grows, the self-contained instances can be moved to another server).

  The partitioning itself (i.e., associating the right entry to the right server) can be done in the client (if the client knows which server owns a particular key), in a middle-layer server (the so called proxy assisted partitioning: an intermediate database knows or calculates which server owns a key) or in the server instances (in the case that only the server knows or can calculate who owns a key; in this case, the clients send their queries to a random node, and the overlay network then manages the request and forwards it to the right server).

In practice, DHT implements storage elements called **buckets**; the various Application programming interfaces (APIs) allow to deal with the buckets. Inside one bucket, one can store:

- a file system;
- a database;
- a disk image;
- in general, objects.

At the same time, every functionality that limits scalability is dropped. This includes:

- hierarchical storage in folders/directories;

- permission inheritance;
- authorization based on groups/roles.

<checksums, ? slide 160>

1.5.2 Block storage

It is a level of **abstraction** for storage, where data is organized in blocks, where a block is a set of data with **fixed size**. When the size of our data exceeds one block, data is stored in multiple blocks. Whenever the data is not an exact multiple of block size, the last one is not filled, but it is left partly empty.

It is the important to choose the block size in an appropriate way, so that the **empty space is kept at a minimum**. Small block sizes reduce the amount of wasted storage space, but increase the number of blocks that must be read in order to get a constant amount of data. On the contrary, large blocks allow for less readings needed, but increase the wasted space. Whatever the chosen size is, block storage leads to space inefficiency, as data are rarely (if ever) exact multiples of block size. Then, why using block storage at all? There are multiple reasons:

- block storage allows variable size storage to become somewhat **standardized** and fixed size;
- it also allows a more effective **load balancing** of the storage (we just know the amount of blocks and not the sizes of each and every file);
- **error correction algorithms** are more easily implemented in a block storage, even across multiple servers. It also allows to have data correction with less wasted space compared to data replication;
- **performance** is **proportional** to the deployed **resources**, and less dependent on access patterns (since 'hot blocks', i.e., the most frequently requested blocks, are scattered on multiple servers);
- software **abstraction** is identical to traditional hard disks, as a **block is analogous to a hard disk sector**. Block based storage can be actually mounted and seen as virtual hard disks; this is typical usage for VMs. Also, due to striping and replication across the various nodes of the system (basically what happens in RAID0 and RAID1 architectures for physical disks), performance and reliability increase. Block storage is better than physical hard disks since it provides more functionalities: for instance, virtual snapshots (allowing the creation of journals and rollbacks, that is, the ability of doing Ctrl+Z), but also resizing/renaming/copying etc. of disk images/partitions.

1.6 Analytics

It is an umbrella term, under which many different attempts at finding patterns in data are grouped. Analysis is typically done on large computer networks, which process large amounts of data. Our main concern is actually on the data side of things rather than on pure computing.

On this note, the data we face is usually **unstructured**, meaning it does not have a predefined data model nor is it organized. It is typically text based (documents, email, messages) and contains heterogeneous data (dates, numbers, assertions and so on). It usually contains irregularities, ambiguities and errors: every information is wrong until proven right. This makes unstructured data particularly hard to process using traditional programs.

A typical problem is that, in a distributed computing environment, data and CPUs are not necessarily in the same place. This separation can lead to severe inefficiencies, due to round trip times (i.e., the time between the message being sent and the acknowledgment of the receipt arriving) and latency in general.

In this framework, **map-reduce schemes** allow partial data processing directly on the site where data is stored (i.e., the data servers). This type of processing is easy enough to implement when the type of processing is uniform and well defined; however, difficulties arise when the processing interferes with the need of the server to provide data with high efficiency (the server is in the condition that it must provide two services at the same time, both computing and providing the data to other requests).

Map-reduce is often based on open source implementations of the Hadoop ecosystem. The analysis consists of:

- a design **splitting** the processing between multiple parallel data processing and aggregation algorithms;
- a '**map**' program, distributed and running on every data node. The map operation performs filtering and data analysis;
- a '**reduce**' program, which aggregates the results coming from all the 'map' jobs.

Of course, such a paradigm requires parallelization of the process among multiple servers, while also providing a way of managing all status, communications and data transfers between all components of the system (including redundancy and fault tolerance).

1.7 Data replication

Transferring information between pools, under many different scenarios (LAN and/or WAN, master/slave, multiple masters, and so on...).

- **One way (master/slave)**: the slave copy is read only, so that only the replication operation can alter its contents. Each operation involving the master can then be replicated on the slave. To ensure consistency between master and slave, at fixed times, the files are scanned and any difference is found and replicated (the scan can be at different levels, from simple metadata up to the content of each file).
- **Two ways (multiple masters)**: all pools are read/write. Each write operation can be intercepted and replicated to other masters. This obviously gives rise to conflicts, which require handling: for example, possible policies can include defining a 'super-master', last modification wins, duplication on conflict (thus keeping everything). Once again, consistency is guaranteed by a scheduled scan at various levels.

  In multi master systems, something as simple as a file being present in one pool and not in the other, can be faced in two ways. The file could be copied in order to fill the void in one disk, or erased from where it is, as to bring everything to the same level. What should be preferred? One option may be remembering the last sync date and compare the file's modification/creation dates; another may be simply keeping a database of files of last sync and log all requests. In all of this, the possibility of pool errors must also be taken into account (maybe the file is absent in one of the pools just because that pool failed, and not because it was actually erased).

In general, **replication errors** are similar to those in the multi master case, and there are many possible policies re synchronize the pools. For instance, a reference pool may be defined, or a third copy may be employed.

When data is replicated, **unique file identification** is important, as it allows moving/renaming and so on. There are many options, like filename, filename+size+last write time, filename+size+last write time+create time, hash of file content, hash of file content+size+last write time, server-side file ID...

Aggregated transfer efficiency is bound to the size of the data we want to transfer (actually only true for storages that require sequential access). Datasets predefine the ‘access patterns’ to data (different datasets may have different policies for accessing the data). In general, recalling multiple files from the same dataset guarantees maximum efficiency (no need of jumping from dataset to dataset, with the risk of having to change the access pattern halfway through). However, this is also bound to physical implementation: co-locating dataset files on the same media is better for tapes, while scattering datasets among various media is better in the case of disks, as it provides the maximum throughput.

The scheduling of data replication can follow many approaches:

- real-time, triggered by data changes: this way, data are always synchronized;
- at scheduled intervals: easier to keep track of the history and to make rollbacks, backups;
- manual replication;
- custom.

1.8 Data caching

The cache is a **fast storage** area, which is used to store frequently accessed data in order to increase the speed (as it avoids using disks). The storage is **temporary**, meaning that after a while the cache erases its contents. The data to be cancelled is decided by the **caching algorithm**, which tries to guess what can be cancelled and what can not, based on the guessed possible future needs. For a cache to be effective, the number of correct guesses (hits: data that was kept in the cache and then actually reused) must be higher than the number of incorrect ones (misses: data that was either cancelled but then needed again o data that was kept without it actually being used again).

Adding cache memory increases the cost but not the capacity; however, it may increase performance (depending on the efficiency). **Efficiency** is defined as number of hits/number of requests. Depending on how the data is accessed, cache can have different levels of efficiency:

- streaming data: data in never requested again. Efficiency = 0%;
- true random access: in this case, efficiency = (cache size)/(total size); since the access is true random, the efficiency can be computed as the ratio between the two sizes. No further information can be obtained by past accesses (as we are randomly accessing);
- in real life: efficiency depends on how much we can predict the access patterns, so that cache usage can be optimized.

On that note: how can we predict data usage in order to deploy cache in an optimal way? This amounts to predicting future data requests from past access patterns. Some approaches are

- Temporal: recently accessed data have a higher probability of being accessed again;
- Spatial: data which are close to the recently accessed have a higher probability of being accessed again;
- Read ahead: data ahead of recent requests have a higher probability of being accessed than data behind (implying that there is some measure of sequentiality in the way data is accessed).

Some algorithms are then LRU (least recently used), MRU (most recently used), LFU (least frequently used) and so on.

However, the efficiency is not only dependent on the chosen algorithms. Cache size also plays a role, even if it is not a linear effect:

- below a minimum size, cache is useless (as there are only misses; the cache can not store enough information, so re-fetching the data happens very often);
- above a maximum size, adding cache is once again useless (as the hit/miss ratio does not increase after a certain point).

The result of adding cache size on efficiency is a 'knee shaped' curve, with a region of fast improvement followed by a region of plateauing.

Additional complications come in the form of the **cache volatility**: if we perform write operations, these must be 'flushed' (i.e., passed down) to the slower (but permanent) memory. This operation can be done synchronously (as soon as data is changed) or asynchronously, by scheduling (this choice has some implications on the reliability: what would happen if there was a power outage between two scheduled updates of the permanent memory? All changes which have still to be 'flushed down' would be lost).

Another complication is posed by the possibility of dynamically changing the caching algorithm in order to adapt to the length of read and write requests. Also, if we assume that each IO operation has a cost, the choice of a cache algorithm may depend on the 'size' of the IO requests, and could perform some kind of aggregation of multiple small IO requests.

In the end then, how can we tune the cache? The idea is to start from some theoretical model from which we define the algorithm. Then we start the system and measure hits, misses and latencies in real time. Then, we measure the sensitivity of the caching efficiency towards all adjustable parameters (cache size, cache algorithm, requests size...) an tune these accordingly in order to obtain the desired compromise between efficiency and cost.

1.9 Monitoring and alarms

An essential component in Data Management, monitoring is a way to **measure** the key **performance** indicators of the service. There are several of these parameters that need monitoring:

- Network performance, memory and CPU load for each server;
- IO rates for every disk;
- Popularity of each file (i.e., how much each file is requested, so that hot spots can be identified and managed);
- hit and miss analysis in caches;
- IO rates per user (so to identify the most active users);
- Length of the pending request queue (maybe to anticipate bottlenecks);
- Access latency.

Monitoring the performance, however, means monitoring from some point of view. We can either take:

- **service viewpoint**: aggregated performance of the system, statistics, global indicators (everything that the provider of the service may be interested in; no specific info about this or that user, but rather info about the provided service);
- **user viewpoint**: performance is measured from the user's viewpoint: queuing times, latency, throughputs (everything the user experiences may be monitored, so to identify user-specific problems or better grasp other critical aspects in the service that may not be identifiable from an aggregated viewpoint).

Monitoring data must be aggregated in real time so to provide the maximum, the minimum, the average, the current values in time. These values must constantly be compared to those in the Service Level Agreement and linked to the alarm system (as soon as the service quality drops under some threshold, define in the agreement, an alarm must be issued in order to solve any possible underlying problem). Comparing current values with logged values in the may also be useful in order to perform statistical analysis, identify trends...

1.10 Quota

This is the **limit** that the admin sets on the **usage of storage resources**. Storage resources include, but are not limited to, storage space, transfer bandwidth, request per second...

Quotas can be applied to accounts, volumes, pools, directories... and can be either **hard** (the quota verification is done synchronously with the request; while this adds another operation to be done concurrently with the request, hard quotas allow the system to be more consistent) or **soft** (quota verifications happen asynchronously; this allows for better performances, as there is no added operation during the request). Quotas can either be linked to **physical** limitations (for instance, there are not enough disks, so each user can have a maximum storage space) or can be **arbitrary** (as in our lab. The arbitrary framework could give rise to an overbooking problem, that is, the sum of all quotas is higher than the effective physical size of the storage.).

Some approaches can be the following:

- Quota per user: if user A has a big file in pool A, he can not write in pool B as he has already exceeded his quota;
- Quota per pool: if user A has a big file in pool A, then user B can not write a file there, as pool A has already exceeded its quota;
- Quota per user and per pool: a mix of the two, which makes it harder to understand for the end-user.

2 Campana hands-on

2.1 I/O performance

When dealing with distributed data, the measurement and general assessment of IO performance is fundamental. There are 4 main observables that summarize the IO performance:

- **Bandwidth** (IO volume/time);
- **IOPS** (IO operations/time);
- **latency** (the signal delay, time/IO operation);
- **blocksize** (payload/operation).

The last item is linked to the way a typical OS works: files are organized in blocks of a certain size, and each block is transferred in a single operation. IO can be **synchronous** or **asynchronous**. In synchronous IO, each process (or thread), after entering the IO operation state, enters a wait state until completion of the operation. In asynchronous, the process does not wait: instead, it sends the kernel an IO request. Once the kernel completes the IO operation, the process is notified. Another possible distinction is between local and remote IO, and **sequential** vs. **random**. Sequential means that the reading occurs on contiguous memory registers, while random implies no particular order. Generally speaking, random access is slower than sequential access. Even when drawing randomly, it is usually better to do so in order. As examples:

- end user analysis: employs local and synchronous IO;
- video streaming, bulk data analysis: sequential and synchronous IO;
- big data analysis: remote, asynchronous IO;
- selective data analysis: random access, asynchronous IO.

Different types of memory have different performances on the 4 metrics introduced before. In general, performance wise,

```
Tape < Disk < SSD < RAM memory
```

However, the downside of RAM lays not only in its cost but also on the fact that it is volatile: it only exists as long as the process (in general, the OS) is active. Some useful Linux commands for checking IO performance are:

- `time`
- `dd`: copy/block IO tool;
- `vmstat -n 1`
- `iostat -x 1`
- `dstat`
- `top`, `htop`, `iftop`, `iotop`: task overview, a sort of task manager
- `strace <program> [program args]`: trace system commands of a program. If `-c` is added, then it counts the system calls.
- `sync`: flush dirty pages as root or user;
- `echo 3 > /proc/sys/vm/drop_caches`: clean the cache

Some commands:

- baseline performance:

  ```
  dd if=/dev/zero of=/dev/null bs=1M count=16000
  ```

- writing to a hard disk:

  ```
  dd if=/dev/zero of=/data01/16G bs=1M count=16000
  ```

- reading from a hard disk:

  ```
  dd if=/data01/16G of=/dev/null bs=1M count=16000
  ```

  This command results in really low times. This is due to the fact that the data has been saved in the cache, which makes accessing it much quicker. If we want to know the actual speed, we first need to clear the cache, and then perform the reading again:

  ```
  echo 3 > /proc/sys/vm/drop_caches

  dd if=/data01/16G of=/dev/null bs=1M count=16000
  ```

There are two main caching strategies: **write-through** and **write-back**. In write through, the cache is used only after the changes have been made permanent (acknowledged). In write-back, instead, the changes are stored in the cache, and are acknowledged from time to time. Of the two methods, write-back is faster, as data does not have to be saved on disk each time it is modified; however, it is also a riskier approach, as any failure (e.g., a simple power outage) may erase any change that occurred after the last commit to the disk. Linux based systems use the write-back paradigm: as a consequence, if we want to make sure that a change has been saved in the disk, it is not sufficient to re-open the file after the modification, but we must look at it from the OS.

- taking a look at how the blocksize impacts actual performance: by trying

  ```
  dd if=/dev/zero of=/dev/null bs=N count=1000
  ```

  with N in [1,10,100,1000,10000,1000000] we can take a look at how the bandwidth (MB/s) varies. For both bandwidth and IOPS, there is a bottleneck somewhere in the set of N.
  For increasing blocksizes, the bandwidth increases accordingly, while the IOPS decreases.

It is also important to take into account how **latency** impacts the analysis of data. Take ROOT as an example. Here, the software issues thousands of small read requests to iterate over data structures inside ROOT format files. What if such an application had to retrieve that data from a remote location? Then latency may become highly impactful, and reduce the IO efficiency. ROOT itself employs various techniques to reduce the impact of such latency. Let us use the process of ordering a beer as an analogy:

- **uncompensated**: this is the easiest way to act, since no particular countermeasure against high latency is taken. A beer is delivered only after it has been requested. This is the best way to act when a fast disk is available. This is the method used by Python notebooks, and this is why notebooks are not used in a HEP context.
- **pre-fetching**: carrying one beer to a table may take a long time. It is possible to cut on the overall waiting time by simply carrying around more than one at a time. This way, the waiting is divided among multiple orders. If the waiter knew how many beers each customer wanted, he could bring more than one at a time and then improve the efficiency. The same is done by ROOT, which can 'learn' each user's preferences over time, and optimize itself accordingly. In a sense, pre-fetching requires some previous knowledge.
- **asynchronous pre-fetching**: as soon as a beer (the data on a disk) arrives, a customer may ask for another one. Then, the time spent on drinking (processing the data) would partially overlap with the time required for the second beer to arrive. This way, efficiency is improved without requiring previous knowledge.

2.2 Hash and lookup tables: building a cloud server using Redis

Hash tables are managed using Redis, a tool designed specifically for this. In the following, `aperin` is the user name when logging in Cloudveneto.

```
ssh aperin@cloudveneto.it
ssh root@10.67.22.14
cd /tmp/
```

We now download Redis:

```
wget http://download.redis.io/releases/redis-5.0.5.tar.gz
```

Now print what's inside the folder:

```
ls -latr
```

where '-latr' means long listing (l), all (a), sorted by modification time (t), reverse (r). So we are getting the most recent changes as the top results. Inside the folder there is a file we must unzip and untar:

```
gunzip redis.5.0.5.tar.gz
ls -latr
```

We successfully unzipped the file, but it is still in '.tar' format. This is a Tape ARchive format, which is a file format used to store on tape and other mass memories. We now untar it with the following command:

```
tar -xvf redis-5.0.5.tar
```

where the options '-xvf' mean that we want to extract (x) in a verbose fashion (v), by giving the name of the file to untar (f). Now inside the folder 'redis' there is a README.md. Inside there is an explanation about redis itself, together with a set of instructions. We now have to compile the package by using the 'make' command:

```
cd redis-5.0.5
cat README.md
make
```

When we run `make` inside a directory, the command looks for every '*.c' file and compiles it. Now:

```
ls
less Makefile
cd src
./redis-server
./redis-cli
```

Now, the IP of the server is `127.0.0.1:<some port>`. This is the localhost IP. The server is now running: from now on, we will be working as Clients.

```
quit
```

RECALL: if you want to login as (in my case) `andreaperin`, the following commands must be typed: `exit` and then `ssh andreaperin@10.67.22.14`.
Now, from root access, we can 'impersonate' any other user. To do so, we can use the 'switch user' command, `su`:

```
su andreaperin
cd
/tmp/redis-5.0.5/src/redis-cli
```

This will now work as before, even if we are no more root users.

```
quit
ls
```

However, the command `redis-cli` will not work outside its directory. We must use an environment ('env') variable:

```
export PATH=/tmp/redis-5.0.5/src:$PATH
redis-cli
```

and now this will work properly even outside its directory. A command is actually a program; when it is launched, it must be retrieved somewhere. Exporting the PATH variable makes the OS look for the command in the mentioned directory. That is why after exporting the command works everywhere. We can also take a look at the contents of the PATH environment variable by typing

```
echo $PATH
```

This returns a list of places where the OS looks for the command; when it is found, it is executed. Commands can be built: after all, they are executable files, e.g. a C implementation.

IMPORTANT: if two commands have the same name, UNIX just runs one of them. It scrolls through the list of paths and the first time it finds the wanted command, it executes it. Only the first encountered file is found! If we want to overwrite a command, we must append the path where our implementation is to the left: this way, our brand new command will be found before the other and thus take priority.

Now go to the exercise at the link

```
https://apeters.web.cern.ch/apeters/csc2018/cloud/hashtable.html
```
Then, type in the terminal
```
wget https://apeters.web.cern.ch/apeters/csc2018/
_downloads/4eea42a0909a188c81ba98ae00d64083/cloud.sh
```
This will download the required stuff, that is, a bash script library concerning hash tables. We will work on this library by understanding its components and also extending it. Then,
```
less cloud.sh
```
This is a nice command: it prints the contents of the file (in this case, a bash script), and allows browsing up and down the lines of the file by simply using the up and down arrows. We will have to edit this file in order to go on with the exercise. Choose an editor: in my case, 'vi'.
```
vi cloud.sh
```
For now, one may try and make the library executable by means of the following command:
```
source cloud.sh
```
The `source` command takes every executable file inside (in this case) 'cloud.sh' and translates it into the OS environment. We may be tempted to simply run this file:
```
./cloud.sh
```
but this only results in a 'permission denied' type of answer. We can then take a look at the permissions by typing
```
ls -l cloud.sh
```
where the option '-l' lists also permissions. The resulting output is something the likes of
```
-rw-r--r-- 1 root root 10572 Oct 10 2018 cloud.sh
```
And, as can be seen, there is no 'x' for anyone (recall that 'x', in the permissions language, means execution).

Let us move on with the exercise: implementing a cloud server. Let us define 8 DataBases (DB) via an 8-disk system, each identified by a number from 1 to 8.

Hash Key	Hash Value	KV Server Host	Port	DB
0,1	1	localhost	6379	1
2,3	2	localhost	6379	2
4,5	3	localhost	6379	3
6,7	4	localhost	6379	4
8,9	5	localhost	6379	5
10,11	6	localhost	6379	6
12,13	7	localhost	6379	7
14,15	8	localhost	6379	8

The situation is exemplified in table1. To build such a scheme, we need to first define a hash function. In our case, that will be SHA1. Typing the following (NOT INSIDE THE DIRECTORY!)
```
sha1string /etc/passwd
```
will return the hash of the string '/etc/passwd', that is,
```
63ce9c1433c0fad87dcc9d5d22081acc7ff60df4
```
What if we want to save the output of a hash function to a variable? Simply type
```
hash=`sha1string /etc/passwd`
```
Mind the backticks! We can now look at what is inside the variable 'hash' by typing
```
echo $hash
```
We now want to keep only the first character of the digest. To do so, we will use some UNIX magic: both `echo $hash` and `echo ${hash}` return the whole content. However,
```
echo ${hash:3:5}
```
means that we want the portion of string starting from 3 and of length 5: '3' is the offset (places

from the beginning) and ‘5’ is the length. This can be used in order to get only the first character of the string, by typing

```
echo ${hash:0:1}
```

and this can itself be saved into a variable that we will name ‘hashkey’:

```
hashkey=‘echo ${hash:0:1}‘
```

Nice, but hashkey=6, so we need to convert it into an octal representation (for the hashkeys). To do so, we use the function ‘h8d’:

```
h8d $hashkey
```

We now upload the file `/etc/resolv.conf` in the system, by using the following sequence of commands:

```
filename="/etc/resolv.conf"
hash=‘sha1string $filename‘
hashkey=‘echo ${hash:0:1}‘
index=‘h8d $hashkey‘
upload $filename $index $filename
```

if this worked, an ‘OK’ should be printed.

IMPORTANT: `redis-cli` must be ‘exported’, otherwise this won’t work. `redis-cli` is reset each time we make a new login; ‘bash.rc’ must be edited if we want a permanent addition of the path.

We can now try and download the file we just uploaded:

```
download $index $filename /tmp/resolv.conf.downloaded
```

and we can check the eventual differences between the uploaded file and the downloaded one by typing

```
diff /tmp/resolv.conf.downloaded /etc/resolv.conf
```

This command should output no result if the two files are identical. Otherwise, it should output the places where the two files are different. To test this, just do

```
vi /tmp/resolv.conf.downloaded
```

and start modifying from there.

IMPORTANT: only one possible download with a given name for a single machine. Permission is denied to whoever tries downloading the file after the first one has been downloaded. To avoid this, just add a small specification (if it is deemed necessary).

We can now check the contents of server 3 by typing

```
list 3
```

How can we remove an uploaded file? We must use the function `delete`:

```
list $index
delete $index /etc/resolv.conf
```

So we can upload a file, download a file, remove a file, listing all the files. Something more high level: a function to upload a list of files. Download a dummy set of files by typing

```
wget https://apeters.web.cern.ch/apeters/csc2018/
_downloads/3a9788b203f55cf8b78398bebbe880c3/pictures.tgz
ls -latr
```

What is a tgz file? It is a file that is both compressed and tarred. Untar/unzip as usual:

```
tar -xvzf pictures.tgz
```

x=extract, v=verbose, z=the file is to unzip, f=the file name is given. If we list the contents once again, a directory ‘pictures’ has been created. Let us move there:

```
cd pictures
ls -latr
```

all the pictures are listed there. We can find the number of pictures inside the directory by typing

```
ls pictures | wc -l
```

There are 128 of them.

```
find pictures -type f
```
This searches files of generic (the meaning of 'f') type inside 'pictures'. If `find pictures -type b` were to be typed, no files would be returned, as 'b' means 'binary'. Now, for some nice loops in bash:
```
for name in ‘find pictures -type f‘; do echo "my file is $name"; done
```
this searches in 'pictures' all the files (since 'f'). It loops over all the 'name' in the list of names returned by 'find pictures -type f' (which is formatted as a column of names, conveniently). Then. for each of them, it just prints (because there is the 'echo' command) "my file is ...". Alternative syntax (not in line):
```
for name in ‘find pictures -type f‘
do echo "my file is $name"
done
```
the semicolon is just a portmanteau for newline. We can now implement a 'cloud_upload' function: just modifying the one in 'cloud.sh'. After modification, we must run 'source' once again, so to make files executable. **In case of an error while editing**: if an error at line x, type `vi +<line number> cloud.sh`.

Then, use a for loop with cloud_upload:
```
for name in ‘find pictures -type f‘
do cloud_upload $name $name
done
```
An interesting question is: how were the files spread amongst the various disks? Let us find it with a for loop:
```
for name in 1 2 3 4 5 6 7 8
do list $name | wc -l
done
```
Recall that 'wc' is a word count function. The set '1 2 3 4 5 6 7 8' is seen as a list, that is run over by 'name'. The distribution is somewhat uniform. How can we list files in a more intelligent way? We can modify the function 'cloud_ls'. Open 'vi', edit what must be edited, and in the end we should have

```
function cloud_ls() {
    tmpfile="/tmp/.cloud_ls.$RANDOM"
    for name in 1 2 3 4 5 6 7 8; do
            list $name >> $tmpfile
    done
    sort $tmpfile
    unlink $tmpfile
}
```

Now source;
```
source cloud.sh
```
and try the new function:

```
cloud_ls | wc -l
```

Notice how much time it takes;

```
time cloud_ls | wc -l
```

These are times for a very small system. Imagine scaling these times for million of entries... It may take entire minutes just for a simple listing.

We can instead use the concept of **buckets**: listing is restricted to smaller subsets, called buckets. A listing involves only a single bucket. Let us implement such a system. Download stuff as usual.

```
wget https://apeters.web.cern.ch/apeters/csc2018/_downloads/de5792e3bdec6634fbdbdb0b943e810d/clou
```

Cloudset gives us even more functions towards string manipulation. We can ‘embed’ cloudset in cloud.sh by adding the string ‘. cloudset.sh’ just after the initial comments. After it’s done, recompile using the usual ‘source’ command. Now, let us edit ‘cloud_upload()’ once again. Open ‘vi’ as usual, an in the end we should have

```
bucketname="$USER"
buckethash=‘sha1string $bucketname‘
buckethashkey=${buckethash:0:1}
bucketindex=‘h8d $buckethashkey‘

function cloud_upload() {
        #checking if the file exists
        if [ ! -f "$1" ] ; then
                echo "error: source file <$1> does not exist!";
                return 1;
        fi

        hash=‘sha1string $2‘;
        hashkey=‘echo ${hash:0:1}‘;
        hashvalue=‘h8d $hashkey‘;
        echo "==> Uploading $1 with hash $hash to DHT location $hashvalue"
        upload $1 $hashvalue $2
        set_add $bucketindex $bucketname $2
        #adding bucket information to the uploaded file’s name
}
```

Once again, recompile, and then compare the performance of ‘cloud_ls’ and ‘set_ls’:

```
time cloud_ls | wc -l
```

```
time set_ls $bucketindex $bucketname | wc -l | sort
```

Using a bucket system makes the matter significantly faster! 5 times improvement with a simple mod. Now, to kill a server: from root,

```
jobs
```

and if not root,

```
ps -elf | grep redis
```

This should return two jobs: one is the command grep itself, the other is the server. Then,

```
kill -9 <process id>
```

The process ID is the 3rd number. Insert the correct ID, and then verify that the server has been properly killed:

```
ps -elf | grep redis
```

The server should be dead.

2.3 SSO

This exercise is focused on certificates. Using a non-Chrome browser, go to the following link:

```
https://www.digicert.com/secure/saml/discovery/?entityID=https%3A%2F%2Fwww.digicert.com%2Fsso&ret
```

then

- select ‘Universita di Padova’;
- accept (even without trusting) and then go back. You will find a certificate in one of the previous pages;
- three files can be downloaded. One of them is actually our certificate (+ private key), the others are just files referring back to the organizations providing the certificate;
- install the certificate.

What is the purpose of this certificate? If a site requires a certificate and it trusts the authority that issues it, we can then access to that service. For now, however, we can not actually do much with this certificate. Then, let us start working on the Cloudveneto VM. We will need to copy the obtained certificate on our VM, so we will need some specific commands. Let us start by performing a remote copy of a dummy file on a gate (that will act as a home directory):

```
echo "Andrea Perin" > myfile.txt
```

which creates a file with my name on it; take a look at it by typing

```
cat myfile.txt
```

Now, let us use the ‘scp’ command to do a remote copy on the gate:

```
scp myfile.txt aperin@gate.cloudveneto.it: /
```

The same can be done with the certificate, and that is exactly what we are going to do. Now, let us log in our gate:

```
ssh -t -l 1238:localhost:1238 aperin@gate.cloudveneto.it
```

Now, a short digression about Kerberos. Kerberos used to issue challenges in the traditional way: Bob asks Alice to encrypt a message, if she passes the test then she is actually Alice. However, such system is faulty (it is weak to a reflection attack). Instead, now Kerberos uses a different method. Kerberos also has an authorization system. If we write in the terminal

```
ls -l
```

We get something like this:

drwxr-xr-x	5	perina	fisica	260	May	16	18:06	adv stat
drwxr-xr-x	2	perina	fisica	6	Mar	11	17:16	Desktop
drwxr-xr-x	2	perina	fisica	113	Apr	2	18:59	Documents
drwx——	7	perina	fisica	4096	May	23	08:56	Downloads
-rw-r–r–	1	perina	fisica	0	May	16	13:34	global.lock

Table 1:

Going by column:

- the first string of chars tells us:
  - whether the file is a directory (d) or a file (-);
  - the following 9 chars, instead, tell us the permissions: reading (r), writing (w) and executing (x). The three ‘rwx’ blocks refer respectively to user, group and other;
- the number (not explained);
- the user (in this case ‘perina’);
- the group to which the user belongs (in this case ‘fisica’);
- other info (size, date and so on).

We can use the command ‘chmod’ to change the permissions; for example, creating a dumb file and modifying its permissions:

```
   echo "huivhjhuvjjhbvb" > cose.txt
   ls -l
   chmod g+w cose.txt
   ls -l
```

and hopefully this should result in a change of permissions, as the group (g) was given writing (w) permission. Before:

```
   -rw-r--r-- 1 perina fisica 16 May 23 10:08 cose.txt
```

and after

```
   -rw-rw-r-- 1 perina fisica 16 May 23 10:08 cose.txt
```

In order to change more than one at a time, we can use the compact form of the command, which employs a numerical input.

```
   chmod 666 cose.txt
```

Use the following map to interpret it.

```
permission to:  user(u)   group(g)   other(o)
                /    \     /    \     /    \
octal:            6          6          6
binary:         1 1 0      1 1 0      1 1 0
what to permit: r w x      r w x      r w x

binary         - 1: enabled, 0: disabled

what to permit - r: read, w: write, x: execute

permission to  - user: the owner that create the file/folder
                 group: the users from group that owner is member
                 other: all other users
```

Now we are ready to actually work with the certificate. Enter in the VM as root:

```
   ssh -t -l 1234:localhost:1234 aperin@gate.cloudveneto.it
```

and put the certificate in the VM.

```
   scp perin_andrea_andrea_perin_10_studenti_unipd_it.crt andreaperin@10.67.22.14: /
```

From there,

```
   ssh root@10.67.22.14
```

to enter as root. From there, get to the folder

```
   cd /home/andreaperin
```

and get the files from the github repository;

```
   git clone https://github.com/marcoverl/training.git
```

and then pretend to be a specific user by using the command ‘sudo’ (switch username do).

```
   sudo -u andreaperin /bin/bash
```

we are effectively becoming ‘andreaperin’, and by running ‘/bin/bash/’, a new terminal is opened. In case of identity crisis, fear not and type

```
   whoami
```

to receive the latest updates about who you are. **IMPORTANT**: once we are no more root (e.g. we have become andreaperin), we may no more be able to do some things. Now,

```
   exit
```

to return to root mode. Access the folder that was downloaded from git by typing

```
   cd training/Grid
```

Make everything executable by typing

```
source setup-cvmfs.sh
```
we can not simply execute it; permission denied. This is because we do not have the authorizations needed.

IMPORTANT: source takes the file and runs each of its lines in the terminal, without opening some separate process in which to run everything. In order to make this file executable, use chmod.

What is cvmfs? CernVM File System (CernVM-FS) is a network file system based on HTTP and optimized to deliver experiment software in a fast, scalable, and reliable way. Let us make things runnable by using chmod;
```
chmod 777 setup-cvmfs.sh
```
and then running the file we just made runnable by typing
```
./setup-cvmfs.sh
```
Repeat this for all other files:
```
chmod 777 setup-euindia-vo.sh
./setup-euindia-vo.sh
chmod 777 setup-x509.sh
./setup-x509.sh
```
And start another terminal,
```
sudo -u andreaperin /bin/bash
cd
```
This teleports us to ‘andreaperin’ home, not root’s home.
```
scp pippo@10.67.22.26:/tmp/userkey.pem .
scp pippo@10.67.22.26:/tmp/usercert.pem .
```
this command remotely copies stuff from ‘pippo’ (a portmanteau of prof. Campana in that particular session). Lok now at all files, also hidden ones, by typing
```
ls -al
```
Find the directory ‘.globus’, and copy the files ‘pippo’ gave us inside it.
```
cp usercert.pem .globus
cp userkey.pem .globus
```
go inside it,
```
cd .globus/
```
take a look at all files
```
ls -al
```
Set all permissions in the appropriate way:
```
chmod 400 usercert.pem
chmod 400 userkey.pem
```
Back to home,
```
cd
```
and move to the ‘Grid’ folder:
```
cd training/Grid
```
Now do some exporting of environment variables:
```
export VOMS_USERCONF=$PWD/etc/grid-security/vomses X509_VOMS_DIR=$PWD/etc/grid-security/vomsdir
export LCG_GFAL_INFOSYS=lcg-bdii.cern.ch:2170
```
and then type
```
voms-proxy-init -voms euindia
```
but credentials could not be loaded. No adequate permissions!

3 Lucchesi's part

3.1 Distributed computing: an introduction

There is not a real review about this subject, as it is a still growing area where developments are continually achieved. It is important to, at least, identify what 'distributed computing' actually means, why it is such an important topic, and also understand where the research about it is headed towards.

The origins of distributed computing stem from the need of more computing power, in order to face the challenges of an ever-increasing amount of data to process. The idea of distributed computing dates back to 1961, when computing pioneer John McCarthy said that **'computation may someday be organized as a public utility'**. It is important to note that, back in those days, computation was sort of an umbrella term under which both storage and computation 'per se' were denoted.

This is actually where the term 'GRID' comes from; ethimologically, grid usually refers to the power network. The natural implication is that **computation** is just as electricity: **an on-demand service**, which users can access according to their needs (obviously, also paying). Ian Foster was one of the people that believed in standardizing the protocols used to request computing power, in order to approach the power grid model. This means that, in the same way that the power grid follows some common rules (voltages and wattages are standardized), **a computing grid should be subject to a set of common rules** to allow a widespread use. Some early examples of a computing grid are the Worldwide LHC Computing Grid (WLCG), and the European Grid Infrastructure (EGI).

3.2 The GRID

Historically, in the 60s, researchers used to travel to the sites of the various experiments they were involved in, in order to both gather data and analyze it; most of the times, this meant CERN (which was already amongst the largest sites in the world). Then, as the years went on, some form of delocalization began to take place. Other smaller facilities were given copies of the data, in the form of tapes (which are the best tool for long term storage, the reason being their low price and high reliability), as data of that magnitude (Petabytes) could not travel through the internet.

GRID started in the mid 90s, as a way to face large scale computation problems using a network of resource-sharing machines, so to achieve a computational power which was achievable only using dedicated supercomputers or large dedicated clusters. This solution (a large number of smaller machines, connected through a global network) was chosen for its cheapness and because it was realizable, whereas accessing supercomputers was both hard and expensive. Furthermore, a wide array of geographically distributed resources guarantees a better reliability, while being more heterogeneous and dynamic. It is also due to this heterogeneity that GRID is focused on integrating existing (and possibly very diverse!) resources with their hardware, operating systems, local resource management, and security infrastructure.

In this regard, GRID is actually composed of an ensemble of small farms, connected through the internet; of course, in such a system, there is the need of a uniform communication protocol (there is no pre-existing "common language"). GRID then groups these 'local farms' under an 'umbrella portal'. In order to access some data, we need knowledge of the **metadata** pertaining to it. This acts like a pointer, telling us where the data we want is stored; only the owner of the computer itself knows the actual location. This system, based on distribution, makes so that jobs, too, need to be distributed. GRID faces this task, choosing where to deliver the various jobs; the user does not know where his data/computing is.

Regarding the data/computing dichotomy, there are two main computing paradigms:

- **High throughput computing**: computing that streamlines access and writing, used when there is lots of data;

- **High performance computing (HPC)**: small amounts of data, but high amount of computing operations.

When we are dealing with big data, the HPC approach can not be realistically used: HPC is not (yet) high-throughput. There is no way of having direct accesses of parallel HPC to data, so that all the computation power is wasted (it only acts on small chunks of data). Possible ways to improve this would be

- higher memories for GPUs;
- toroidal architectures.

Since HPC is not suitable to big data processing, the switch was made to a distributed system (like GRID): a large number of smaller sites which, while being less performant than a HPC center, are better at facing the challenges of big data.

Initially, all available resources were employed, regardless of their features (like the manifacturer). Of course, this aforementioned **heterogeneous computing** lead to great complications in the management of the system.

3.2.1 Security: the VO system

GRID (but also the Cloud) is built around the **Virtual Organization (VO)**, a a logical entity that refers to a **dynamic set of individuals or institutions**, for which Grids define and provide a set of standard protocols, middleware, toolkits, and services built on top of these protocols.

VOs are used (for example) in order to decide whether a particular individual can submit a job; if he/she belongs to the right VO, then permission can be granted: on the basis of the VOs, **authentication and authorization** are performed. For each VO there is an administrator, which is held accountable for the VO itself. Some actual examples of VO are:

- each LHC experiment;
- cosmology theoreticians;
- CNR;
- INFN;
- even part of the public administration!

A VO is a very serious concept. It can be thought of as **virtual identity card**; this requires an authorization for each VO, and this authorization is given by a specific authority (and there is one of these authorities for each nation). Individual users can be banned/eliminated from VOs. Initially, Cloud had not an authorization/authentication system.

3.2.2 Interoperability

GRID resources may come from different administrative domains, which have both global and local resource usage policies, different hardware and software configurations and platforms, and vary in availability and capacity. This also means that these resources can be added/subtracted at a moment's notice.

GRID is a **midware structure**: it is a set of code that works on several layers and help translating all these possible differences in order to make everything work properly. Actually, the GRID protocol architecture can be decomposed into **5 different layers**:

- Fabric;

- Connectivity;
- Resource;
- Collective;
- Application

3.2.3 Fabric

By fabric, we denote the set of hardware resources (both computing and storage), networks, code repositories (that is, all the needed code libraries) GRID is based upon. GRID assumes the existence of these fabric components, such as local resource managers like Condor and BPS (systems which can manage resources and communicate which component is free at any time and so on), and disk/tape managers.

Related to this layer, jobs also have descriptors, and it may be that their results need to be stored in some specific place. If no place can be used as a storage for these jobs, then they can be sent back (this, however, depends on the VO).

3.2.4 Connectivity

A software layer (even if it is much more than just 'a software') with communication/authentication protocols, which allow easy and secure network transactions. This can be the set of internal farm protocols (specific to the particular site) + the external security protocols. The connectivity layer 'can decide' which VOs to ban. The connectivity layer is also related to the Grid Security Interface (GSI), which underlies every GRID transaction (for example, maybe only one type of job can be run, or a job that can be run can not be stored).

3.2.5 Resource

Still, a software/protocols related layer: it defines protocols for the publication, discovery, negotiation, monitoring, accounting and payment of sharing operations on individual resources. More specifically:

- publication of data: where should it be stored?
- access to data: where is it?
- negotiation: which job has to be executed first? Priority (obviously, this is linked to the VOs);
- monitoring: who uses the resources, when, whether there are any errors... Monitoring takes place on two different levels, local (of the specific farm) and/or global;
- accounting: keeping track of transactions, and how much each user uses the resources. This is related to money: the more you use, the more you ought to spend. GRID make so that computation really is like electricity.

3.2.6 Collective

The collective layer captures interactions across collections of resources. It deals with the directory brokering services, scheduling services, data replications services, and diagnostics/monitoring services. These services are not associated with any one specific resource but focus on interactions across resources. The programming models and tools define and invoke the collective layer functions. This layer is a key component in the whole grid architecture and its functioning. This is the layer that glues all the resources together in expedient exchange.

3.2.7 Application

This denotes anything a user may want to execute. It comprises whatever user applications built on top of the above protocols and APIs and operate in VO environments. In the years several application have been developed. It is the software through which one actually accesses the GRID.

3.3 Components of the GRID environment

Components are related to atomic tasks for building a GRID software, referred to as middleware. The most important three are

- Resource managers;
- Data model;
- Security and monitoring.

3.3.1 Resource managers

The name is pretty self-explanatory. The role of this component is to act as a scheduler, which is responsible to find what resources are available at any particular moment, which user called, who to assign them to.

Since the fundamental feature of Grid is the coordination and sharing of resources located at various places in decentralized ownership, the architecture may be classified in a way these resources being contributed, consumed and shared. The Grid architecture can be classified in two ways: **push** vs. **pull model**.

- **Push model**: pools of resources are geographically distributed and are heterogeneous. They are known by the scheduler, which can *push* the various jobs to the computational resources available from the pools. This model implies that all knowledge about the resources is inside the scheduler itself. In the case of huge datasets, this model is not scalable. Furthermore, the database of current jobs is not able to take a snapshot of the state of the various components;
- **Pull model**: tasks are retrieved, *pulled from* the 'central pool' of submitted jobs, which are VO dependent. This system uses **pilot jobs**: these are particular jobs, sent to all sites by the VO; they 'talk to' the heads of the sites, If there is 'enough space', the pilot then communicates with the resource manager, effectively pulling the job when the site is ready. This system is VO based: the VO manager uses a scheduler which only refers to his jobs. The push model, instead, requires a global realtime database of all jobs (which is also why it does not scale).

Each pool of resources has a **head node**, responsible for the scheduling and dispatching of jobs to the locally connected nodes: in a sense, the head node manages the contact with the outside world (head nodes are the ones which communicate with pilot jobs in the pull model).

Pull and push can also be referred when talking about output deployment (as output can be pulled/pushed); the output, however, can also be simply stored locally.

3.3.2 Data model

By 'data model' we refer to the set of software that has to deal with questions like where is the data stored, how to access it, how to preserve it...

Data is increasing more and more in recent years; data used to be stored locally, its movement was kept at a minimum, and it was located in a relatively small place. In GRID, data management follows some of these same main ideas. An important concept is **data locality**: this allows to reduce the transfer of data to a minimum, while also improving the performances of applications and the overall scalability. In this framework, **jobs go to the data**, meaning that jobs are scheduled close to the data they refer to. This makes data-aware schedulers critical in achieving good scalability and performance.

In general, data management architecures need to be aware of the structure of data, and also need to take into account CPU, disk amount, network etc. This means that each particular experiment needs its own data model (HEP is different from biophysics!).

On the same note, attention must be directed towards the physical side of storage, i.e., whether we use disks, tapes or something else altogether, as this affects not only performance but also the efficiency of moving that data:

- **Tapes**: these are stored in dedicated buildings. The service is (nowadays) automated; still, if one job requires access to the data stored in a tape, said tape must be physically retrieved, then read (sequentially, which may mean additional time consumption) and cached (so that some disk is needed also when dealing with tapes; there are dedicated disks that are used specifically to serve as cache for tapes). Only after all of this is done can data be sent to the entity requiring it;

- **Disks**: these are faster in both writing and accessing (since they are not necessarily operating in a sequential fashion), and are also easier to support from a 'human' point of view. Disks can be divided into **hot** and **cold**, depending on how much access requests they face (and also their speed). This means that 'cold data' is usually stored on cheaper, less performant disks, as they are accessed less frequently (hours may pass between successive accesses). 'Hot disks' are instead dedicated to data which has high access frequency, and sees no waiting time between requests.

3.3.3 Security, authentication and authorization

One of the most challenging task on Grid was to to develop a secure system: this means that only properly authenticated and authorized users and resources participate and collaborate in the Grid environment.

- **Authentication**: a process through which each physical person (or institution) is distinguished. Authentication is achieved through the usage of digital certificates, released by Certification Authorities (CAs). Digital certificates are a double key system (they employ asymmetric encryption: a set of public+private key);

- **Authorization**: this determines what each person/entity is allowed to do. Inside each VO, people cover various roles and thus have specific roles and rights. This system is what prevents a simple student from requiring and obtaining instant access to (say) 300 PB, as this is usually moderated and higher levels of authorization are required.

3.4 Cloud

The Cloud is the commercial answer to the GRID. While it shares some of its features (and its resources: the data centers used by GRID and Cloud are the same), it aims at solving some of its underlying problems. Clouds are usually referred to as a large pool of computing and/or storage resources, which can be accessed via standard protocols via an abstract interface. There are multiple versions of definition for the Cloud architecture. In the following, a 4-layer version is defined, in comparison to the GRIS's 5. These are:

- Fabric;

- Unified resource layer;

- Platform;
- Application.

Fabric has the same meaning/scope of the fabric layer found in GRID. The hardware components are pretty much the same (computing. storage and network resources). **Unified resource layer** contains those resources which have been abstracted enough that they are available for use from the higher level applications. Usually, some level of virtualization is involved; the end result is a set of integrated resources, for instance, a virtual computer/cluster, a logical file system, a database system, etc. **Platform layer** denotes where the middleware is; it adds a collection of specialized tools, middleware and service on top of the unified resource layer that provide a development/deployment platform. For instance, a Web hosting environment, a scheduling service, etc. **Application layer** contains the usual set of software which actually runs in the Clouds.

3.4.1 The Cloud Services

Clouds in general provide services at three different levels: IaaS, PaaS, and SaaS.

- **Infrastructure as a Service (IaaS)**: provides hardware, software and equipment (usually at the unified resource layer level, but also at the fabric level) to deliver software application environment. The infrastructure can dynamically scale up or down depending on the specific needs. Examples: Amazon EC2 (Elastic Cloud Computing) Service and S3 (Simple Storage Service);
- **Platform as a Service (PaaS)**: offers a high level integrated environment to build, test and deploy custom applications. Generally, developers will need to accept some restrictions on the type of software they can write in exchange for built-in application scalability. An example is Google's App Engine, which enables users to build Web applications on the same scalable systems that power Google applications;
- **Software as a Service (SaaS)**: delivers dedicated, high level services with specific aims, remotely accessible by consumers through the internet.

3.4.2 Recent evolution and future ideas

The GRID is progressively moving to the Cloud. The most dramatic (and impacting) development in recent years is the improvement in network technology. This makes it possible to transfer high volumes of data in relatively short amounts of time. As a consequence, data locality (part of the GRID paradigm) is no more an absolute requirement like it was in the past. Jobs no longer need to go to the data; there may be a job running in Padova reading data from Bari. As a consequence, data can be kept in **datalakes**, which are like larger data pools (and hence the name): a set of few large repositories where structured, unstructured and in-between data can be stored. Datalakes and datalake analytics are already being used by commercial companies.

Among future ideas:

- high-speed networks and large data centers;
- computing centers as cloud (also including HPC centers and commercial companies);
- users connect through users cloud.

HPC in particular is the subject of a lot of interest, as it is the least-integrated part in this whole GRID-Cloud story. The mission of PRACE (Partnership for Advanced Computing in Europe) is exactly that of enabling high-impact scientific discovery and engineering research and development across all disciplines to enhance European competitiveness for the benefit of society. PRACE seeks to realise this mission by offering world class computing and data management resources and services through a peer review process.