Distributed Systems: examples and trends

Agenda

Distributed systems: examples

Challenges

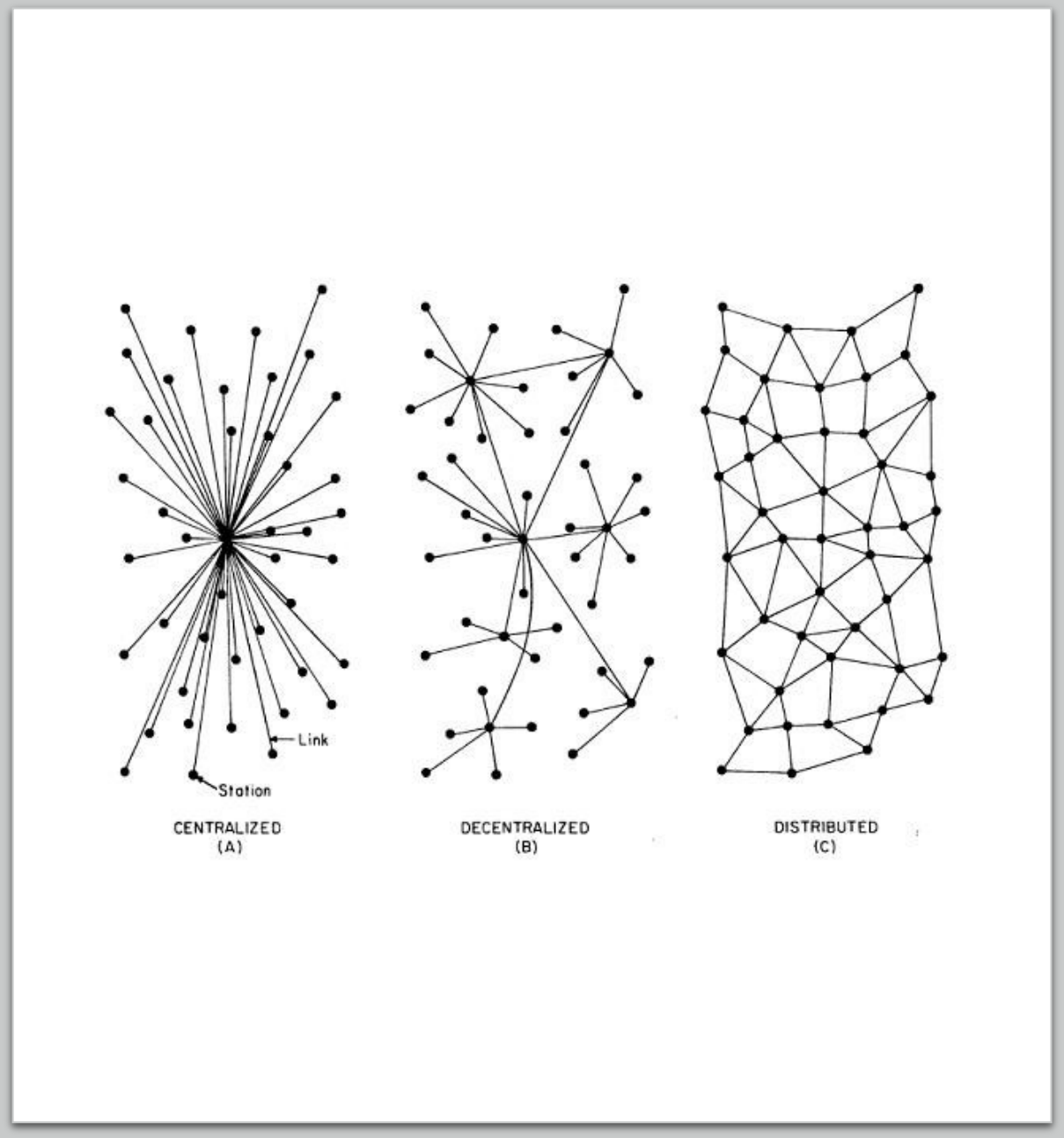
Case studies

Distributed vs decentralized

Distributed means that the processing is shared across multiple nodes, but the decisions may still be centralized and use complete system information.

Decentralized means that there is no single point where the decision is made: every node makes a decision for its own behaviour and the resulting system behaviour is the aggregate response. A key feature is that no single node will have complete system information.

Remark: A decentralized system is always (a subset of) a distributed system.



Centralized vs distributed: important points

Centralized

- Only one component with non-autonomous parts
- Component shared by users all the time
- All resources accessible
- Software runs in a single process (possibly with simulated concurrent threads)
- Single point of **control**
- Single point of **failure**

Decentralized

- Multiple autonomous components
- Components are not shared by all users
- Some resources may not be accessible
- Software runs in really concurrent processes on different processors
- Multiple points of control
- Multiple points of failure

Definition: Scalability of a system

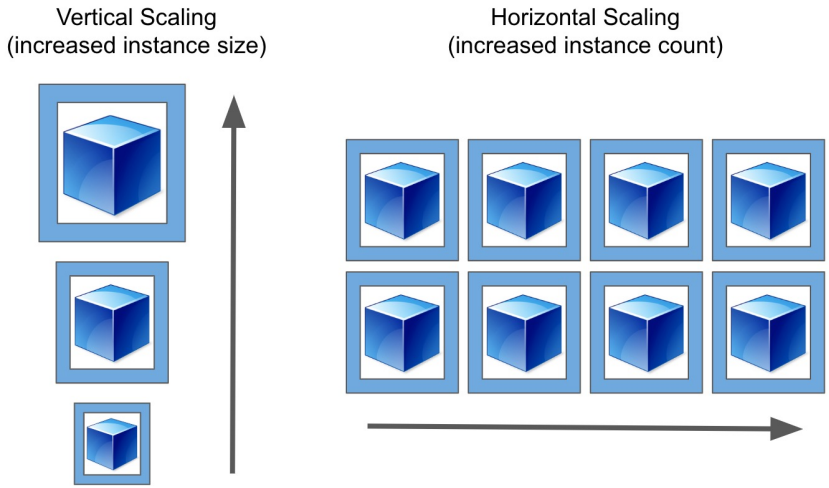
Definition: scalability

the ability of a system to handle increasing amounts of work or traffic without sacrificing performance

A scalable system can easily accommodate more users or data without becoming overloaded

Related property: elasticity

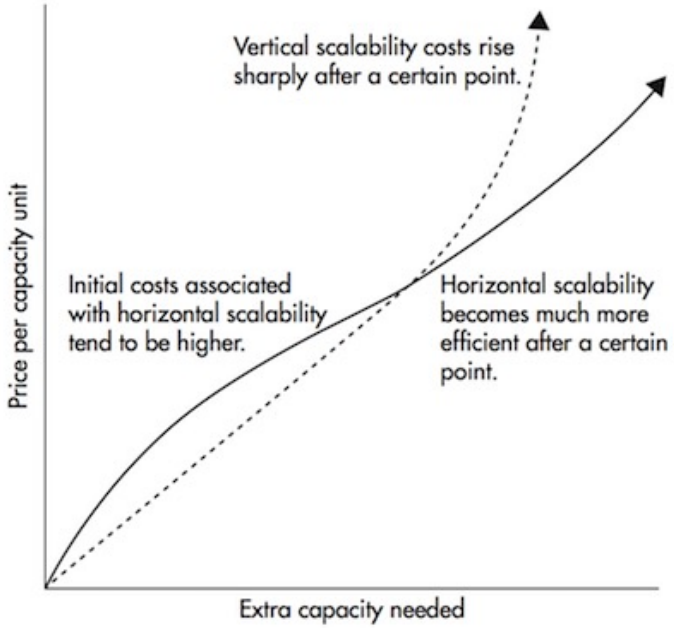
It is the ability of a cloud to automatically expand or compress the infrastructural resources on a sudden up and down in the workload, so that it can be managed efficiently. The usage of elastic resources helps a company to minimize infrastructural costs, as elasticity is automatically scaling up or down resources to meet user demands.



Why distribute a computing system?

The only way to handle more traffic with a single computer consists of upgrading the hardware: this is called **scaling vertically**.

Scaling vertically is good, but after a certain point you will see that even the best hardware is not sufficient for traffic, and impractical to host.

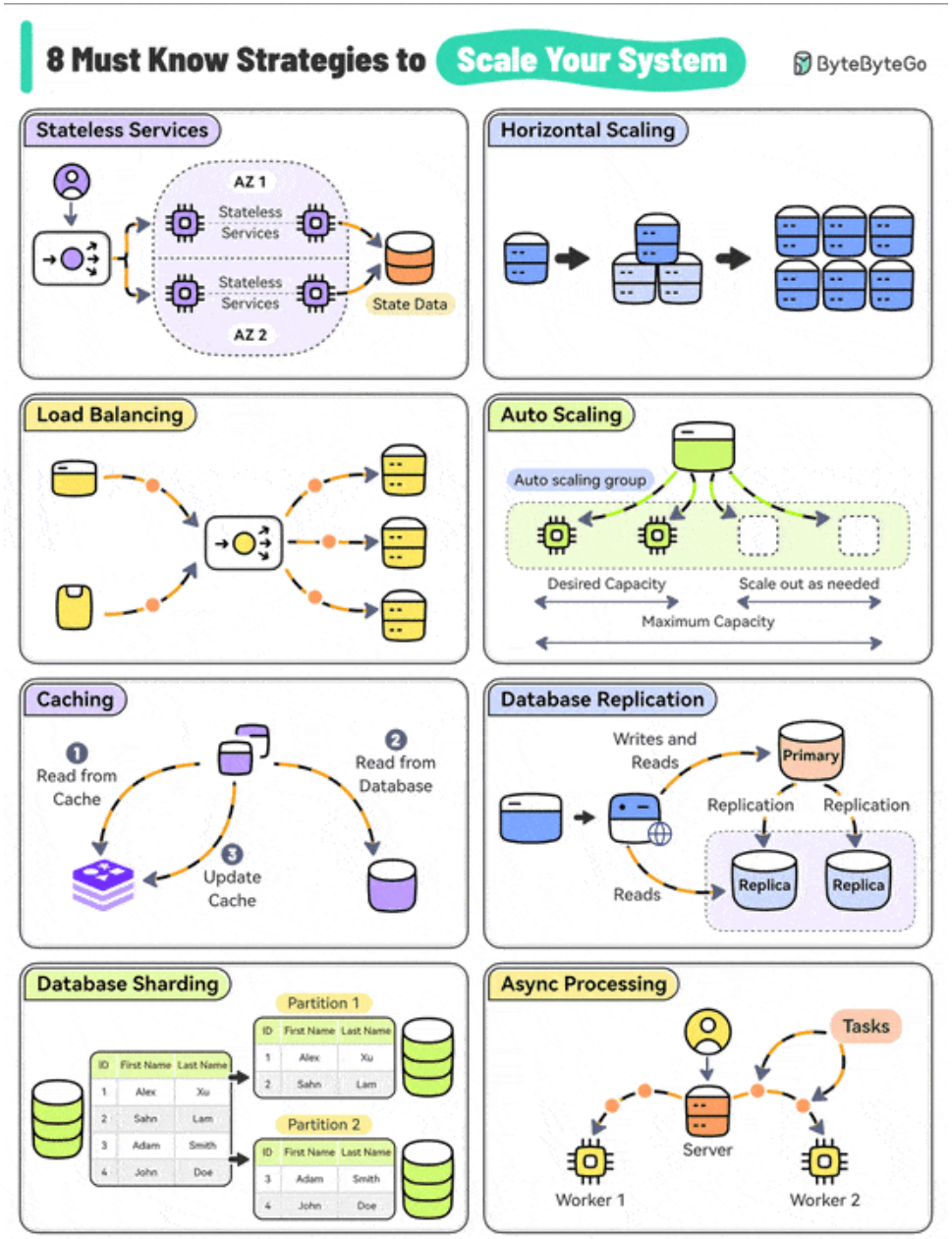


A distributed system can **scale horizontally**.

Scaling horizontally simply means adding more computers rather than upgrading the hardware of a single one.

https://medium.freecodecamp.org/a-thorough-introduction-to-distributed-systems-3b91562c9b3c

Scalability



1. Stateless Design stateless services because they don't rely on server-specific data and are easier to scale.
2. Horizontal Scaling Add more servers so that the workload can be shared.
3. Load Balancing Use a load balancer to distribute incoming requests evenly across multiple servers.
4. Auto Scaling Implement auto-scaling policies to adjust resources based on real-time traffic.
5. Caching Use caching to reduce the load on the database and handle repetitive requests at scale.
6. Database Replication Replicate data across multiple nodes to scale the read operations while improving redundancy.
7. Database Sharding Distribute data across multiple instances to scale the writes as well as reads.
8. Async Processing Move time-consuming and resource-intensive tasks to background workers using async processing to scale out new requests.

Which other strategies have you used?

Definition: Availability of a system

Availability:

the percentage of time a system is up in a timeframe

Availability class	%	Downtime per year	Downtime per month	Downtime per day
Two nines	99%	3.65 days	7.31 hours	14.40 minutes
Three nines	99.9%	8.77 hours	48.83 minutes	1.44 minutes
Five nines	99.999%	5.26 minutes	26.30 seconds	864 milliseconds
Seven nines	99.99999%	3.16 seconds	262.98 milliseconds	8.64 milliseconds

Definition: resiliency

the ability of a system to recover from failures or disruptions.

The key strategies to obtain resilient systems:

- Redundancy
- Fault tolerance
- Monitoring and testing
- Disaster recovery

Distributed systems in enterprises

Large businesses and enterprises are decentralized, with multiple users and services

Distributed computing moves information and services closer to users who need them

Personal and mobile client computers connected to network servers are less expensive and more available and resilient than mainframe computers

However, the total cost of ownership is still expensive when there are many many servers

Solution: cloud computing (centralization again!)

See https://queue.acm.org/detail.cfm?id=1394131

What is a distributed system

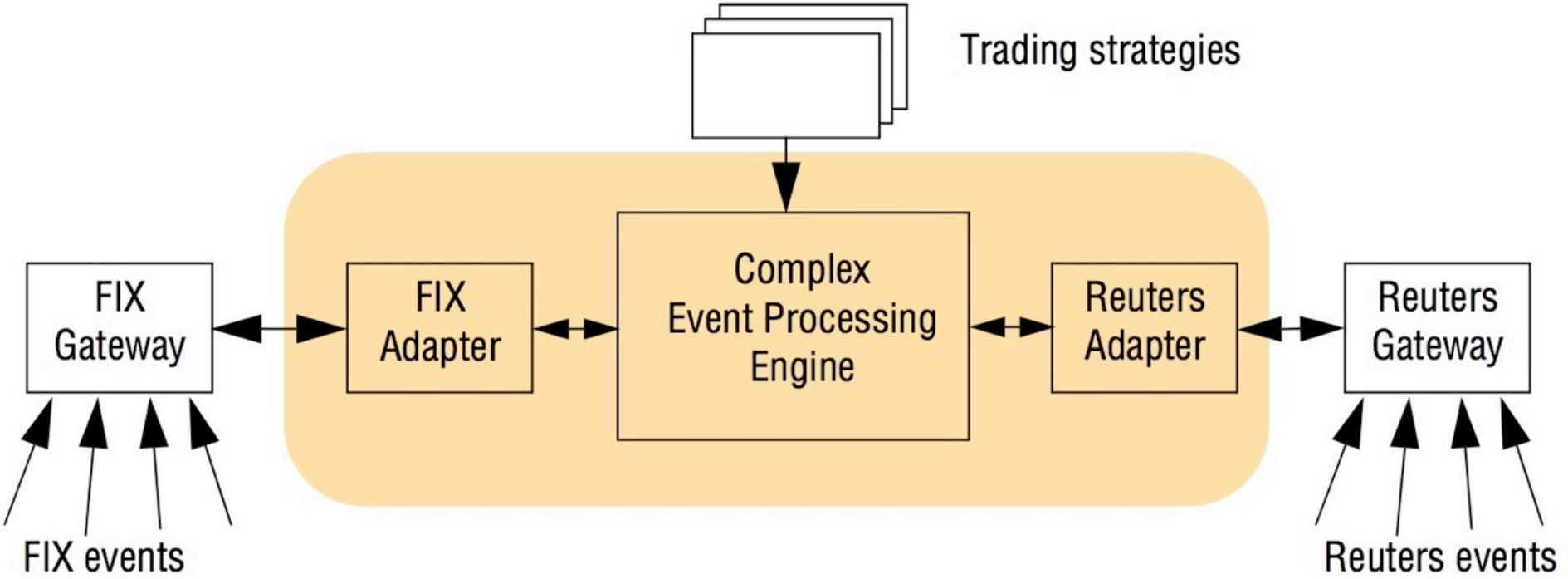
A distributed system is
a collection of autonomous computing elements
that appears to its users
as a single coherent system

(note that this is a definition emphasizing transparency!)

Single coherent system: features

- **Single system view**, behaving according to the expectations of users, that can be ubiquitous
- The collection of nodes **as a whole** operates the same, no matter where, when, and how user-system interactions take place
- Data can be replicated: **distribution transparency**
- **Middleware** to support programming and deploying distributed applications

An example financial trading system



Important non functional property: **simultaneity** (very difficult)

See https://queue.acm.org/detail.cfm?id=2745385

Collection of autonomous computing elements

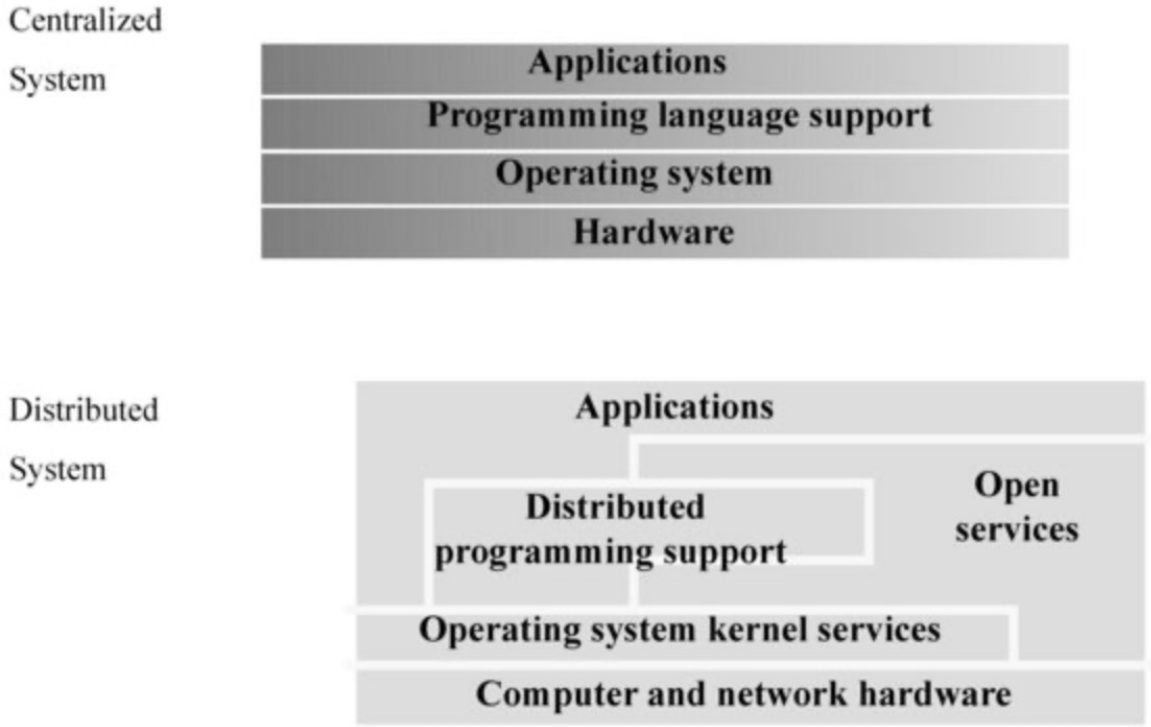
- In a distributed system nodes act independently from each other
- They are programmed (usually by different people, with different technlogies) by exchanging messages
- **IMPORTANT:** Each node has its own notion of time (the system has **no global clock**)
- **IMPORTANT**: each node can **fail** independently from others

Distributed systems vs networks of computers

- A computer network is an interconnected collection of autonomous computers able to exchange data.
- A computer network usually requires users to explicitly login on one machine, explicitly submit jobs remotely, explicitly move files/data around the network.
- In a distributed system the existence of multiple autonomous computers is transparent to the user.
- The middleware automatically allocates jobs to processors, moves files among various computers without explicit user intervention

The software structure of distributed systems

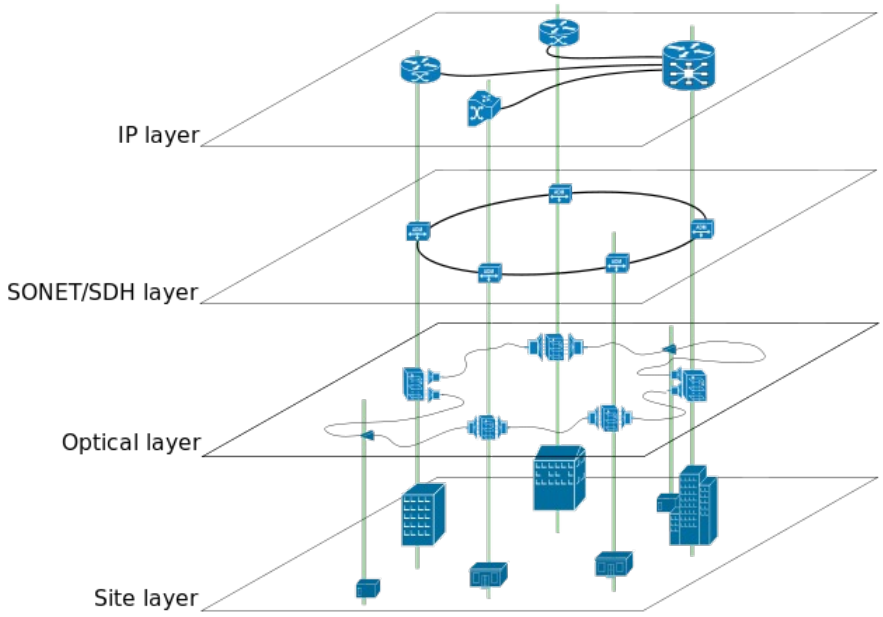
- Network protocols
- IPC mechanisms
- Overlay networks
- Middleware
- API and services

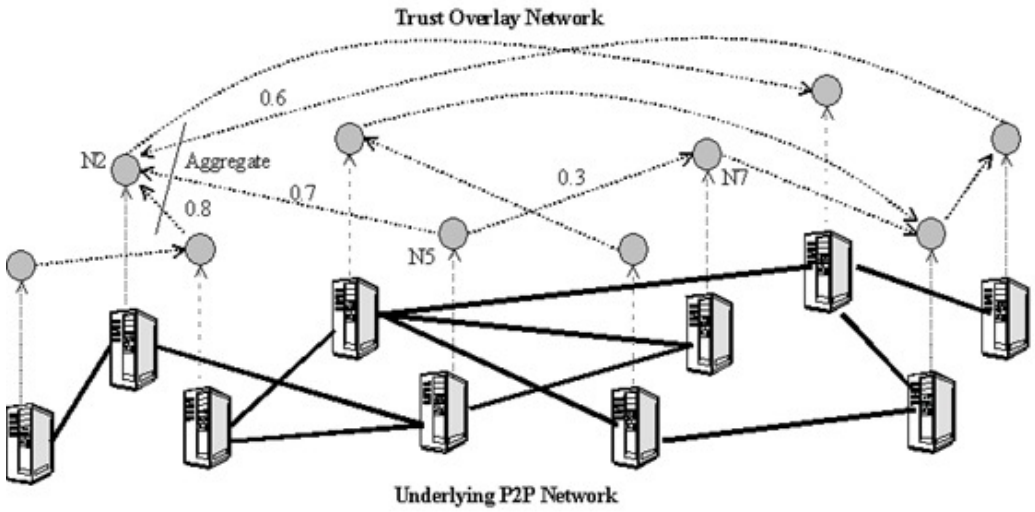


Concept: Overlay network

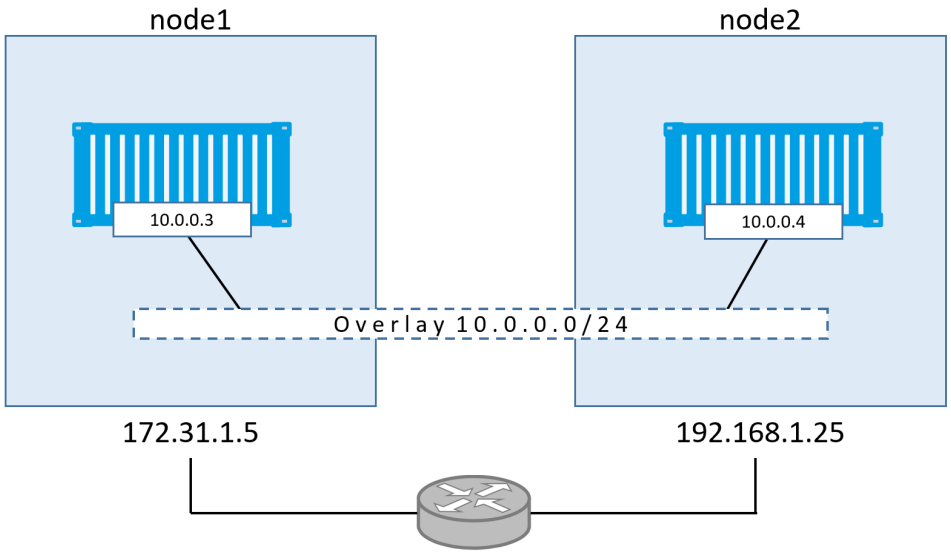
A distributed system is often organized as an overlay network, that is a (software) network built on top of another network

It is a method of using software to create layers of network abstractions to be used to run multiple separate, virtualized network layers on top of the physical network, often providing security benefits.





Overlay networks



Each node in an overlay network communicates only with other nodes in the same network, its *neighbors*.

The set of neighbors may be dynamic, or may even be known only implicitly (i.e., it requires a lookup).

Well-known example of overlay network:
peer-to-peer systems.

Overlay network types

- **Structured**: each node has a well-defined set of neighbors with whom it can communicate (tree, ring).
- **Unstructured**: each node has references to randomly selected subset of the nodes from the system

http://blog.nigelpoulton.com/demystifying-docker-overlay-networking/

Scalability

- Scalability is the capability of a system to handle a growing amount of work
- For example, a system is considered scalable if it is capable of increasing its total output under an increased load when resources (hardware) are added.
- A system whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.
- An algorithm, design, networking protocol, program, or other system is said *to scale* if it is suitably efficient and practical when applied to large situations (e.g. a large input data set, a large number of outputs or users, or a large number of participating nodes in the case of a distributed system).
- If the system fails when a quantity increases, it does not scale.

Scaling

- Horizontal scaling: you increase the number of machines
- Vertical scaling: you increase the performance of your system by increasing the performance of one machine

- Scaling is limited by Amdhal's law

The Amdhal law

The Amdhal law is a formula which gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.
It is often used in distributed computing to predict the theoretical speedup when using multiple processors.

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

where:

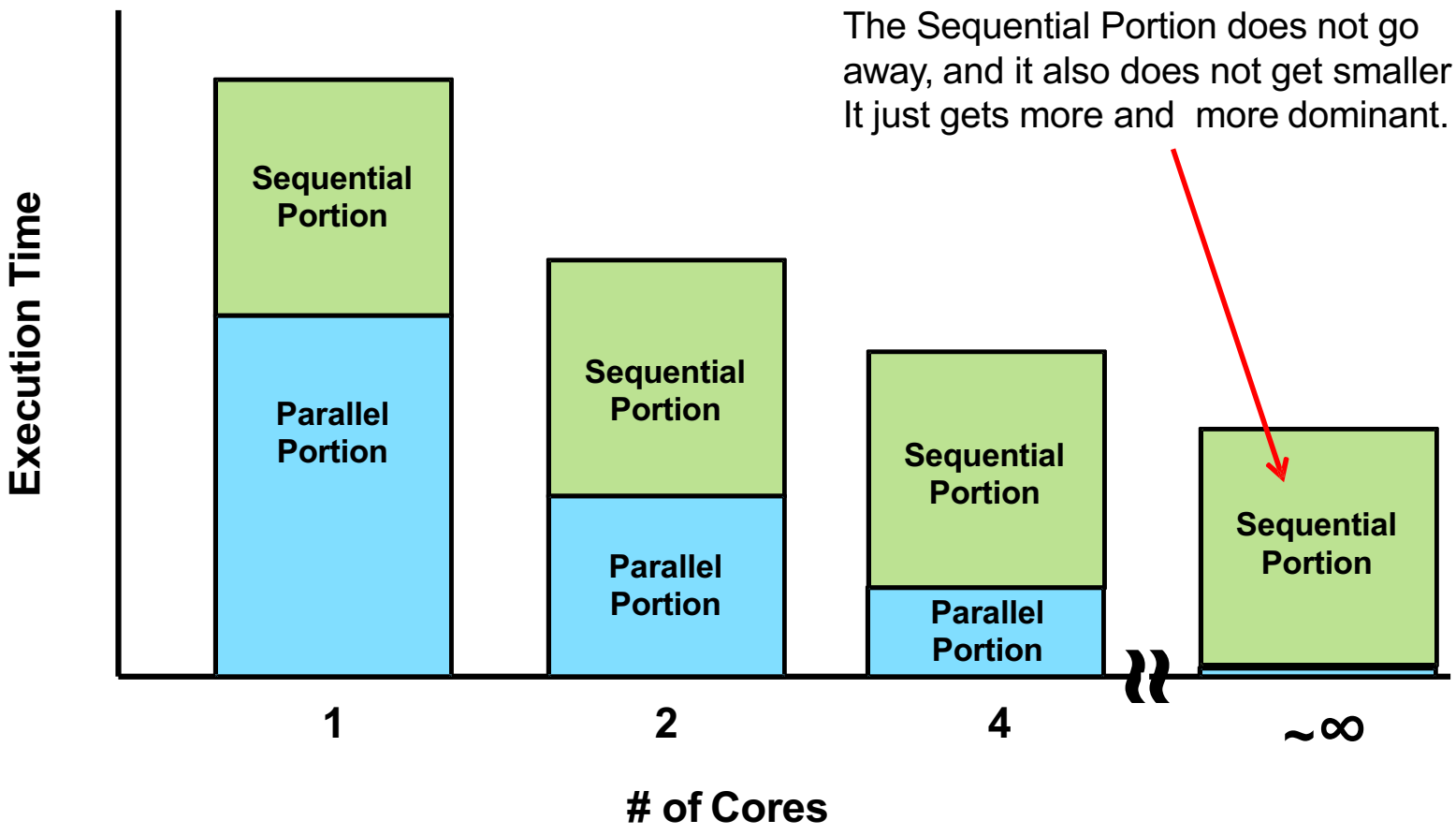
- S_{latency} is the theoretical improvement of the execution time (*latency*) of the complete task
- *s* is the improvement in execution (*speedup*) of the part of the task that benefits from the improved system resources.
- *p* is the proportion of the execution time that the part benefiting from improved resources actually occupies.

Example (source: Wikipedia, article on Amdhal law)

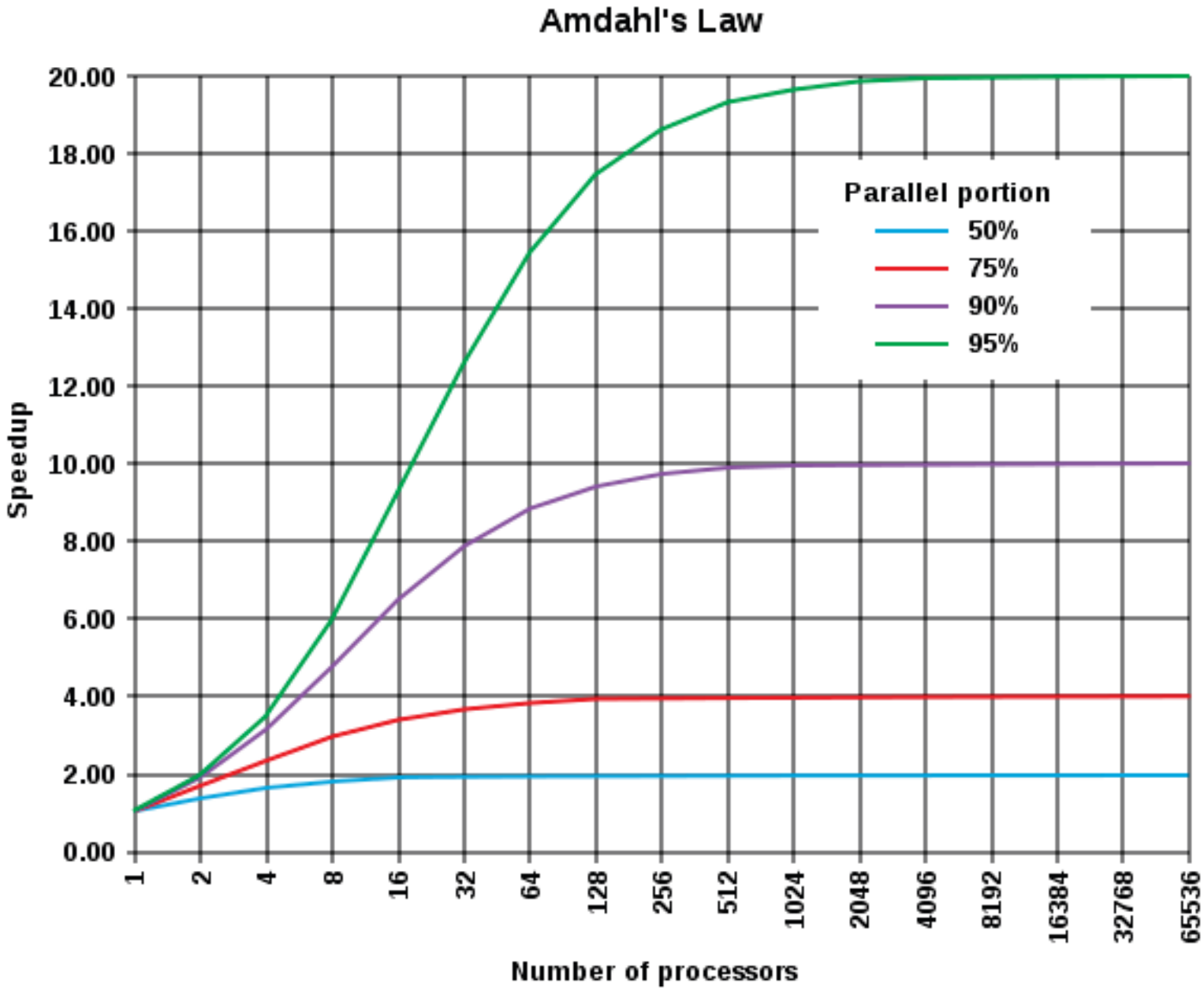
If a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours ($p = 0.95$) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour.

Hence, the theoretical speedup is limited to at most 20 times ($1/(1 - p) = 20$). For this reason, computing with many processors is useful only for highly parallelizable programs

A Visual Explanation of Amdahl's Law



The Amdhal law



Consequences of the Amdhal's law

- An implication of Amdahl's law is that to speedup real applications which have both serial and parallel portions, heterogeneous computing techniques are required.
- For example, a CPU-GPU heterogeneous processor may provide higher performance and energy efficiency than a CPU-only or GPU-only processor
- Amdahl's law applies only to the cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, Gustafson's law gives a less pessimistic and more realistic assessment of the parallel performance.

Challenges of distributed systems

Heterogeneity of computing resources

Openness

Security

Scalability

Failure handling

Quality of Service

Transparency

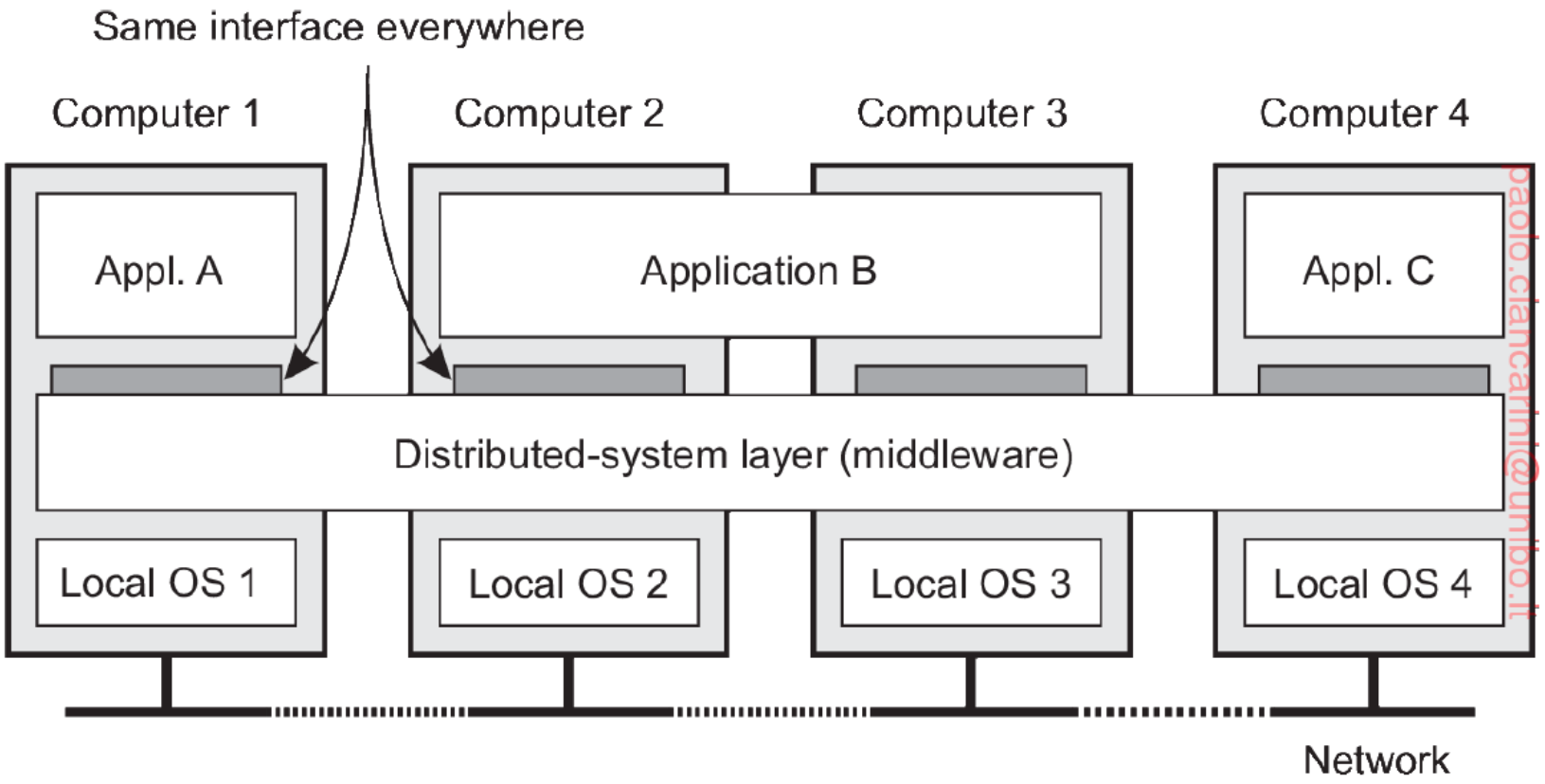
Absence of absolute time

Challenge: Heterogeneity of resources

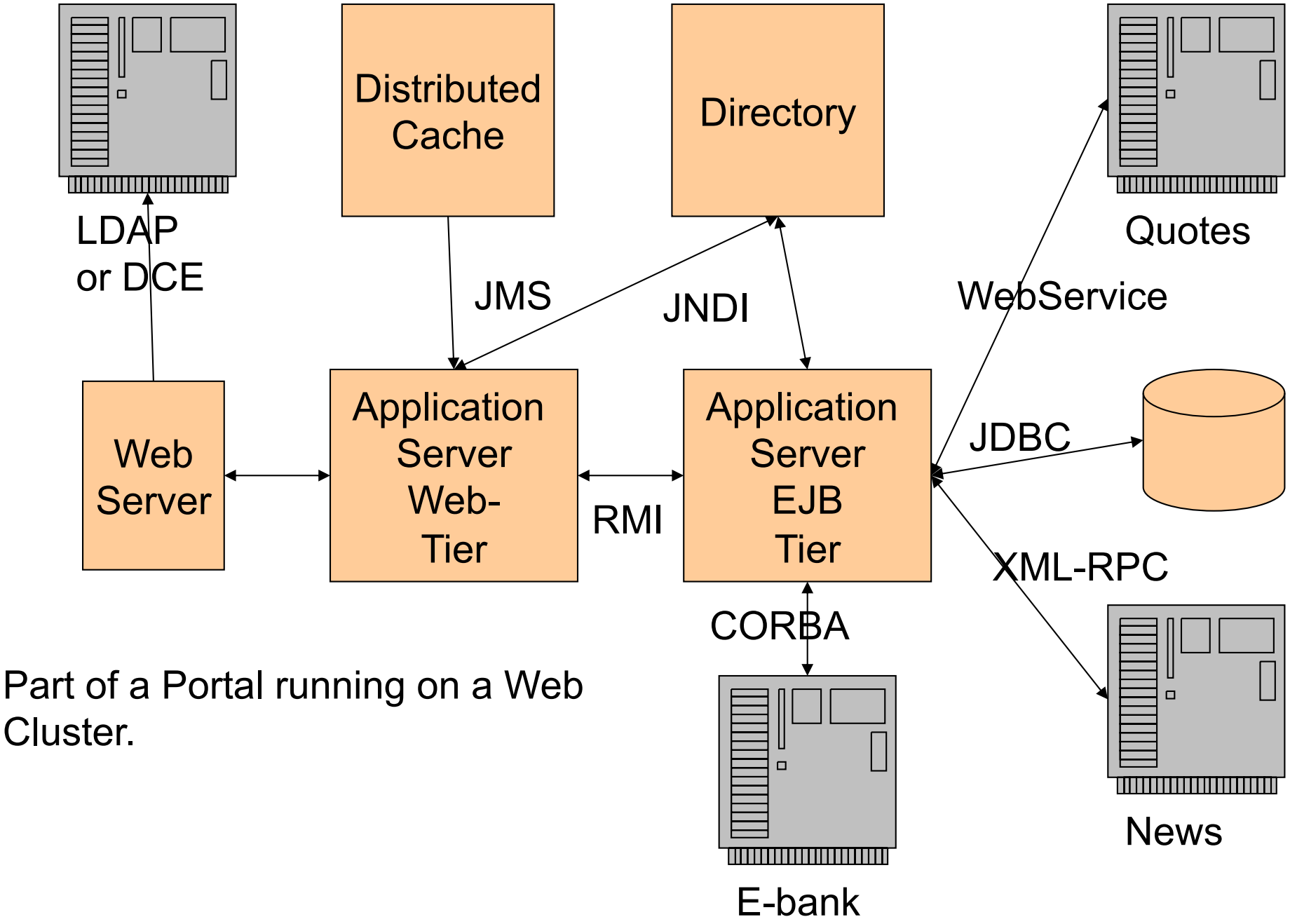
- Networks (both physical media and protocols)
- Computer hardware;
- Operating systems;
- Programming languages;
- Implementations by different developers;
- Use cases: eg mobility vs home computing
- Systems of systems

Solution: Using middleware

1. Communications, eg. RPC
2. Transactions, eg. ACID
3. Service composition, eg. Web services
4. Reliability, eg. Horus



Where do we find Middleware?



Part of a Portal running on a Web Cluster.

The Transparency Dogma

Middleware should hide both topology (i.e.remoteness) and concurrency by hiding distribution behind local programming language constructs

Critique: Jim Waldo, SUN

Full transparency is impossible, and the price is too high

Challenge: Openness

- Open systems have their key API public
  (API = Application Programming Interface)
- Open systems are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open systems can be constructed from heterogeneous hardware and software, possibly from different vendors.
- The conformance of each component to the published standard (we call this interoperability) must be carefully tested and verified if the system is to work correctly.

Challenge: Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system

- Controlling the cost of physical resources
- Controlling the performance loss
- Preventing software resources running out
- Avoiding performance bottlenecks

Challenge: Failure handling

Computer systems sometimes fail.

When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation

- Detecting failures (eg. Checksum)
- Masking failures (eg. retransmit msgs, data replication)
- Tolerating failures
- Recovery from failures
- Redundant components

Network transparency

- The two most important transparencies are *access* and *location* transparency: they strongly affect the usage of distributed resources
- They are referred to together as *network transparency*
- *Network transparency* is the situation in which an operating system allows a user to access a resource (such as some data or an application program) without the user needing to know or being aware if the resource is located on the *local machine* or on a *remote machine* (i.e., a computer elsewhere on the network).

Exercise: Implement a Simple Remote File Reader with Network Transparency

Objective: Build a simple distributed application that provides **network transparency** by allowing a client to read files from a remote server as if they were local files, without the client needing to know the file is on a remote machine.

Tasks:

1. Create a server application that stores several text files in a directory.
2. The server should expose an API (via sockets or REST API) to allow clients to request the contents of a specific file.
3. Implement a client that can request and retrieve files from the server. The client should read files as if they were local (abstracting the network part from the user).
4. Ensure that the client does not need to know whether the file is local or remote, demonstrating **network transparency**.

Server:

- Accepts incoming file requests.
- Retrieves the requested file from its directory and sends it back to the client over the network.

Client:

- Requests a file by providing its filename.
- Displays the content of the file as if it was a local file.

Optional Goal:

- Handle the case where the file requested by the client is available both locally and remotely. The client should always attempt to retrieve the local file first, and if it does not exist, fall back to the remote file (improving network transparency).

Hint to the Solution:

- Network Layer: Use basic socket programming in Python (e.g., `socket` library) or implement a simple REST API (using Flask or similar) for communication between the client and the server.
- Transparency Layer: Abstract the file reading logic in the client, so that the user only provides the filename without knowing the file's location.

Absence of absolute time

- A distributed system has no global management of time
- This is both a theoretical problem and a practical challenge:
  - distributed algorithms cannot exploit any notion of centralized clock
  - distributed applications must manage time-related operations in very specific ways

Conclusions: trends

Distributed systems are always evolving.

Some trends:

- Ubiquitous computing, support user mobility
- Cloud: increase availability, resilience, elasticity
- Smart systems: IoT, autonomy
- Big data: HPC for AI applications

Conclusion: system design

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the process more manageable.

Modularization Dividing the system into smaller, manageable modules helps reduce complexity, improve maintainability, and increase reusability.

Abstraction Hiding the implementation details and showing only the essential features helps simplify complex systems and promote modularity.

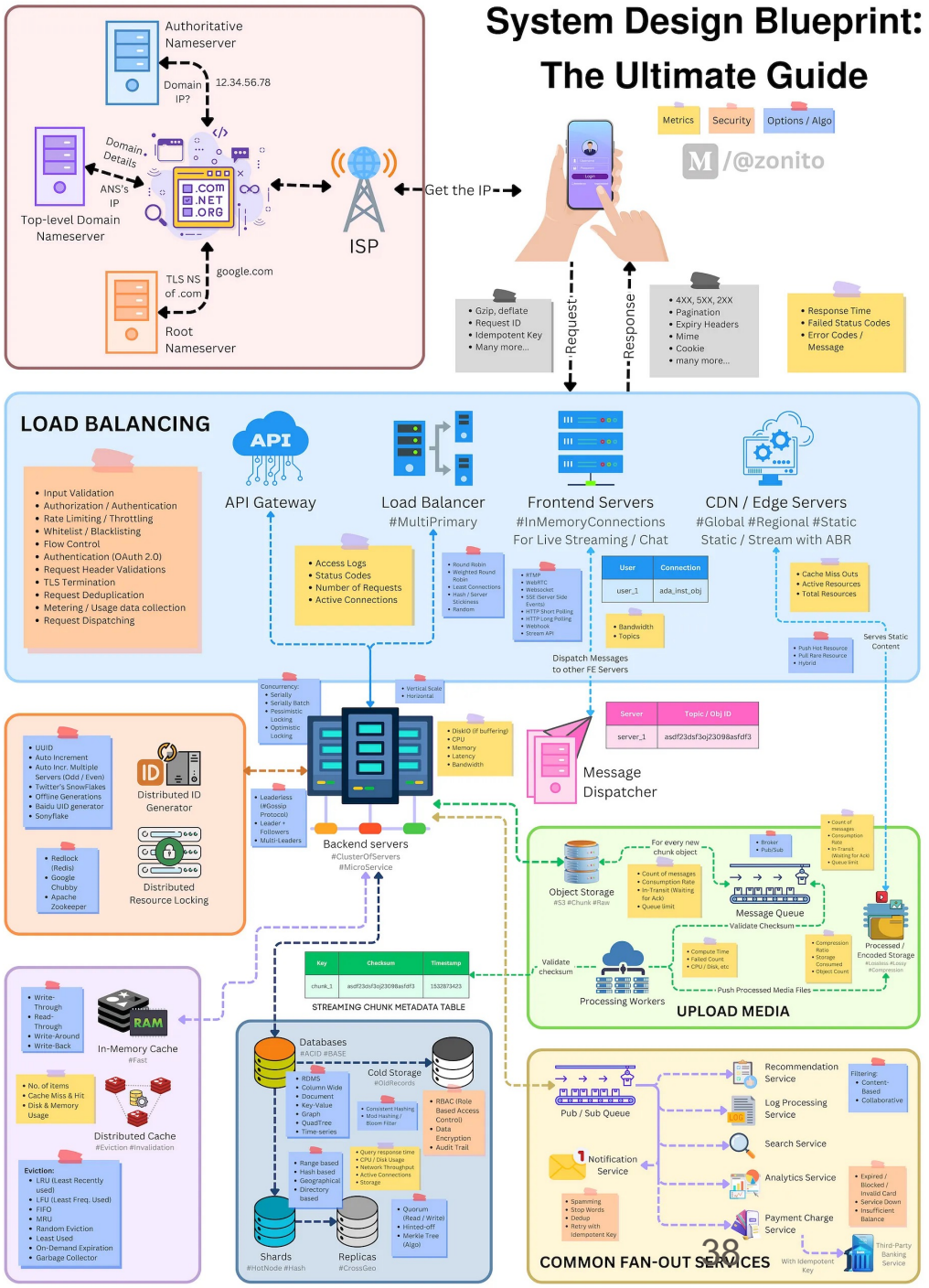
Layering Organizing the system into layers, each layer providing a specific set of functionalities promotes the separation of concerns and enhances maintainability.

Scalability Design systems to handle the increased load by adding more resources (horizontal scaling) or optimizing the system's capacity (vertical scaling).

Performance Optimizing the system's response time, throughput, and resource utilization is crucial for a successful design.

Security Ensure the system's confidentiality, integrity, and availability by implementing proper security measures and practices.

Fault Tolerance and Resilience Design systems to withstand failures and recover gracefully from errors, ensuring reliability and availability.



https://medium.com/bytebytego-system-design-alliance/system-design-blueprint-the-ultimate-guide-e27b914bf8f1

Exercise

What is meant by a scalable system?

Give examples

Exercise

- How might the clocks in two computers that are linked by a local network be synchronized without reference to an external time source?
- What factors limit the accuracy of the procedure you have described?
- How could the clocks in a large number of computers connected by the Internet be synchronized?
- Discuss the accuracy of that procedure.

Cristian's protocol

- The round trip time t to send a message and a reply between computer A and computer B is measured by repeated tests; then computer A sends its clock setting T to computer B. B sets its clock to T +t /2. The setting can be refined by repetition.
- The procedure is subject to inaccuracy because of contention for the use of the local network from other computers and delays in the processing the messages in the operating systems of A and B. For a local network, the accuracy is probably within 1 ms.
- For a large number of computers, one computer should be nominated to act as the time server and it should carry out Cristian's protocol with all of them. The protocol can be initiated by each in turn.
- Additional inaccuracies arise in the Internet because messages are delayed as they pass through switches in wider area networks.
- For a wide area network the accuracy is probably within 5-10 ms. These answers do not take into account the need for fault-tolerance