Unit 1 – Introduction

Student Guide

Goals

- To understand that every system that uses an interconnection network is a distributed system.
- To identify what a distributed system is, why they are relevant and which are their main applications.
- To know several examples of distributed systems.
- To study the evolution of scalable distributed systems, identifying cloud computing as the current stage in that evolution.

Contents

1. Distributed Systems

A distributed system [1] [2] is a set of autonomous agents that interact in order to achieve a common goal.

- Each agent is a sequential process with its own independent state that is executed at its own pace. The interaction with the remaining agents can be done exchanging messages or using shared memory.
- That common cooperation goal may be used in order to evaluate the global behaviour of that system.

In the end, a distributed system is a networked system.

2. Relevance

The study of properties in concurrent systems started because of the need to understand how multiple parallel activities were coordinated when a set of shared resources (mainly, memory) was being accessed. This provided a robust basis for designing and implementing the operating systems that manage how multiple resources are shared among concurrent activities.

Distributed
- Independent machines working towards the same objective

Distributed systems are a particular case of concurrent systems. Their main differencing characteristic is that each one of the elements in a distributed system is executed in an autonomous way and has its own set of private resources (including main memory), compelling it to cooperate with other elements using a communication network.

Concurrent
- Various computations progress & interact at the same time

In the beginning, the first theoretical studies about distributed systems needed some kind of justification about why this kind of systems could be considered interesting since, apparently, they introduced a lot of problems and very few solutions.

Such justification was fourfold (and it is still valid):

- Distributed systems increase the functionality (usage scenarios) of computers, since they allow the cooperation among nodes that were autonomous.
- They increase resource sharing and, due to this, resource usage. Many resources that are available in a computer (e.g., printer, external disks...) may be also accessed from other computers, thus increasing resource usage and profitability.
- They provide an easy way (perhaps, the unique one) of increasing performance, since there are physical bounds on the computing power that a single computer is able to provide (processor frequency, amount of main memory, number of computing cores ...). To this end, each complex activity (or problem) is divided into many simpler activities that may be assigned to different computers.
- They improve system reliability. Distributed systems spread the information to be processed and the computing power needed to process it among multiple nodes. Thus, in a properly designed distributed system, when some failures arise (in one or a few nodes) most of the data and the computing power should be still available and able to continue.

Things started to change in the 80s, with the development of multiple kinds of local area networks and the usage of a client/server interaction model. Programmers and users realized that distributed systems were not so complex.

In the 90s, small sets of computers were used in order to improve the reliability of the distributed services being provided. Those sets were named "*highly available clusters*". With them that third theoretical advantage of distributed systems (listed above) was finally implemented in practice. As a result, this increased the perceived usability of distributed systems.

From the set of local area networks that started to grow in the 80s, Ethernet (and its variants) become the most widely adopted technology. On top of this physical layer, the Internet Protocol (IP) was also the clear winner among the existing network protocol proposals.

The growth of the world-wide web in the 90s popularised the adoption of the IP protocol. This highly increased the growth (and functionality) of distributed systems; i.e., systems based on sets of autonomous computers interconnected by a network.

Nowadays, we may say that all those reasons remain valid since the current computing environment IS distributed and interconnected, with a miriad of connected "computers" that interact providing (or using) many remote services that may be accessed as shared resources, as the World Wide Web.

Among the existing challenges in this scope, we may outline two of them:

1. How to use such existing connectivity in order to obtain useful results. The development of new services using these resources demands a thorough reengineering, since traditional techniques are useless now. These services will be potentially usable by millions of users. Therefore, this new scale breaks the traditional interpretation of the computing problems and their solutions. So, we will start with no reference to any valid solution.
    - Let us design a distributed version of the *sieve of Eratosthenes* in order to compute prime numbers where 1000 computers collaborate in that computation. We find problems in different parts of such task: algorithm redesign (how to divide the algorithm tasks? how to mix the results from each computer?), balance between communication and computation (a message for each subtask? a message for each partial result?), dynamic adaptation to changes in the computer set (how to manage computer failures? how to accept new computers once the computation has started?). Finally, once all these difficulties have been solved... have we achieved better performance than in the traditional version of that algorithm?
2. How to create distributed subsystems that provide dependable services. The new technologies provide solutions for new problems and to reach scales that couldn't be achieved in the past. In this scope, new challenges arise:
    - How can Google implement its search service?
    - How can Dropbox manage the shared use of files by millions of users?
    - How the simulation of new medicines against cancer can be distributed among millions of volunteers?

3. Application Areas

Some of the main application areas in distributed systems are:

1. World Wide Web
2. Sensor networks → Ejemplo del sensor del profesor
3. Internet of Things (IoT)
4. Cooperative computing
5. Highly available clusters

Let us describe each of them in the following sections.

3.1. World Wide Web

It is based on the client/server model. The server waits for document requests, while the clients are the web browsers. They send and receive documents.

- Requests to servers imply an access to a document (read or write action).
- Browsers analyse hypertext documents looking for metadata; e.g., links to other documents. Those linked documents may be placed in the same or in other servers.

This is a simple and powerful paradigm, initially designed for sharing documents, but extended for allowing that those document requests become service requests, returning "documents" that provide the result to those service requests.

3.2. Sensor Networks

These sensors are specific purpose mini-computers (a.k.a. "motes"). They are embedded in daily use devices (e.g., electrical appliances) and usually consist of sensors (humidity, temperature...) and actuators (at least, a basic reporting system). These capabilities allow a wide range of potential applications as surveillance, disaster detection (chemical, biological...), power consumption monitoring and many others.

3.3. The Internet of Things

Nowadays all computer systems are conceived assuming that they will use a communication network. Indeed, it is generally assumed that such a network is Internet; i.e., a global system of interconnecting computer networks with a world-wide scope.

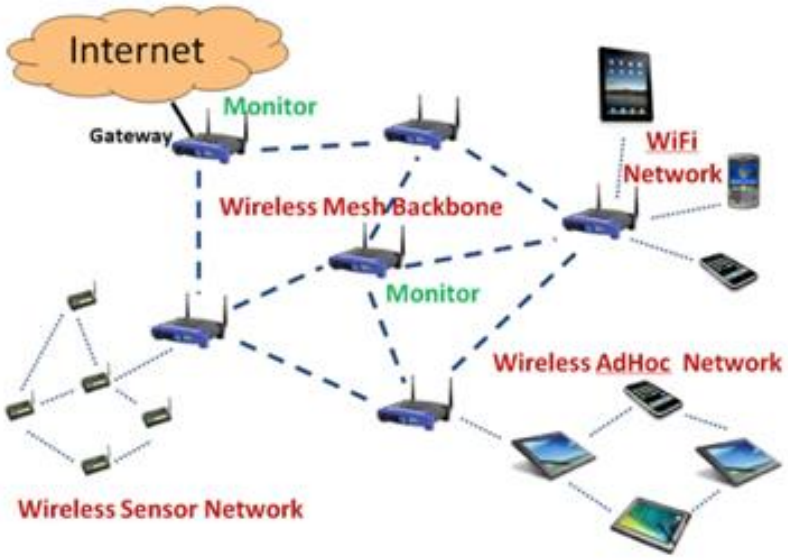
Such dependency on computer networks is so strong that there is a new concept naming this feature: the "Internet of Things" (IoT). This means that every computing device should be connected to Internet. In this way, devices will be able to provide and request information services, acting on their own physical environment or on the environment of other connected devices.

IoT may be seen as a generalisation of sensor networks where all computing devices may interact among them and act on their physical environment. This opens new scenarios:

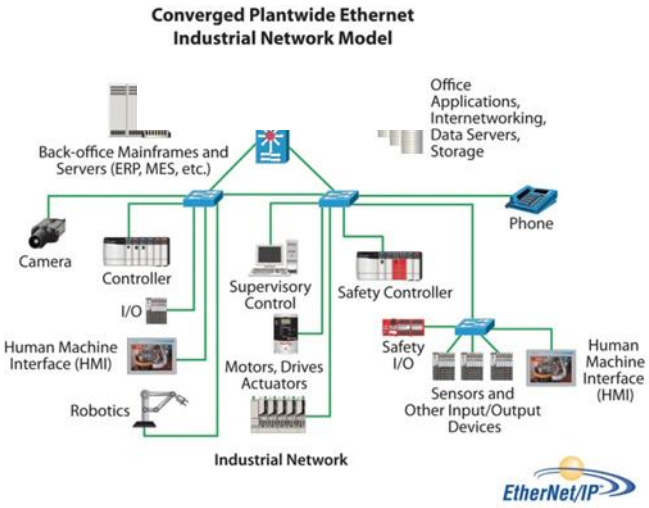
- Smart cities

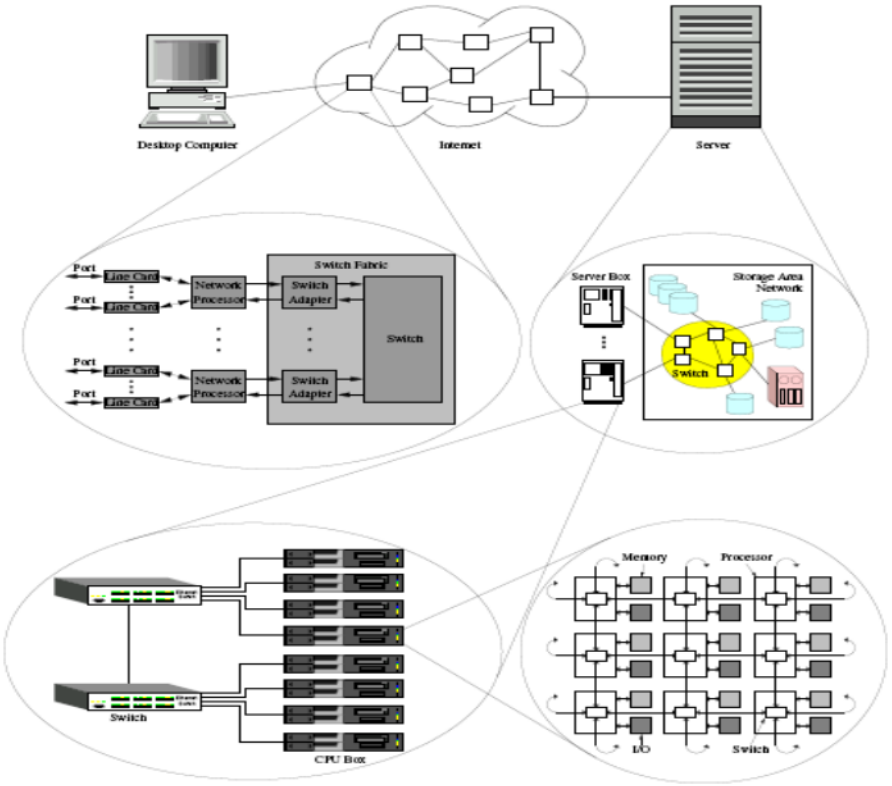
- Automation of many processes (building, manufacturing...)
- Automated medical advice

In this scope, distributed systems improve the connectivity and interoperability of all the involved devices.



This concept (*Internet of Things*) tries to capture the large functional possibilities arising when every device could cooperate with any other connected computing device.





3.4. Cooperative Computing

Many computational resources, mainly personal computers, are underutilized. On the other hand, many scientific and engineering problems may be subdivided in smaller pieces (tasks):

- Each task may be solved in a brief interval. The results of all tasks can be combined in order to build a complete solution to any of those problems.
- The server creates the set of tasks for a given problem and its clients obtain some of those tasks.

Personal computers connected to Internet may be subscribed to those servers, behaving as "volunteers" that receive tasks to be solved:

- They install a special client: a "runtime" for executing tasks.
- The client contacts the server and interleaves off-line processing stages with other communication stages where new tasks and previous results are exchanged between client and server.
- The server spreads tasks among clients and picks up their results.

3.5. High Availability Clusters

Development of the *Internet of Things* was based on an obvious interest on the functionality arising when multiple *smart* computing devices were interconnected being, in some cases, able to interact with the physical environment.

Computer clusters were previously proposed with the aim of improving system reliability. To this end, clusters supported computer and network failures without corrupting data or stopping regular data processing.

Every device may fail. Computers are no exception in this regard. Indeed, a computer may have different kinds of failure: in its hardware and in its software. These failure kinds and failure scenarios will be carefully studied in Unit 5.

One of the aims of the first failure management strategies was the assurance of data consistency. Several types of transactional managers ensured that the data to be persisted were not corrupted when failures broke the activities that were updating such data. In the worst case, these managers could detect such corruption avoiding that incorrect (i.e., inconsistent) data were propagated to other nodes or agents.

Multiple computers were used in several settings in order to guarantee data integrity, but also to ensure that the resulting system was not stopped when any of its computers failed. Redundancy is the solution. When every computing element is replicated, the failure of one of these components is overcome by its replica. That replica replaces the faulty node, receiving and processing all requests initially forwarded to the crashed node.

This replication strategy is the base for developing highly-available clusters. These clusters have been used for managing large critical enterprise applications, where availability and data integrity are essential requirements.

Highly-available clusters try to comply with those integrity and availability requirements. However, the techniques being used in their implementation prevent the resulting systems from scaling.

A system is considered *scalable* when it is developed, configured and deployed allowing an apparently unlimited workload. Some scalability approaches will be carefully studied in Unit 4.

Common replication and consistency protocols being used in highly-available clusters inherently set an upper bound on the amount of computers that compose them. Once this bound is reached, adding other computers to the cluster penalises its performance: the protocols use more time coordinating them than effectively using them.

As a result, the unique approach for scaling a highly-available cluster consists in increasing the computing power, secondary storage and main memory of each one of the computers that is in the cluster. With the current limits in computer design, clusters will be either unable to adapt to high workloads or impose an excessive cost in that adaptation (since it would require a frequent renewal of a set of high-end, i.e. expensive, computers).

4. Cloud Computing

There have been different sets of requirements in the construction of computing systems that have converged in what is currently known as "*cloud computing*" (CC) [3].

The *Internet of Things* maximises the set of scenarios that could get value from the interaction among multiple computing devices. On the other hand, high-availability clusters increase the dependability (i.e., reliability, availability, maintainability, safety and security) of distributed systems. Unfortunately, none of these two approaches is centred on building and managing efficient distributed services nor on guaranteeing an easy way to scale.

Let us follow the evolution of the view of software as a service and how this has generated the concept of *cloud computing* as it is currently understood.

4.1. CC: Programs and Services

The general goal of cloud computing is to "convert the tasks of creating and exploiting software services as something simpler and easier". In the end, programs have always been written with the aim of obtaining some kind of service using computers.

The evolution of the computing industry has partially hidden this fact, especially because personal computers have spread a specific way of interaction between users and computers.

4.2. Roles in the life cycle of a SaaS

Software has been always developed with the aim of providing some service. The current meaning of "*software as a service*" (SaaS) [3] denotes a fact that always has existed.

In the software life cycle several roles may be distinguished:

- Developer. He or she implements the components of computer applications.
- System administrator. The person that places each piece of software and hardware in its correct place, configuring appropriately the entire system.
- Service provider. The role that decides the service characteristics, the components that build it and how it should be configured and administered.
- User. The role that accesses to and uses such service.

4.3. Software Services Evolution

Let us describe in this section how software services have evolved since their origin.

4.3.1. Mainframes

In the first computer systems, the mainframes were managed by specialised operators. They ensured that the system was being executed without problems. To this end, they supervised most of the tasks, even when a job (i.e., an application) could be loaded in order to be executed.

Since these computers were used by only a few operators, there were no problems due to excessive demand or excessive workload.

Most system users need to write their own programs in order to use them. Thus, the roles of user and developer were mixed. Even the role of service provided was also mixed with those two.

- Since roles were not distinguished, users were implied in too many management aspects for the services they planned to use.

On the other hand, hardware was used efficiently in this stage, since computers were used by multiple users, reducing the overall cost per user.

4.3.2. Personal Computers

Later on, with the advances in computer hardware and with better operating systems, the computing model evolved towards personal computing. In this stage, software is built by specialised companies that oriented their developments to a large set of potential users. These

users should install this software in their personal computers or workstations in order to be able to execute it.

This model promotes the idea that the user (i.e., the owner) of the personal computer may manage this system with freedom, installing a potentially large set of software in it.

On the other hand, this view also forces the user to administer that system. This implies that such user should solve any hardware issues and also should ensure that all installed software will always run without problems.

As a result, the roles of user and system administrator are mixed in this stage (and this still is the prevalent view nowadays).

4.3.3. Highly-available Clusters (Datacentres)

Highly-available clusters do not break these assumptions. These clusters are regularly set in the datacentres of their user companies. Those companies need specialised personnel that manage and administer both the hardware and software of these systems. This user company is thus both the system administrator and service provider in this kind of systems.

Depending on the size and needs of that company, sometimes we may find specialised internal developers for that company.

In some cases, we may also find a variant that maintains all software for a given company in external datacentres, minimising economical costs. This approach...

- ...reduces the hardware purchase costs.
- ...minimises the administration- and maintenance-related costs for that company.
- ...avoids fixed electricity costs.
- ...facilitates the management of computer-related costs.

4.3.4. Software as a Service (SaaS)

However, in the last years computer networks have evolved, providing a larger bandwidth for most users (either at enterprises or at home). This allows that final users reduce their tasks as administrators to a minimum. The aim is that users do not administer their PCs in order to access to software services and systems. Instead, those services should be provided through Internet and they will be available to a larger set of computing devices.

In this new scenario, there is an explicit service provider (i.e., SaaS provider), who should manage software deployment on as much servers as needed, depending on the set of service users.

The problems to be managed by a service provider are the following:

- To guarantee the quality of service (QoS), according to an implicit (or explicit, via a Service Level Agreement, SLA) compromise with its users.
- To choose the appropriate dependent software components in order to deploy the service, complying with the guaranteed QoS levels.
- To react against service demand variations, minimising the amount (and cost) of used resources.
- To react against deployment issues, ensuring service continuity.

User = Client

Syst admin
Service Provider
Developer
} SaaS

- To ensure service liveness while components are being upgraded through all service life cycle phases.

In a personal computing model each of these tasks should be executed by the user (who is also the service provider and system administrator). In a SaaS model, those are not user tasks but provider tasks.

Which are the advantages of this SaaS model?

They are obvious for the user: she is now centred on her main role, avoiding other uninteresting tasks (service installation, administration, configuration...) that did not belong to her areas of expertise. This allows the potential participation of a larger number of service users, since now those users do not perceive any adoption barriers. Additionally these provider-related tasks can be now based on efficient and standardised procedures that can be easily automated.

It also permits a minimisation of the service costs, since the specialised provider is able to reach a greater number of users. Additionally, as multiple companies may enter this service providing market, their competence will generate lower prices for the final user.

There are advantages for the service provider, too. It is able to use a single infrastructure and a well-known set of procedures to provide service to a large set of users. This allows a fast amortisation of the investments. The market model is better, too. Instead of a model based on one-time licence sales (as in the PC era), the new one is subscription-based, with periodical (usually monthly) earns, as it happens with electrical or telecommunications providers.

It is worth noting that a provider may reach these economies of scale if it is able to rationalise its own costs. To this end, the provider should use at any moment an adequate amount of resources, carefully distributing them among all its users. Strategies based on an *a priori* assignment of resources to users are highly discouraged. This assignment should be dynamic and based on the actual demands.

For instance, if a provider needs 50 computers, it should pay for only those 50 computers. Later on, if the user demands change and the workload would be served by 25 computers, the provider should only use (and pay for) 25 computers. Additionally, those computers should be used for providing several services to the set of users, distributing their costs among them.

A dynamic environment of this kind raises a new problem: the need to adapt the provided services to the existing demands. To this end, services should be built allowing their fast reconfiguration and reaction.

A service is *elastic* when it is able to adapt its resource usage to the current demands, maintaining a given QoS. In cloud computing, an elastic service should be both scalable and adaptive.

4.3.5. Infrastructure as a Service (IaaS)

A concrete SaaS provider has a limited capacity for sharing its resources. Those resources may only be shared by the users of that system.

The SaaS service model leads a provider to collect a series of resources with sufficient capacity to manage the intended set of system users. In most cases, that set of resources is exclusively managed by the SaaS provider. However, a better scenario could consist in an underlying elastic infrastructure; i.e., the SaaS provider requests to an external infrastructure provider the computers and storage needed at each time depending on the users demands.

These are the properties that should be guaranteed in order to ensure the elasticity of a SaaS provider:

1. There should be an infrastructure provider able to charge it for the exact usage of the computing resources. → Pay-as-you-go
2. The SaaS software should be able to adapt itself to the kind of resources that the infrastructure provider supplies.
3. The infrastructure provider should be able to measure the amount of resources needed by the SaaS software at each time, automating its reconfiguration when needed.

Recently, some companies have provided flexible infrastructure (computers and storage) services, generating what is now known as the *Infrastructure as a Service* (IaaS) [3] model. An IaaS is an elastic service whose exclusive target is the provision of computing, communication and storage resources at some given unitary prices.

Thus, the original SaaS service model is now filtered towards the bottom architectural layers (the infrastructure), converting the SaaS provider in an IaaS user.

The IaaS provider becomes responsible for maintaining the infrastructure, offering to the SaaS provider the possibility to programmatically contracting an arbitrary and dynamic amount of computers, storage and communication bandwidth. This has been possible with the help of hardware virtualisation technologies; with them...

- The assignment of (virtual) hardware resources is simple and fast.
- The capacity of those hardware resources may be easily configured.
- It is easy to install a system image onto a virtual machine.

A SaaS provider may use the services of the IaaS providers for adapting the amount of computers that execute the SaaS software, according to the current workload. This allows, *a priori*, the adjustment of the costs incurred by the SaaS provider for satisfying the QoS compromised with its users. Thus, the SaaS provider may forecast with precision its costs depending on the exact demands, reducing its fixed costs and minimising the costs perceived by the final users.

Horizontal scaling
↑↑ nº of computers

Vertical Scaling
↑ components of a computer

According to the point 2 mentioned above, the SaaS software should be able to scale in machine units, instead of in machine size (i.e., improving some of the computer components: CPU frequency, CPU cores, memory size ...). This new kind of scaling, based on increasing the number of computers, is known as *horizontal scaling* [4] (or *scale out*), as opposed to the traditional one based on improving the components of the single machine that provided the service, that was known as *vertical scaling* [4] (or *scale up*). It is worth noting that *vertical scaling* is limited by the current available technology and is commonly more expensive than *horizontal scaling*.

The IaaS provider offers a simple service (at least, simpler than those offered by SaaS providers). So, it is easy to assume that an IaaS provider may have a large number of users (SaaS providers) that will rapidly amortise the investments made in its datacentres and the software required to automatically manage its resources.

4.3.6. SaaS over IaaS

IaaS introduces a "pay per use" model that is one of the central characteristics of cloud computing, providing this same model to users of SaaS systems. Additionally, IaaS makes easy the creation of SaaS systems adapted to the workload introduced by their users, ensuring their own elasticity.

In order to be profitable, SaaS providers are compelled to an efficient use of resources and, in most cases, those resources raise variable costs. Therefore, the "pay per use" model from IaaS systems is clearly convenient for SaaS providers.

IaaS providers take the risks involved in purchasing hardware, assuming that there will be a large set of SaaS providers who, on their own, will serve a large number of SaaS users, generating a large demand of virtualised hardware.

In this scenario, a SaaS provider still plays several roles, besides its own one (service provider):

- It should manage hardware resource assignment.
- It should manage the system images to be installed, their upgrades and the set of programs to be deployed on those systems.
- It should set its own service management strategy, developing the intended monitoring and upgrading mechanisms.

System admin

4.3.7. Platform as a Service (PaaS)

The usage of an IaaS and of the software that manages horizontal scaling cover the items 1 and 2 of the requirements cited for building an elastic service.

However, in order to comply with item 3, the SaaS provider still has to solve the following problems:

1. Determine at each time the resources needed to comply with a given QoS.
2. Modify the deployment configuration of the SaaS software ensuring service continuity and its compromised QoS.

Ideally, we need a software broker that, consulting the current structural data for a SaaS service and revising the historical metrics about that service execution, could make decisions on elasticity automatically.

This kind of elasticity broker is known as *Platform as a Service* (PaaS) [3].

When a SaaS is integrated for running over a PaaS, this SaaS should comply with a series of additional requirements. Particularly, this SaaS should be specified with additional information that could be taken by the PaaS to make scaling decisions (metadata). It is also needed that the SaaS components comply with some interaction rules with the PaaS, using an appropriate protocol or API.

The function accomplished by a PaaS over an IaaS is similar to that accomplished by an operating system over the hardware. In both cases, the operating system (resp., PaaS) decides which and how many resources should be assigned to each running application (resp., SaaS), basing those decisions on the available resources, the needs of other applications (other SaaS) and the existing metadata about such running application.

The PaaS service model should manage the following aspects:

- Life-cycle management and configuration, including component dependences. Distributed applications provide distributed services. A distributed application is composed by multiple programs, each one with a given functionality. Service configuration demands a specification of the set of programs that compose that service, which dependences exist among them and how the system administrator should deploy all those components. Those configuration and deployment managements should be carried out by the PaaS. Note that this is not related with service provision (a responsibility of the SaaS), but with the administration-related tasks of those elements. A PaaS system is responsible of all those tasks, making the SaaS free of all those taks, since they are not related with its main role.
    - Those managements should be automated using several types of mechanisms: **composition** (that state which programs compose that distributed application and which dependences exist among them), **configuration** (that set the resources needed by each program or component; e.g., the RAM size to be used in the VM where the components will run, the required bandwidth in the network connections, how many processor cores are needed in each VM...), **deployment** (some tools are needed in order to read and process a deployment plan) and **upgrading** (how to adapt all previous mechanisms each time the working environment changes, e.g. when workload variations arise, or when a component program is updated).
- Performance model. In order to deploy an elastic service, we should build a performance model that computes is performance depending on the received workload and the existing instances for each one of its components. To this end, that model relates a set of relevant parameters, since they determine the achievable performance. However, the concrete set of parameters to consider usually depends on the service functionality. Although there is no clear set of parameters to be used in all cases, both service time and request arrival rate are two good candidates. The resulting performance model should be used as a base to take scaling decisions (i.e., to increase or decrease the amount of component instances), complemented with:
    - Automatic monitoring of relevant parameters. The parameters that define the model should be monitored periodically. Thus, the current (or, at least, recent) service state may be determined. Moreover, the monitored values may be kept in files in order to analyse their evolution and identify their current trend. This may facilitate a predictive scaling strategy, since a scale-in or scale-out action needs several seconds to be completed and it is convenient to start them in advance.
    - Definition of elasticity points. As it will be explained in subsequent units, when a service is deployed onto a PaaS system, a *Service Level Agreement* (SLA) should

be set. The SLA specifies which thresholds need to be considered onto several non-functional service characteristics (availability, response time...). The PaaS, with its performance model, should derive the associated thresholds on the relevant deployment parameters stated above (e.g., request arrival rate) in order to start the appropriate scaling actions. Its goal is to minimise the set of hardware resources being needed in that service deployment in order to guarantee a QoS levels stated in the SLA. These thresholds on the relevant parameters are also known as "elasticity points".

- Automatic workload-dependent reconfigurations. Service workload is varying along time, but the PaaS system is continuously monitoring several relevant performance parameters and has also defined the needed elasticity points. With that information, a PaaS may easily start the appropriate scaling actions in an automated way. Thus, services deployed on a PaaS are elastic, since they **minimise resource usage** and **dynamically adapt to the current workload levels**. Those two characteristics define what is known as **elasticity**.

4.4. Summary

Cloud computing is centred in efficiency and facility of use:

- Efficient share of resources:
  - Use only what you need.
  - Pay only what you use.
- Easy adaption to variable amounts of users and workloads.
- Provide easy ways for service implementation and service provision.

Cloud computing is based on three main service models structured in the following way:

1. Software as a Service (SaaS). Its goal is to provide applications as a service to a large amount of users.
2. Platform as a Service (PaaS). It automates resource management for SaaS providers. It also makes easy the creation and deployment of those services.
3. Infrastructure as a Service (IaaS). It provides elasticity to its upper layers.

5. Programming Paradigms

A basic system model usually promotes an asynchronous communication approach: after executing send events, agents can proceed with their execution without blocking. Likewise, agents go about their own business, handling receive events whenever they happen.

A particular kind of programming style well suited to represent the algorithms each agent executes is this:

1. Each algorithm is represented by a collection of actions $[A_1,\ldots,A_n]$, designed to be run atomically (no interruptions).

2. Each action A_i, is linked to a condition C_i (also referred to as a *Guard*). When condition C_i is true, action A_i is enabled, and can be selected for execution. At each point in time, an agent P selects one of its enabled actions for execution.
3. Conditions can depend on the local state.
4. Conditions can depend on the receipt of a message.
5. Actions can send a message at the end of their execution.

There are several programming languages operating under this principle, each offering different syntaxes to express variables, conditions, and actions.

Typical algorithm specification uses some variation of the C syntax to express the actions, conditions and data structure definitions (including message structure) in pseudo code.

In asynchronous programs, data structures must be defined to influence how actions should act once enabled. This works well to specify small, focused algorithms. However, in real world situations, this approach may become tedious and error prone.

To address this need, some programming languages allow the dynamic construction of the available actions, which have already been totally or partially configured with the data relevant to handle once they are enabled, reducing the need to maintain global data structures to direct their execution.

One typical way in which actions can be built is as closures in a functional language. The closure carries with it the access to the concrete variables (possibly local to a function) the function has to consult/modify when ran. The guard specifies as a condition the conclusion of some asynchronous action (reception of a message, completion of a system call, …). Free variables in the form of arguments to the functions may be specified, in which case, the guard must refer to the source of data for those arguments, and the run-time must bind them when it calls the function closure for execution.

For example, in a JavaScript-based environment like NodeJS, callbacks are function closures. Those functions may specify extra parameters that must be filled before the callback is executed (once enabled). This strategy avoids the need of maintaining global structures keeping that extra data needed for the action to proceed with its execution.

5.1. Concurrent State-sharing Paradigm

Agents are modeled as executing atomic actions at each event. If we think of a server in a client-server system, this means the server waits for a request message to arrive, and then processes it to completion before returning the result to the client. This was, in fact, how early servers were programmed.

The advantage of this approach is that atomicity is guaranteed: while a request is being processed no other request can interfere.

However, there is a problem with this simplistic approach. Servers often need to perform requests themselves on other elements: other servers, or the OS on which they run (which could be thought of as a server too). Waiting for the result of those requests would block the server,

and consequently render it incapable of attending further requests from other clients in the system. The end result is a very unresponsive system.

A technique that avoided this blocking situation was: the main server that waits for client requests should launch another process on each request reception. This secondary process handles the client request.

Process scheduling by the OS allowed both the main server and the secondary processes to run concurrently. The main server, after offloading the request to the created process, went back to wait for new client requests.

This way, responsiveness of the service was preserved, as blocking of a secondary process does not block the main server process ability to attend new requests. Global state between the main server and all sub-processes was cumbersomely shared using files (a globally accessible data store within an OS instance). Access to those files needed to be coordinated by the different processes to implement the "atomicity" of operation required by the logic of the server. Coordination was achieved using OS mechanisms for file access.

The awkwardness of the state sharing mechanism promoted a style of programming where global state sharing was minimized: Most of the information needed by the secondary process was prepared for its exclusive usage (including the request from the client). Thus coordination via shared files was kept to a minimum, and there were few problems derived from concurrency control of the various secondary processes.

Spawning sub-processes is inefficient both in time, and resources demanded from the OS. Consequently, multi-threading approaches, where concurrency occurred within ONE process were explored. The server no longer spawns processes to deal with arriving requests; instead, it sets up concurrent threads of execution within its own address space.

Under this approach, each arriving request is handed over to a different thread. State sharing is trivially achieved among all threads, as they all share the address space of the process in which they run. This facilitates a style of programming in which global shared state is used for communication and coordination among threads, requiring usage of concurrency control mechanisms (semaphores, monitors, …) to implement the implicit "atomicity" required by the logic of the server.

As with spawned processes, threads can independently block without blocking the rest of the server, which remains responsive. Moreover, thread handling is more efficient in resources than process handling (the OS has to do less work), and shared state access is much faster.

This has been the prevalent pattern for building servers until today. In fact multithreading is supported directly in some programming languages. Languages without first-class syntactic support can also be used, by linking against threading libraries that provide creation of threads, as well as access to a variety of concurrency control mechanisms.

However there are at least two main disadvantages to using threads:

1. Threads, though cheaper than whole processes, also consume precious resources. Thread creation and destruction is not cheap, and, even though thread pools can be used to alleviate creation/destruction costs, they must be carefully tuned to avoid wasting too many resources. In addition, implementation of concurrency control primitives also incurs sizeable overheads. Taking into account that the main motivation to use threads is to allow them to block independently, what we will have is that most created threads will be doing NOTHING most of the time.
2. Concurrent programming with many shared variables is prone to programming errors. Such errors, may lead to state inconsistencies (atomicity is violated) or blocking of the server (atomicity may be preserved, at the expense of making progress). Despite emphasis placed in many curricula on concurrent programming of multi-threaded systems, the situation does not seem to improve sufficiently. This imposes a barrier to the speed at which code can be produced, and the reliability of the end product.

5.2. Asynchronous (or Event-Driven) Paradigm

Asynchronous programming environments operate similarly to the guard-action programming model presented earlier. They are not new, they have been around as event-driven systems for quite a while, specifically in the realm of human-computer interaction (The user is constantly sending requests to the program in charge of handling the user interface. Those requests have been typically handled as asynchronous events within an event loop).

In practice, *async* programming environments use only one thread within a process. State is, thus, never concurrently shared among various activities, making it unnecessary to use concurrency control mechanisms. Exactly like we had in one-process servers.

Async programming environments must overcome two difficulties

1. Avoid blocking the only thread available in the process (reproducing the disadvantage of one-process servers)
2. Make it reasonably easy to actually write the software. This implies in turn
    a. Facilitating the organization of the state an action needs to handle when executed
    b. Facilitate expression of conditions.

Avoiding blocking requires that all requests performed from an async environment be, themselves, non-blocking (async). The main challenge here is in the calls to the facilities of the OS on which the process is executing. Often, some of those calls are blocking, and they must be converted using multithreading techniques by the run time itself (but not the actual application level software, which sees only one thread).

A common way to avoid blocking is by using a callback style. When making a request, one of the parameters is a function. When the request is done, the function is called with the result of the request as parameters:

```
function handleResults(r) {
        …
}
readFile(f, handleResults);
next_instruction();
```

What this program is doing in essence is this:

1. It creates a condition: “the call to *readFile* finishes”.
2. It defines an action by means of a function definition, *handleResult*.
3. It sets up the above condition (“the call to *readFile* finishes”) as the guard of *handleResult*.
4. It continues executing the rest of the program (in this case, starting with next_instruction), with the implicit understanding that *handleResult* will be executed LATER (it has just created the condition while executing some ATOMIC action: it cannot be interrupted by an action, even if its guard becomes true and the action gets enabled as soon as it is set up).

When *handleResult* is called, it receives the data produced by the *readFile* request. Thus, the callback style seems to be addressing our concerns: avoids blocking by making all requests asynchronous, facilitates creation of dynamic conditions (the conclusion of requests), and makes it easy to handle the state an action needs to use (parameter passing in the callback).

In reality, it is still necessary to be careful with long-running computations, as well as taking special care for state handling.

Long running operations may tie the only thread, rendering it unable to execute any other enabled actions. This may lead to unresponsive systems. If a programmer is not careful, the run time can do nothing to avoid this situation.

To avoid it, the programmer must split the long-running computation into atomic chunks. This may be easy with proper support from the programming environment.

For example, assume we want to compute the factorial function without tying up the only available thread. We could use this approach

```
function factorial(n) {
  if (n == 0) return 1;
  executeLater factorial, n-1
}

factorial(10000);
```

In this pseudo-code snippet, we have assumed the runtime has an operation, executeLater, taking a function and the argument it should receive, acting as follows:

1. It creates a condition that is made immediately true.

2. It associates the condition with an action consisting in executing the function argument, with the data arguments as parameters of the function call.
3. It exits the execution of the function.

So, when *factorial*(10000) is executed, what would have been a continuous long computation within the ONE thread, is converted into 10000 very short actions, giving other enabled actions a chance to execute.

State handling as presented so far, happens as a result of parameter passing to the callbacks. Most runtime environments allow access to some global state in the process. Access to the global state is not concurrent, but in some corner cases when making other requests, atomicity assumptions may be broken, and care should be taken to avoid introducing inconsistencies in the state, by properly building the conditions that enable the callbacks.

In addition, when constructing callbacks it will be necessary in many cases to be able to pass data/state additional to the one the callback receives via its parameters. Different language environments enable this differently. In functional async environments (e.g., JavaScript) the usual way is to pass closures as callbacks, where closures encapsulate access to variable contexts.

6. Case Study: Wikipedia

The goal of this section is to illustrate what kind of problems may be found in current networked information systems, taking as a basis a concrete example where we identify some challenges that generate other problems once they are confronted. As the name of this course suggests ("Networked Information Systems Technology"), we centre our efforts in understanding several technologies that should be known and used for solving those challenges.

Note that a single example cannot summarise the existing variety of problems and challenges to be analysed in this course but, at least, it provides an introductory context to begin with.

The best well-known service in the distributed systems field is, probably, the world wide web. Among its existing web sites, some of them stand out[1], as:

- Google[2] search, with more than a million servers,
- Facebook[3], with more than 1000 million users,
- The official Chinese train booking web site[4], with more than 1000 million clicks per day,
- Amazon, eBay and Alibaba, whose sales in 2013 reached[5] almost 100000 million dollars,
- The Netflix[6] and YouTube video services

[1] **Although the mentioned dates are not current, the orders of magnitude of the data are still representative.**
[2] https://www.google.com/
[3] https://www.facebook.com/
[4] http://www.12306.cn/, according to http://www.chinadaily.com.cn/china/2012-01/09/content_14406526.htm
[5] http://uk.reuters.com/article/2014/09/19/alibaba-ipo-idUKL1N0RK1KW20140919
[6] https://www.netflix.com/

- In 2015, Netflix has 60 million subscribers[7] that download an average of 45 GB per month… each one! In USA and Canada this represents 35% of the total Internet traffic.
- YouTube figures[8,9] are even larger, with more than 1000 million users and 7000 million of requests per day (in 2015[10])!

Although those systems are very large, similar problems may be easily found in smaller services:

- The popularity of a modest website, designed for a few users, may be boosted when it is mentioned in any widely known communication medium, provoking the "*Slashdot*[11] *effect*"; i.e., a collapse and "death" of that service caused by its impossibility to manage all incoming requests.
- Some services are specifically designed, built and configured for managing a short (although widely known) event. While the event takes place they should provide a high level of service, managing millions of users. Some examples are different ATP tournaments (Wimbledon[12], Roland Garros[13]…), the Summer Olympic Games[14], or the FIFA World Cup[15].

Therefore, we should choose a highly scalable and demanded web site, providing good examples of the existing challenges and their solutions. To this end we should look for sites with good documentation. Unfortunately, this is not the common case in private companies since their infrastructure may be tagged as confidential. On the other hand, academically, it is advisable to simplify all details in order to clearly explain the overall function of a system.

So, considering all those aspects, **Wikipedia** is a good candidate since it has a well-documented architecture and its architects have identified and solved many different challenges. Additionally, Wikipedia was awarded with the 2015 International Cooperation Prize from *Fundación Princesa de Asturias*[16] and this also confirms its social relevance, providing other reasons to select it as the best example.

[7] http://expandedramblings.com/index.php/new-updated-netflix-stats/
[8] https://www.youtube.com/yt/press/statistics.html
[9] http://expandedramblings.com/index.php/youtube-statistics/
[10] http://www.forbes.com/sites/edmundingham/2015/04/28/4-billion-vs-7-billion-can-facebook-overtake-youtube-as-no-1-for-video-views-and-advertisers/
[11] https://en.wikipedia.org/wiki/Slashdot_effect
[12] http://www.wimbledon.com/index.html
[13] http://www.rolandgarros.com/en_FR/index.html
[14] http://www.olympic.org/london-2012-summer-olympics
[15] http://es.fifa.com/worldcup/archive/brazil2014/
[16] http://www.fpa.es/en/princess-of-asturias-awards/laureates/2015-wikipedia.html?texto=trayectoria&especifica=0

6.1. Wikipedia Nowadays

The jury of that prize summarises the relevant data of Wikipedia in the date such award was given:

"Established in 2001, Wikipedia is a free-access digital encyclopedia written in a variety of languages by volunteers around the world, whose articles can be edited by registered users. It uses wiki technology, which facilitates content editing and storage of the page's history of changes. Founded by American entrepreneur Jimmy Wales, with the help of philosopher Larry Sanger, it has been overseen by the Wikimedia Foundation since 2003.

Wikipedia started on 15th January 2001 as a complementary project for Nupedia, an encyclopedia written by experts whose articles were peer-reviewed [...]. Both projects coexisted until the success of Wikipedia ended up eclipsing Nupedia, which ceased operations in 2003. Figuring among the ten most visited websites in the world, the encyclopedia has continuously grown. It currently offers more than 35 million articles in 288 languages [...]. English Wikipedia is the most extensive, with over 4,800,000 articles. Wikipedia currently has more than 25 million registered users, about 73,000 of whom are active editors, and receives about 500 million single visits a month.

Created in 2003 with headquarters in San Francisco (USA), the Wikimedia Foundation controls and manages Wikipedia. It also has other projects that complement the encyclopedia, all of which are multilingual, open and supported by wiki technology [...]"

Considering[17] the usage of Wikipedia (around 18500 million pages monthly accessed, 12th in the Alexa ranking), its reduced number of employees (around 400) and that it survives with what is being donated by particulars (around 170 million dolars), we should understand that its Foundation has been very cautious investing money in such system infrastructure and technology.

6.2. MediaWiki, the Wikipedia Engine, is a LAMP System

The Wikipedia engine has been developed by the Wikipedia administrators and evolved with time. It is based on the Wiki[18] concept: "A wiki is a website which allows collaborative modification of its content and structure directly from the web browser. In a typical wiki, text is written using a simplified markup language (known as "wiki markup"), and often edited with the help of a rich-text editor."

Wikipedia has boosted this technology, developing its software since January 2001[19], date in which they created UseModWiki[20]:

- Written in PERL, on LINUX, and storing its information in regular files instead of databases.

[17] https://stats.wikimedia.org/, https://wikimediafoundation.org/about/annualreport/2019-annual-report/, data from 2019
[18] https://en.wikipedia.org/wiki/Wiki
[19] https://en.wikipedia.org/wiki/History_of_Wikipedia#Hardware_and_software
[20] https://en.wikipedia.org/wiki/UseModWiki

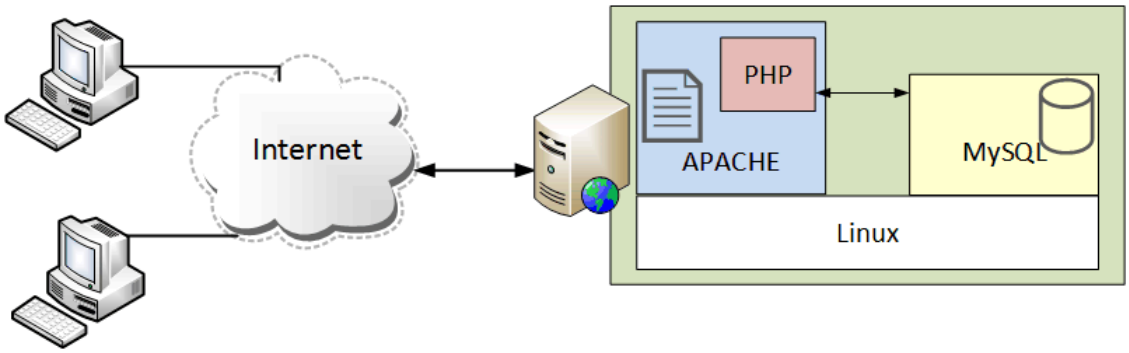
This software was replaced by Phase-II at the beginning of 2002, using a wiki engine written in PHP, but in July it was again improved (Phase-III = MediaWiki) to adapt itself to the growth of the project. In July 2003 a second server was added, deploying on it the database (separating it from the web server) and the webs in other languages. Other details may be found in Sections 1 to 4 from Chapter 12 (MediaWiki[21]) from the book "*The Architecture of Open Source Applications, Volume II*".

These stages show the evolution of Wikipedia software and they always share their technological kernel: they are LAMP systems. This means that they consist of four main components: Linux + APACHE + MySQL + PHP.

- Linux is a UNIX operating system.
- APACHE is a web server.
- MySQL is a relational database management system (DBMS).
- PHP is a scripting language, with many libraries, easily generating hypertext documents and simplifying the interaction with relational DBMSs.

LAMP platforms adhere to the 3-tier model of client/server applications:

1. The **user interface** layer contains the components that provide the application user interface (presentation logic). In this case, it is implemented by the hypertext documents returned by APACHE and shown by the client web browser.
2. The **application** layer (or business logic) holds the application processing rules, without considered concrete data. This corresponds to the PHP programs in this example.
3. The **data** layer (or persistent layer) maintains the information that the users access through the other application components. In this example, it corresponds to the database being managed by MySQL.



A **simplified** sequence of the processing of a client request could be:

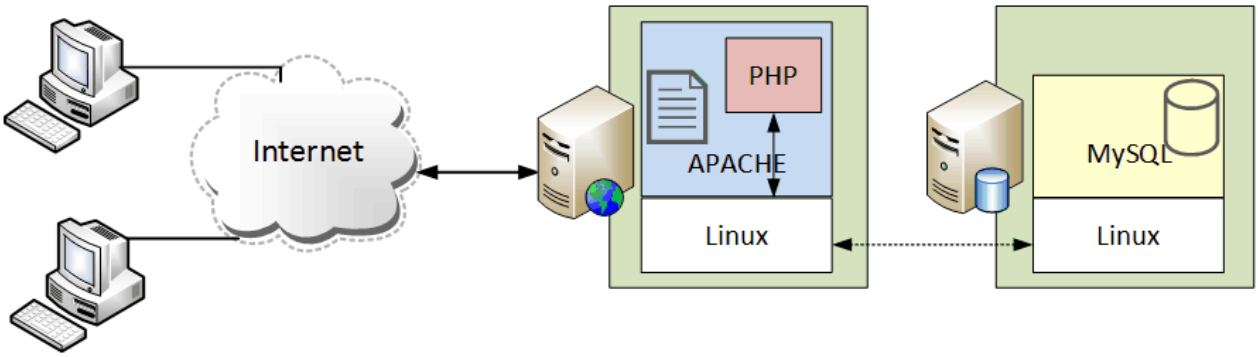
0. Initially, there is an APACHE process and a MySQL process waiting for incoming requests.
1. The client sends an HTTP request to the server. Such client remains blocked, waiting for an answer.
2. The APACHE server is woken up by the operating system. Now it can process the incoming request.
3. APACHE determines that a PHP program should be run, passing the data to it. Since the PHP interpreter is embedded in the APACHE process, APACHE remains running.

[21] http://www.aosabook.org/en/mediawiki.html

4. The PHP program is run sending SQL sentences to the DBMS. The PHP program (i.e., the APACHE process) waits until the DBMS returns a reply.
5. The MySQL server is woken up by the operating system, receives the request and executes its sentences, returning a reply to the PHP program. MySQL is waiting now for new requests.
6. Steps 4 and 5 may be replayed several times. In the end, the PHP program builds a page that APACHE sends to the client. Once this action is completed, the PHP program ends and APACHE waits for new requests.
7. The client receives the reply. Its browser shows the received document in its window.

Since each server is being run as a different process, it is not compulsory that the DBMS and the PHP program stay in the same computer.

- The difference between nodes (computers) and components (in this example, APACHE and MySQL servers/processes) is very important when we design the architecture of a distributed system.
- Note also that it is not possible to run the PHP program in a computer different to that where the APACHE process is being run.



So... what is MediaWiki? MediaWiki is a concrete example of LAMP system that includes scripts in its PHP component, plus a given configuration of the involved servers (APACHE and MySQL), several databases with tables and several hypertext contents.

6.3. Using MediaWiki in Wikipedia.

Most of the statistical information about Wikipedia is related to the number of editors, wiki pages and available languages. We may also find some figures about its amount of read accesses, but we should process all those data in order to find how many resources are needed.

For instance, in order to know the needed network bandwidth, once a monthly access rate is known, we may multiply the average page size per that rate, thus obtaining the gross bandwidth needed for read accesses[22]. To this end, let us find two different page sizes:

- Short article: *albufera* (1130 KB according to the inspection console from Firefox)
- Long article[23]: *United States* (7210 KB)

[22] Data from July 2020 to June 2021 available at https://stats.wikimedia.org/#/all-projects/reading/total-page-views/normal|bar|1-year|~total|monthly. On average, there are 24 billion accesses per month.
[23] Actually, this is not the biggest size, but we are only looking for any approximated value.

Therefore, the global monthly network traffic is in the range 27120 TB[24] to 173040 TB, and assuming a uniform access rate, the needed bandwidth would be in a range from 83.70 Gbps to 534.07 Gbps. According to some sources[25], peak workloads may reach three times this average, so they require a bandwidth between **251 Gbps and 1602 Gbps**. No network infrastructure can reach those values nowadays.

Could this workload be processed by a single server (i.e., using a single communication line)? Let us try.

6.3.1. Internet Access

An internet service provider (ISP) may host a server in its buildings for ensuring an optimal access to the network. Considering data[26] from 2014, in USA, the best conventional output bandwidth was 60 Mbps. With more recent data from December 2017, the best output bandwidth for service providers was that of Xfinity in the USA with 2000 Mbps. In August 2021[27], the best provider is Google Fiber in the USA, but still with 2000 Mbps.

In the worst case, we would need **801 lines from the best Internet connection** in the world!!

Let us look for other alternatives... The first one consists in avoiding sending all that amount of information. There are many applicable techniques, such as the reusage of style sheets, icons and scripts. Once the browser has loaded them, it may reuse them many times. Unfortunately, this does not ensure enough bandwidth savings. Other techniques are still needed.

A second alternative consists in "distributing" the service. This means that multiple servers are spread through the world, considering the aggregate bandwidth of them all. But using "multiple servers" raises many problems... since they all provide the same service:

- Are the servers identical clones? What is their entry point?
- How are the updating operations managed?
- What happens if a replica is desynchronised from the rest? Should they interact with all the others or only with a central manager?
- Is there any backup in case of failure?

These new problems may be accepted if their benefits exceed their costs but, in any case, all these difficulties are hard to solve.

It can be ensured that replicating elements is always problematic, since system reliability depends on components reliability: an increase in the number of elements is a potential problem. Let us see why. Let us assume that we need 10 replicas to improve throughput and that the probability that in a given day a concrete replica fails is 0.1%; which is the probability that none of the replicas fails in a month?

- No failure of a replica in a day: 1-0.001

[24] 24,000,000,000 accesses/month = 9,259.26 accesses/s, multiplied per 1130 KB and per 8 bits, and expressed in Gbps

[25] https://grafana.wikimedia.org/?orgId=1

[26] http://www.pcmag.com/article2/0,2817,2465506,00.asp

[27] https://www.highspeedinternet.com/resources/fastest-internet-providers, July 2021. In September 2022, Xfinity announces 3000 Mbps.

- No failure of a replica in two days[28]: (1-0.001)*(1-0.001)= $(1-0.001)^{2}$
- No failure of a replica in a month: $(1-0.001)^{30}$=97.04%
- No failure of any replica (there are 10) in a month: $((1-0.001)^{30})^{10}$=74.07%

As a first partial conclusion: *the greater is the amount of components, the lower reliability will be.* Therefore, to increase throughput using more components worsens reliability.

How can this be solved? With a careful design where components are monitored and, as soon as any of them fails, it is detected and replaced, retaking such interrupted service. Easy to say, but difficult to design and implement!!

As a second conclusion: *the problems being introduced by replication may be solved with additional replication.* The first "replication" is intended for increasing throughput, while the latter tries to solve the reliability problems introduced by the former.

These considerations make sense, with their particularities, for all components of the LAMP system the Wikipedia is based on.

6.3.2. The APACHE Web Server and the PHP Scripts

This software receives client requests, using the HTTP protocol, and decides whether a static content should be returned or a PHP program should be executed. In the first case the server behaves as an intermediary between client and contents; in the second case it forwards the client data (e.g., a query for a local searcher) to an application that should return a tailored answer.

The first part can be easily accelerated: using more memory!! If we could place all (static) documents in main memory, we could access them very fast (but this is too much expensive!!). Since this is not attainable, some kind of cache memory is used. It is called a *reverse proxy* and it holds the most requested documents, images or other types of static content. Some products of this kind are *Squid, Varnish* and *nginx*.

May we place several instances of a reverse proxy when a single one does not boost enough performance? Since query operations are regularly independent, yes, we can. To this end we need any criterion for relating the name of the resource to be obtained to the address of the instance to be used.

- For instance, to use as many proxies as letters in the alphabet. Each one holds a copy of the static objects that begin with that letter.

We already saw that an increase in the amount of components increases also the error probability. Therefore, we should also add more reverse proxy instances to deal with the possibility of failure in any of them.

Let us discuss now what could be made with client requests associated to dynamic content; i.e., requests that imply the execution of a PHP script. There are four main classes:

[28] It should not fail in the first day NOR in the second.

1. **Write operations** that create or update a Wikipedia article: what happens with all copies of that article that have been placed in the reverse proxies?
    - We need to ensure their consistency (i.e., that all copies hold the same contents). In the common case a signal is sent invalidating all those copies, compelling their processes to refresh those contents.

2. Pages that **depend on the requesting user**, determining which contents should be shown and which operations are admitted.
    - This is similar to a session and, as such, demands service continuity. Therefore, the replica that served the first user access in that "session" should remain attached to it, processing all further requests from that user.
3. **Queries that find information** depending on some search terms. This is an activity commonly processed by the database server but it may be complemented with a cache of recent results for the same query.
    - Its main difficulty is to know in a precise way when a result has become obsolete.

4. **Queries that find associated contents** in order to obtain derived information: related links, page attributes, history of updates, etc.
    - These results are mostly static, so a caching strategy may manage this kind of requests.

As in the previous cases, a key aspect for improving throughput is the maintenance of multiple copies of the information to be returned, avoiding any (remote) effort for carrying or computing them again. Potentially, however, this is a source of inconsistencies.

This caching strategy reduces the amount of times that we need to execute programs or scripts. Its importance is not negligible but there are still many requests that cannot be replied in this way. Although their percentage is not high (e.g., 12% in the winter Olympic Games from Nagano in 1998) their processing power may be high, surpassing in many cases 50% of the global time devoted to processing power.

Although there are specific accelerating techniques for PHP programs, most of the time consumed by those programs is devoted to:

- The overhead involved in dynamic requests.
- The interaction with DBMSs. This aspect is further explained in the next section.

6.3.3. MySQL

The interaction with the database in Wikipedia is crucial: it is not only needed for organising the Wikipedia articles but also for collecting informations, drive the articles life cycle, manage users, maintain usage statistics, etc.

We have seen that one of the first structuring actions adopted in Wikipedia consisted in separating the database server, locating it in another computer. As a result, APACHE-PHP and the MySQL server interacted, from that time on, exchanging messages through their IP addresses and ports.

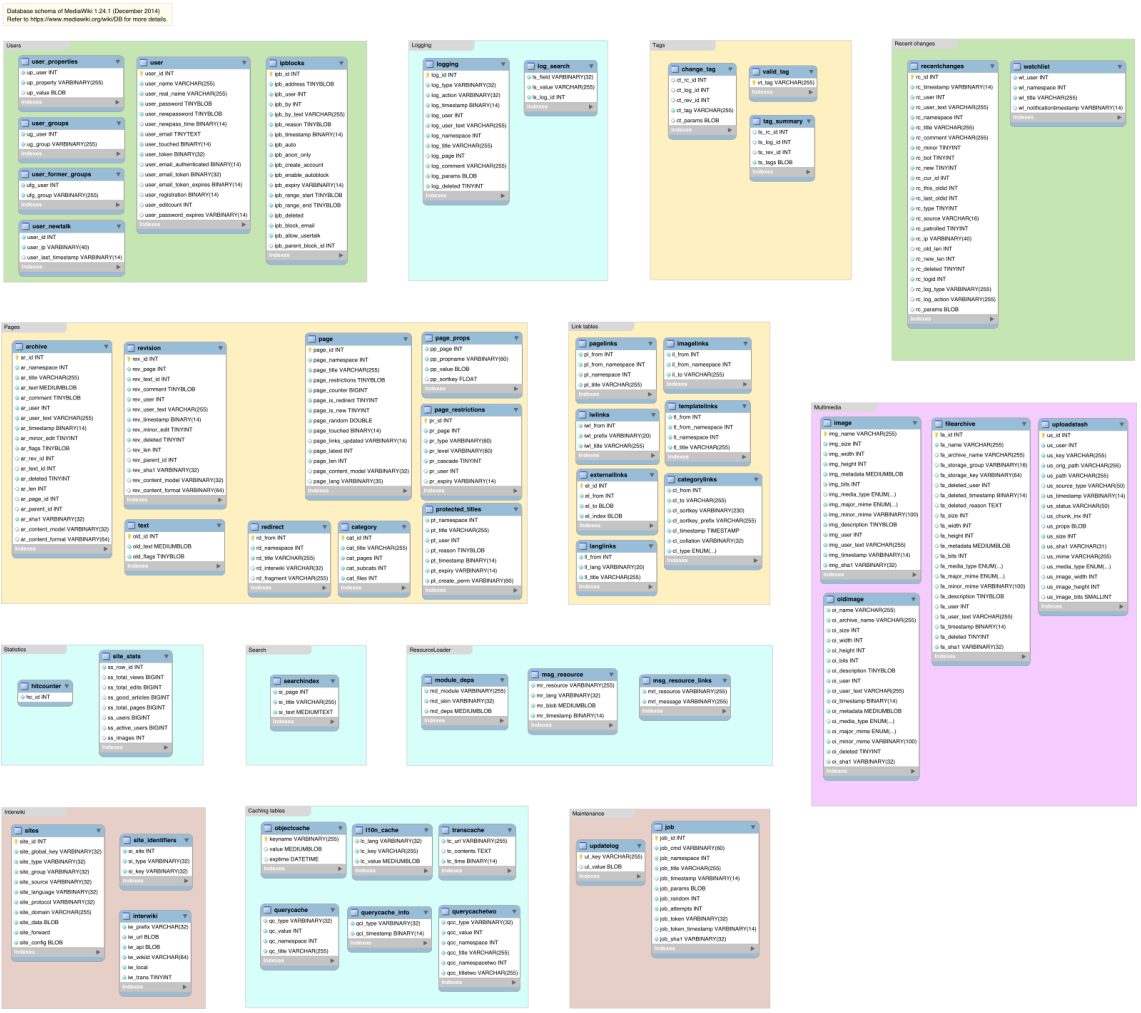
The web server was replicated later on. This introduced the need for replicating sooner or later the DBMS server itself. Otherwise (i.e., maintaining a single DBMS server instance), those services would be clearly unbalanced.

- Imagine a restaurant with a waiter and a chef. The waiter receives the requests from customers and forwards them to the chef. If we plan to admit many more customers contracting only additional waiters, we may easily foresee which will be the restaurant bottleneck.

DBMS replication is not an easy task if all replicas should manage the same information: we should check initially whether those contents may be divided in multiple independent databases.

- This depends on the usage of those data and on the relations among the existing tables.

Although this subdivision was already made, its result would certainly be complex[29]: 13 databases with a total of 50 tables in December 2014.



[29] https://upload.wikimedia.org/wikipedia/commons/f/f7/MediaWiki_1.24.1_database_schema.svg

Throughput constraints are not important for query operations, at least when they are compared with update operations. The databases that should be carefully tuned are those that serve a lot of updates. They may be grouped in different sets:

1. **Very frequent** updates, usually followed by multiple queries: *Logging* (it adds data in *append* mode regarding accesses), *Statistics* (self-explaining), *Caching tables* (it is used internally, in order to manage the caching strategies in an adaptive way)

2. **Edition-based** updates. They happen less often than the former, since they are caused by updates started by editors that modify the Wikipedia contents: *Multimedia* (all non-textual resources), *Recent changes* (self-explaining), *Pages* (the main articles database) and *Link tables* (reference links among articles)

3. **Eventual** updates: *Users* and all other databases. They are seldom updated.

The update frequency may be important although it is not decisive; e.g., updates without subsequent reads seldom raise conflicts (*Logging* and *Statistics*), but those updates that are immediately read may easily generate many conflicts (*Caching tables*, *Pages*…).

6.4. Wikipedia Architecture

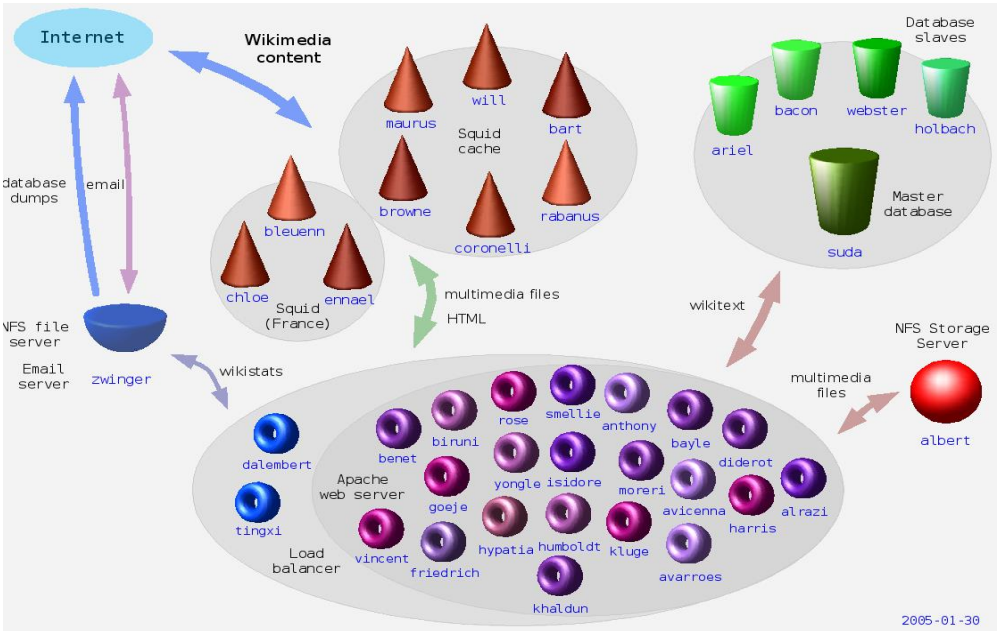
(This information is not completely updated since the sources[30] neither are. Configuration files may be found at http://noc.wikimedia.org/conf/)

6.4.1. Global Evolution of its Structure

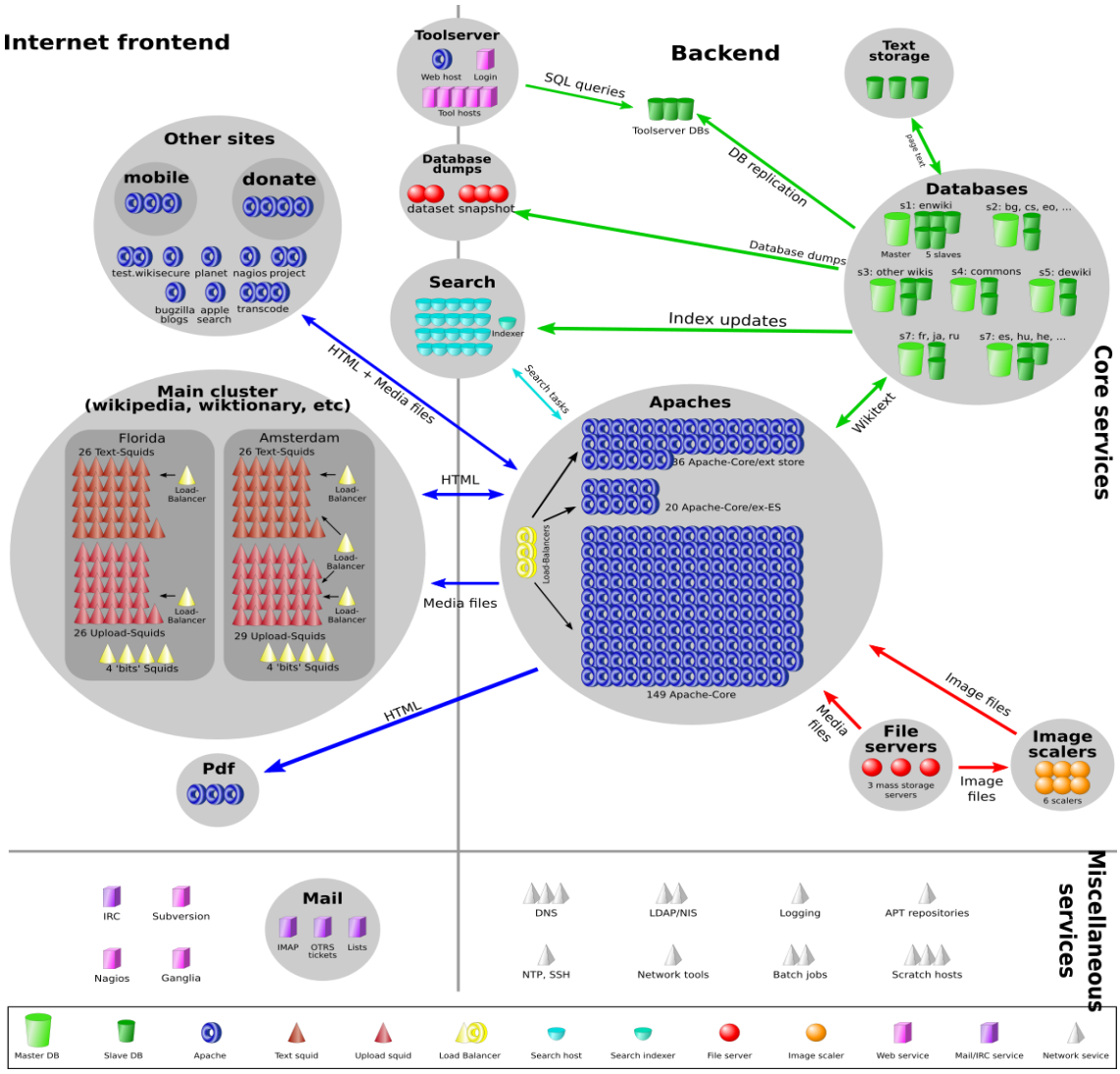
Starting with a minimal infrastructure, with a single computer that holds all components, Wikipedia evolves in hardware, organisation and additional services. In order to illustrate this evolution we present two snapshots from 2005 and 2010.

- **2005**. Multiple APACHE servers (although still with a name per computer), two reverse proxy services, distributed DB.

[30] https://meta.wikimedia.org/wiki/Wikimedia_servers

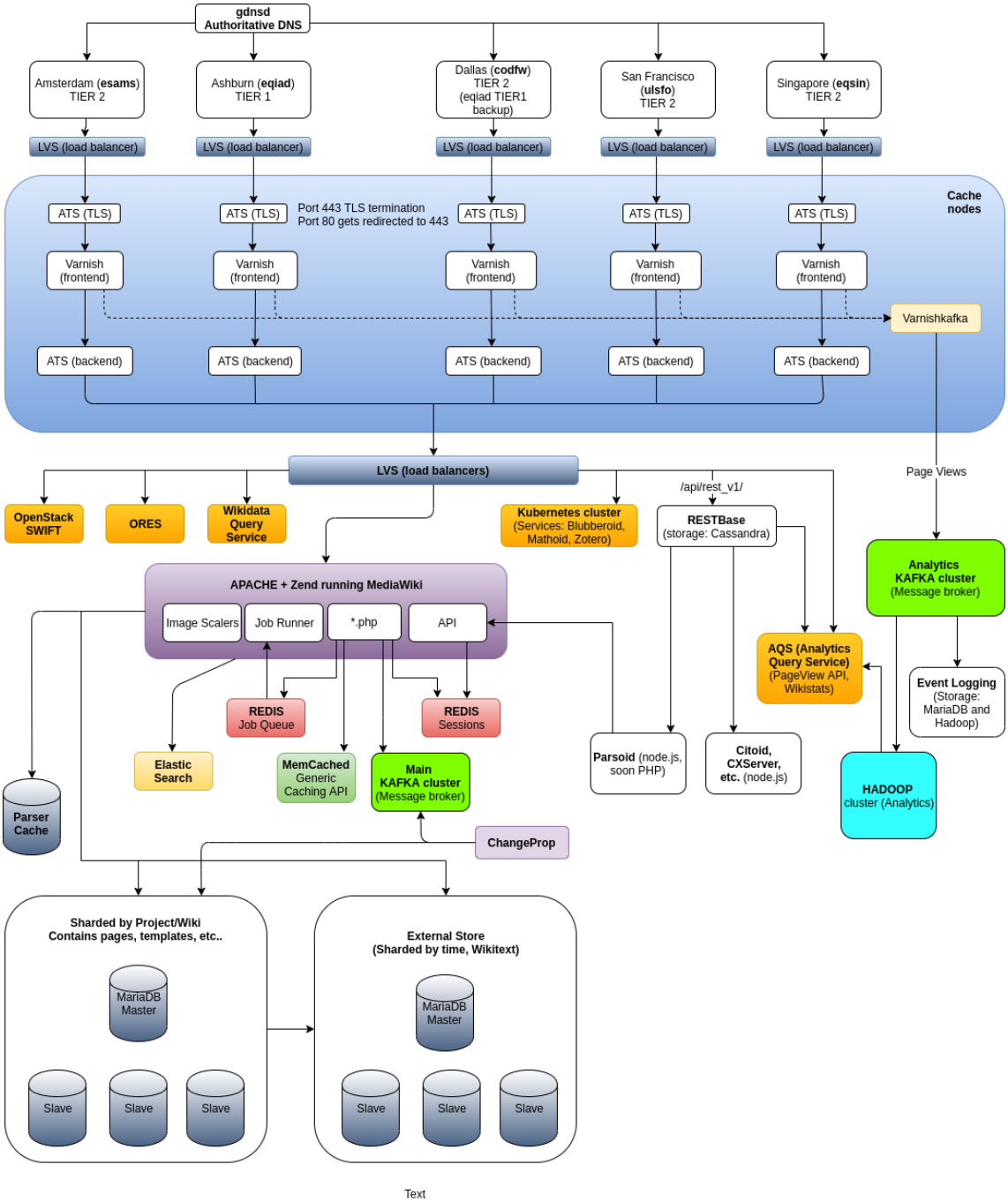


- **2010**. There are so many APACHE servers that their names become numbers, multiple specialised semiautonomous services (reverse proxies, databases, indexes), growth of other auxiliary systems, other projects of the Wikimedia Foundation start their growth.


Internet frontend
Backend
Toolserver
Web host
Login
Tool hosts
SQL queries
Toolserver DBs
Text storage
DB replication
page text
Other sites
mobile
donate
test.wikisecure
planet
nagios
project
bugzilla
blogs
apple
search
transcode
Database dumps
dataset snapshot
Databases
s1: enwiki
Master
5 slaves
s2: bg, cs, eo, ...
Database dumps
s3: other wikis
s4: commons
s5: dewiki
s7: fr, ja, ru
s7: es, hu, he, ...
Search
Indexer
Index updates
HTML + Media files
Search tasks
Wikitext
Core services
Main cluster
(wikipedia, wiktionary, etc)
Florida
26 Text-Squids
Load-Balancer
26 Upload-Squids
4 'bits' Squids
Amsterdam
26 Text-Squids
29 Upload-Squids
4 'bits' Squids
HTML
Media files
Apaches
36 Apache-Core/ext store
20 Apache-Core/ex-ES
Load Balancers
149 Apache-Core
Image files
Media files
File servers
3 mass storage servers
Image scalers
6 scalers
Image files
HTML
Pdf
IRC
Subversion
Nagios
Ganglia
Mail
IMAP
OTRS tickets
Lists
DNS
LDAP/NIS
Logging
APT repositories
NTP, SSH
Network tools
Batch jobs
Scratch hosts
Miscellaneous services
Master DB
Slave DB
Apache
Text squid
Upload squid
Load Balancer
Search host
Search indexer
File server
Image scaler
Web service
Mail/IRC service
Network sevice

6.4.2. Current architecture (data from 2019 to 2021)

In 2019, the system architecture may be summarised in the following diagram. Its top part (i.e. its three top layers, including LVS) shows the resources needed to facilitate world-wide access to the Wikipedia, while the bottom part depicts the resources that store data.



Text

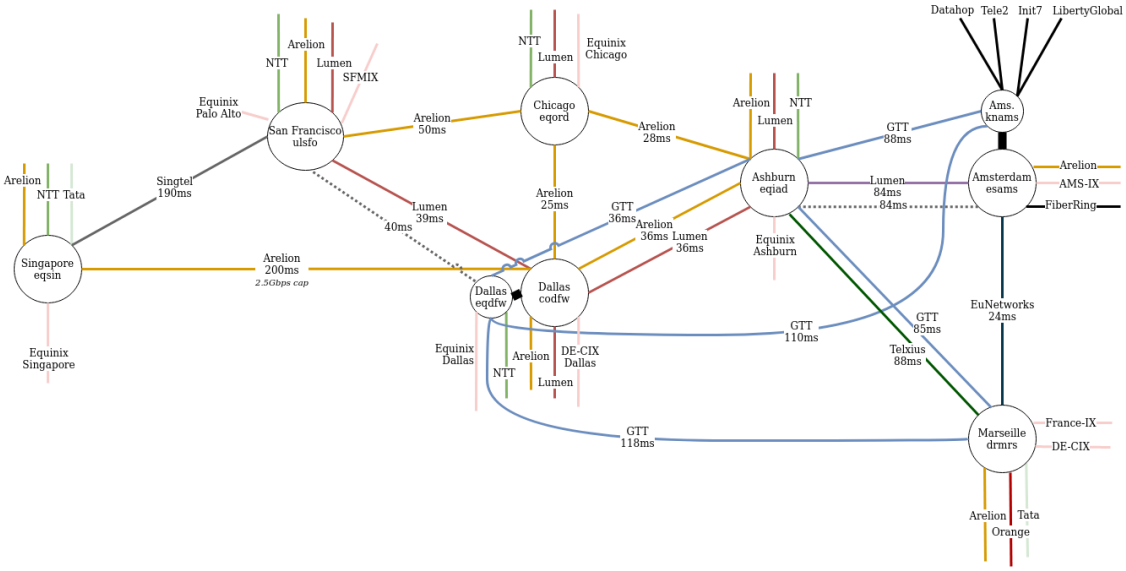
Although subsquent paragraphs explain some relevant components, a detailed study is out of the scope of this document.

- **Network design**[31]

In May 2018, Wikipedia spread its resources in these datacentres:

- **eqiad** (Equinix in Ashburn, Virginia). This is the main location. It includes the internal web servers and DBMSs.
- **esams** (EvoSwitch in Amsterdam, the Netherlands). A Squid caching cluster.
- **eqsin** (Equinix in Singapur). Caching for Asia and the Pacific area (besides USA).
- **ulsfo** (United Layer in San Francisco). Caching for West USA, China and Japan. Other datacentres will be used in the near future as complementary caching systems for Latin America, and other parts of Asia.
- **codfw** (CyrusOne in Carrollton, Texas). Emergency support.

The infrastructure that supports the internal communication among these centres and the interaction with Internet clients is depicted in this schema[32]:



Each of these nodes has three or four ISPs (that handle the traffic from or to close clients) and other internal networks (that handle the communication with other Wikimedia sites). The resulting scenario depicts a heterogeneous and complex world-wide system.

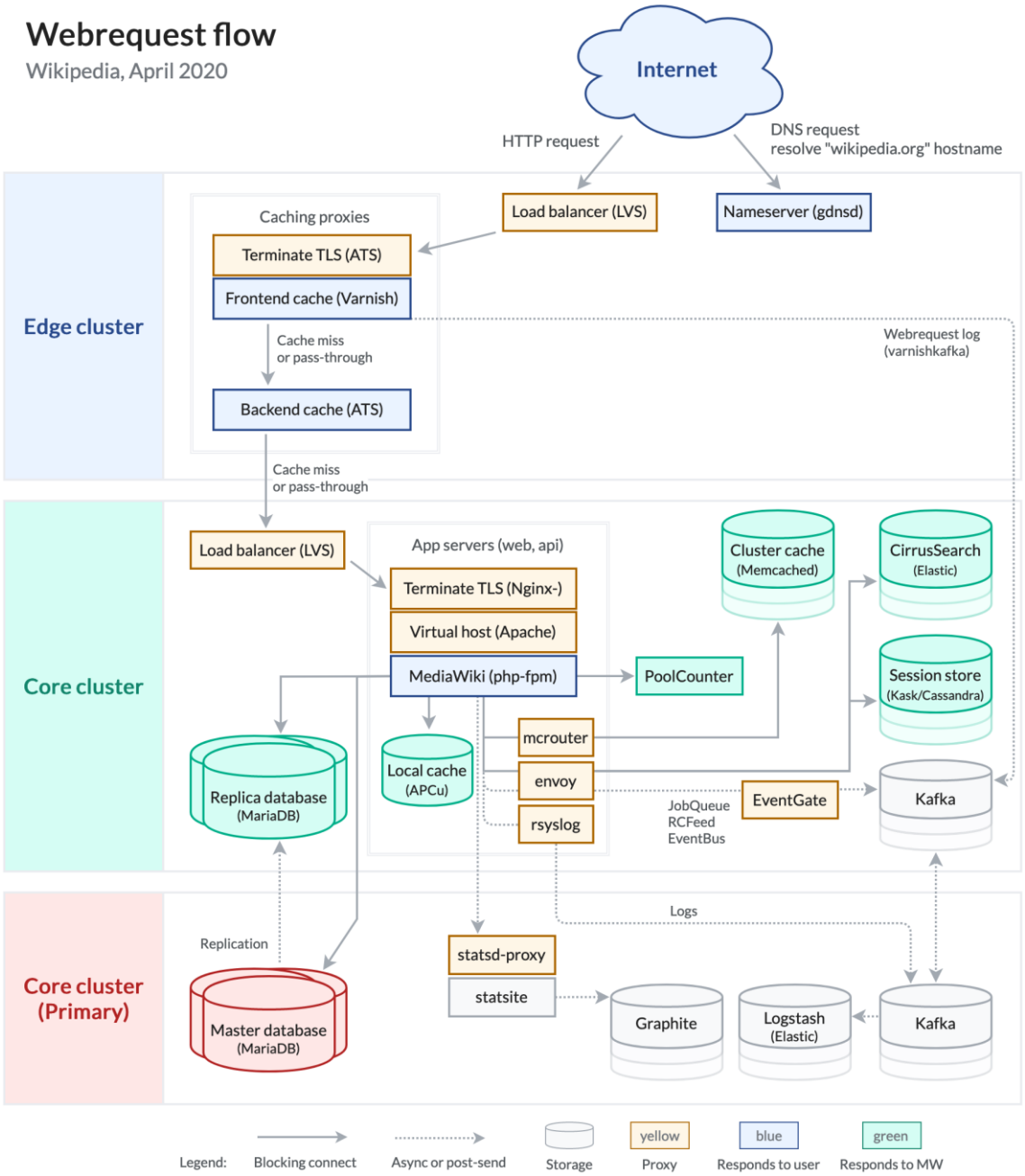
- **Components**[33]

These components define a complex LAMP system, adapted to a very large deployment:

- Wikipedia uses *gdnsd* to geographically distribute the incoming requests among its five datacentres (3 in USA, 1 in Europe and 1 in Asia), depending on the client location.

[31] Additional information at: https://wikitech.wikimedia.org/wiki/Network_design
[32] In 2022, another node has been added in Marseille (France).
[33] https://meta.wikimedia.org/wiki/Wikimedia_servers

- Linux Virtual Server (*LVS*) balances the incoming workload. LVS is also used for spreading the internal MediaWiki workload. Internal monitoring is carried out by PyBal.
- Regular web requests (API and Wikipedia entries) are hurried up by reverse proxies (Varnish and Apache Traffic Server).
- All servers run on *Debian GNU/Linux*.
- Distributed objects are stored in *Swift* databases.
- The main web application is *MediaWiki*, written in PHP (~70 %) and JavaScript (~30 %)
- Since 2013, *MariaDB* stores the structured data. Wikis are grouped in clusters and each clusters is managed by several MariaDB servers, using passive replication.
- DB internal caching is managed by *memcached*.
- *ElasticSearch* (Extension: CirrusSearch) manages fast text searching.



This "simple" high-level schema depicts the main components of this distributed service. It is easier to understand it than a deployment-based figure that shows hundreds or thousands of computers with many network connections. A distributed system depends heavily on the type of components that build it, since they define its architecture.

7. Conclusions

Currently, information systems are networked systems. All sample systems being discussed in this unit are examples of *distributed systems*.

Adequate design and development require a thorough knowledge about concurrent programming and distributed architectures. In a distributed system there are two architectural interaction models: client/server and peer-to-peer.

There are some important aspects that guide the design of distributed systems:

1. Flexibility. A system is created with a set of initial user requirements, but those requirements are varied through time. The system should be able to adapt to those changes in its requirements.
2. Dependability. The system should be reliable and able to overcome failures, maintaining data integrity and ensuring service continuity.
3. Performance. System performance should be predictable. Performance is regularly expressed combining throughput metrics (e.g., requests/second) with latency (time needed to complete the service of a request).
4. Scalability. The system should be able to serve an increasing workload without requiring a redesign or upgrade of its components, ensuring always a given QoS.
5. Simplicity. The system should be easy to build, easy to administer and easy to deploy. Reconfigurations should be light and fast.
6. Elasticity [5]. Besides being scalable, a system should be also adaptive, minimising its resource usage without endangering its compromised QoS.

We have considered cloud computing as the latest important stage in computing evolution:

- Characterised by the efficiency in resource usage.
- With an access model of “pay per use”.
- With elasticity (that also includes scalability) as its main goal.

Another implicit property exists: security. Security may condition the technologies being chosen for implementing both the system components and the applications. Some aspects of the distributed system architecture have an impact on the security properties that may be achieved in the resulting system.

We have chosen Wikipedia as a world-wide service for identifying the problems that may arise in distributed systems, and their solutions. Since those solutions cannot be perfect, they provoke again other specific problems and those new problems are, hopefully, smaller.

As an evolving system, the solutions being developed have been also evolving throughout time, as it has happened to the technologies and available resources. No system of this scale may remain immutable, since its adaptation is a compelling requirement.

The figures (i.e., numerical values) in this case study illustrate the scale of the problems being confronted. The assumption of unlimited resources, besides being unrealistic, does not provide any solution: intelligence is needed for ensuring that the union of large sets of resources implies the union of their capacities and good properties in the resulting system.

In order to provide service, the incoming requests should be spread among as many nodes as possible, making each node responsible of a small part of that workload. Node replication is a good approach for ensuring service continuity and caching and reverse proxies increase service throughput.

We have dissected the Wikipedia system in order to find its internal components and learn their functionality. This allows a thorough study that may generate some improvements. The identification of components is a critical part in the design of the systems to be studied in NIST.

NIST is a course on distributed system architectures. Its main aim is the analysis of the design alternatives in the specification of distributed systems, considering the existing technologies and analysing their usage depending on the aspects previously cited.

References

[1] A. S. Tanenbaum and M. van Steen, Distributed Systems: Principles and Paradigms, 1ª ed., ISBN: 978-0130888938, Prentice-Hall, 803 pgs, 2002.

[2] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems: Concepts and Design, 5th Edition ed., Addison-Wesley, 2011.

[3] T. Grance and P. M. Mell, *The NIST Definition of Cloud Computing,* NIST, 2011.

[4] M. M. Michael, J. E. Moreira, D. Shiloach and R. W. Wisniewski, “Scale-up x Scale-out: A Case Study using Nutch/Lucene,” in *21th International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, USA, 2007.

[5] N. R. Herbst, S. Kounev and R. Reussner, “Elasticity in Cloud Computing: What It Is and What It Is Not,” in *10th International Conference on Autonomic Computing (ICAC)*, San Jose, CA, USA, USENIX, 2013, pp. 24-28.