Distributed systems

On this lecture, we will discuss an important area of technology which was not yet covered in this course, that is, networking and **distributed systems**. These allow delegating computing tasks or storing data to external machines.

Networking technology allows combining multiple computers into a single **distributed system** in which all the hardware resources can be used for more complex tasks. The word *distributed* here is intended to distinguish this case from *centralized* systems, in which all exchange of information happens via a single central computer. Yet another alternative to the distributed model is a single *supercomputer*, equipped with extremely powerful components. These approaches are presented on the following diagrams:

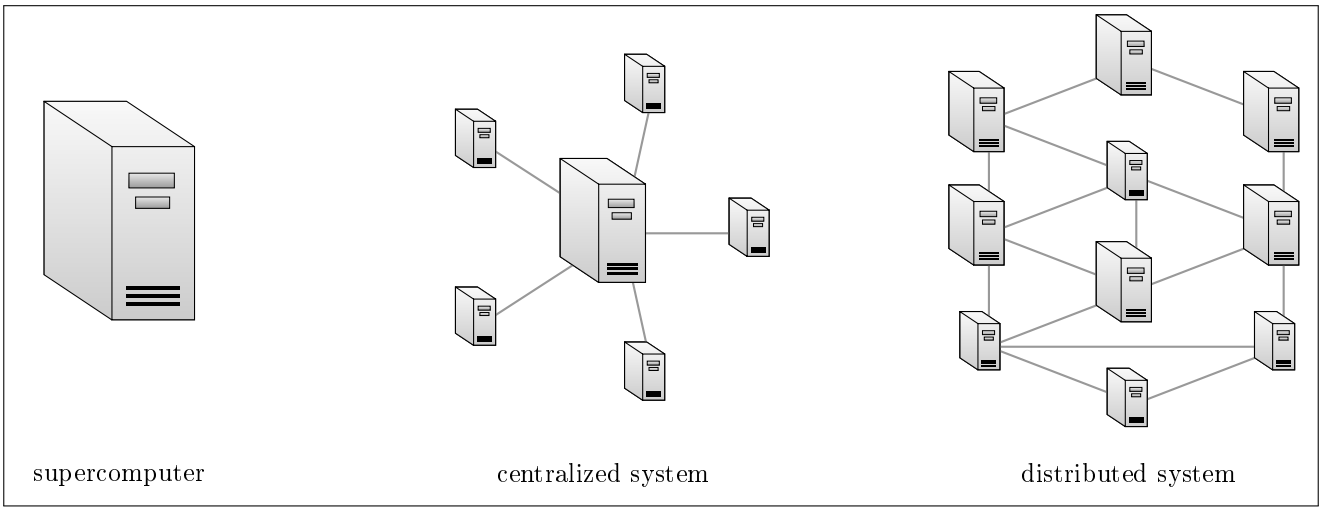

Figure 1. Types of large-scale computer systems.

Pros and cons

The advantage of distributed systems over centralized ones is typically better resilience to a failure of single node, or to network overload. In comparison to supercomputers, the advantages are following:

- increasing computing power at relatively lower costs;
- better resilience to hardware failures;
- quite easy scalability of the system
  (when more computing power is needed, just add new nodes).

On the other hand, a weakness of distrbuted systems compared to supercomputers is limited computing power of a single node, which matters in case of most complex tasks, in which parallelism may be hard to introduce, and longer latencies of transmitting data between threads may be not acceptable.

For these reasons, supercomputers are typically used for specific, most demanding tasks (e.g. complex metheorological models, intensitve training of artificial intelligence), while distributed systems are more often used for parallel execution of multiple, relatively simpler and often mutually independent, tasks.

Another disadvantage of distributed systems is the necessity for managing its complexity; moreover, in a decentralized way. This brings a whole palette of challenges, including:

- handling **asynchronous communication**, in which delivering messages takes a significant and unpredictable time (or can fail altogether);
- **replicating** information across different nodes (to make it available even in case of local hardware failures), and on the other hand, ensuring **consistency** of that information across different replicas;
- administering the system on the software level (e.g. ensuring compatibility between various operating systems, or between various versions of the same program — and/or introducing an appropriate strategy of updates).

These are all very interesting topics, though unfortunately they are outside the scope of this course.

Finally, let us note that many networks used in practice have an "intermediate" structure, that is, they are distributed systems built from *powerful* node machines (which allows efficient execution of more demanding tasks), though of course lying still far behind the top supercomputers. In such cases, when we want to expand such a system, we can choose between two strategies: **horizontal scaling** (i.e. adding new nodes, which will make the system "more distributed") or **vertical scaling** (i.e. improving the efficiency of individual nodes, which goes more in the direction of centralization).

Organzation at software level

We will now discuss a few main types of logical architecture of computer system. Depending on the ownership landscape of nodes and their role, these are:

- **Peer-to-peer architecture** (*p2p*): network nodes have diverse owners (e.g. often they will be simply personal computers of unrelated people) and have all the same role; each node may begin to be served by another node, upon establishing an appropriate bilateral conection between them.

  This model is used by popular *torrent* systems — each node in the network can send a request and start downloading data from any other node; similarly, each node can be sending files. Another example are instant messengers, like Skype.

- **Client-server architecture**: it distinguishes a central *server* (being either a single computer or — increasingly often — a whole computer system), exposing a specific functionality to other (typically numerous) nodes, which are called *clients*.

  The server is responsible for (almost) whole logic related to its functionality; in practice, the cient side may contain an additional user interface.

  For example, although the essential operation while browsing internet is sending requests and receiving responses from HTTP servers, it is also an important functionality to display the received web page in a legible way, and that is the task for a web browser (i.e. *user interface*) installed on the user's computer (i.e. on the *client side*).

In case when the server itself is a whole computer system, there are more approaches to choose from:

- **Cluster architecture**: the server consists of many machines, each of which takes its share of tasks which are similar to each other. In this model, the server itself becomes a computer network, though when speaking of a *cluster*, we typically mean a network of machines located close to each other. (In case of servers distributed to large distances, e.g. to diverse continents, we would typically describe them as a *network of local clusters*). Anyway, in all cases, it becomes important to effectively (which usually means: evenly) handle spread the work among the nodes, which is the responsibility of a *load balancer*.
- **Multi-layer architecture**: here, the functionality of the server is split into separate *modules* (or *layers*) assigned to different machines. In the simplest case, the server is split into a database and a web server communicating with the client. However, one can also distinguish a larger number of specialized layers (for example: authorization layer, cache, task scheduler).

In practice, naturally, we can meet mixed architectures, e.g. a multi-layer server in which the layer dealing with the main computations (and sometimes also other layers) will be a distributed system of a cluster architecture.

Let us also mention the **grid architecture** in which a distributed system is built of diverse machines, often belonging to various owners, which agree to dedicate some computing resources to a shared goal. Modern examples include volunteering projects for specific intensive computations (e.g. *Folding@home*, a network performing bioinformatical simulations for the purpose of fighting certain class of diseases); as another example, we can consider the block-chain networks for verifying cryptocurrencies (e.g. Bitcoin).

In the context of our classifications above, grid architectures do not allow simple labeling as a whole class; their classification depends on the particular case. Verifying cryptocurencies happens typically in a decentralized peer-to-peer network. In contrast, a typical *citizen science* project may be regarded on one hand as a single distributed *server*, whose service (computation) is used e.g. by the scientists at the root of the project; on the other hand, the individual machines involved in those computations are commonly described as *clients* which connect to a *server* (or *servers*) in order to receive subsequent shares from the central pool of tasks. (In this way — quite unusually — it is the *server* who requests work from the *client*).

Services in the cloud

In the modern distributed systems, special focus is usually put on:

- a specific **service** delivered by the system (for example: storing user's files, providing videos on demand, providing an environment for the customer's web app operation);
- ease of use, understood as hiding from the customers the whole technical layer and leaving them just with a well-defined **user interface**, through which the service is used;
- high **availability**, understood as minimizing the time of the service being inactive (due to any failures), but also minimizing the waiting time for connecting to the server and having the task done (which typically is achieved by distributing the servers over the world regions).

Such philosophy of building computer systems has a common name of **service-oriented architecture**. In the context of distributed servers of a global scope (i.e. possibly close to all users), and an

implementation hidden behind an abstract interface, this technology is commonly described as the **cloud**.

Below, you can see a graph illustrating the main providers of cloud solutions. As it shows, in the case of cloud, we typically observe a single owner controlling a whole ecosystem of services and managing that whole system in a uniform way. The service delivered by the cloud provider is usually also much more extensive than in the case of grid systems.

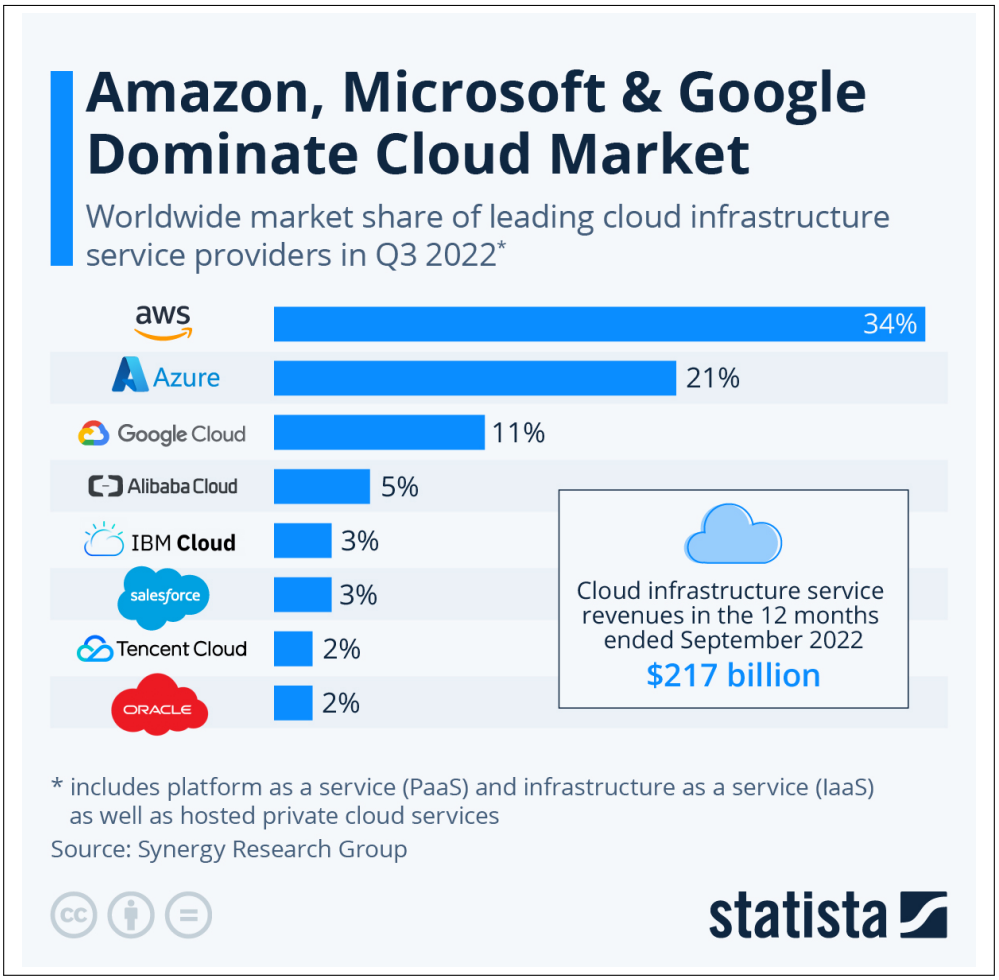

Figure 2. The leading providers of cloud services.

On the level of technical network structure, as a cloud system typically involves lots of users with independent computing tasks, cloud servers are almost always distributed systems; more specifically, clusters or distributed networks of clusters.

Levels of service

Considering the type of delivered services, we can distinguish 3 main models of cloud systems:

- **IaaS** (*infrastructure as a service*): here, the provider offers infrastructure, that is, access to servers together with maintaining them in an operational state, but no accompanying software.

  Physically, the computers could be located at the customer's headquarters, or even be customer's property. Typically, however, the customer remotely uses hardware belonging to the provider. In the latter case, it's a frequent practice to place customer's processes on *virtual*

machines (special programs emulating a given computer model and an operating system) — which allows e.g. enforcing **isolation** between processes from different customers running on a single physical machine (as it will now suffice to run those processes on separate virtual machines).

- **PaaS** (*platform as a service*): here, in addition, the provider offers an *environment*, meaning mostly software related to the servers, network, databases. That is typically used by developers for building web applications, abstracting from technical details of the underlying hardware.

  This layer is also traditionally considered to contain utils oriented at software engineering: frameworks for automated testing, deployment, monitoring the quality of service as perceived by end user, etc.

- **SaaS** (*software as a service*): here, the provider offers specific applications used by the users for particular purposes. An example of this is Dropbox, allowing for uploading, viewing and downloading files via a web browser (thus creating a disk space "in the cloud").

As individual private users, we often use cloud solutions even without realizing that. This happens whenever we send an e-mail, listen to music from the internet, play internet games, or watch movies.

The picture becomes somewhat different when a cloud system is used by a company. Here, delegating some tasks to the cloud provider helps in areas such as:

- data analysis (regarding e.g. customer choices, application performance, etc.);
- delivering the product to end users;
- ensuring scalability of hardware
  (upwards, but also downwards — to avoid being overcharged for unused hardware resources);
- creating and maintaining data backups;
- networking security issues;
- testing the product and its new versions;
- applying the *dev-ops* culture — that is, a culture of cooperation between the developer teams and the product teams reponsible for product health and maintenance;
- applying the *agile programming* concept, which assumes e.g. frequently confronting the product with the actual users' needs, and frequently adjusting the project.

The concept of cloud is also related to the notion of **virtualization**, though we're now considering this word in a much broader sense than just *RAM virtualization* which we saw in an earlier lecture. Generally, *virtualizing* means emulating some resources by using other ones; often, by uing appropriate software. In the case of cloud solutions, this is typically implemented by the abovementioned *virtual operating system*, which ensures isolation of a single customer's processes, and may also provide a *remote desktop*, which lets the user connect to the remote server ("in the cloud") and use the emulated desktop just as if it was a part of a standard operating system. *Network virtualization* may also take place; this means splitting the capacity of physically existing network connections into independent virtual channels.