**Getting Started**

Learn what a distributed system is and why we need it.

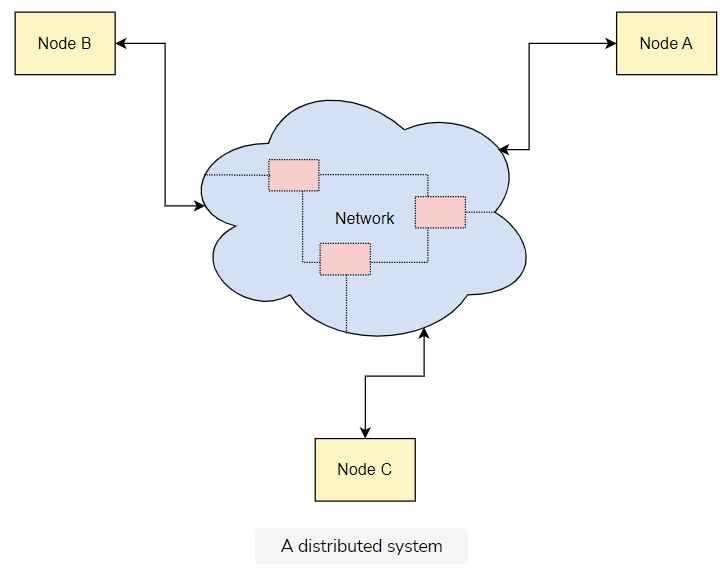
**We'll cover the following**

* [What is a distributed system?](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#What-is-a-distributed-system?)
  + [Parts of a distributed system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Parts-of-a-distributed-system)
* [Why we need a distributed system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Why-we-need-a-distributed-system)
  + [Performance](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Performance)
    - [Problem with a single computer](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Problem-with-a-single-computer)
    - [Solution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Solution)
  + [Scalability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Scalability)
    - [Problem with a single computer](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Problem-with-a-single-computer)
    - [Solution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Solution)
  + [Availability](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Availability)
    - [Problem with a single computer](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Problem-with-a-single-computer)
    - [Solution](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5814258537594880#Solution)

**What is a distributed system?**

According to Coulouris et al., “A **distributed system** is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another.”

The components of this system can be thought of as software programs that run on physical hardware, such as computers. These components take many forms; e.g., they can be web servers, routers, web browsers, etc. To keep a generic view, we assume that each program runs on a separate machine. We refer to each of these machines as a **node**.



The above illustration shows that the network either consists of direct connections between the distributed system components, or more components that form the backbone of the network (e.g., if communication is done through the Internet).

While the generic node view is helpful to understand the diagram above, sometimes real-life examples of how the nodes work may be more helpful… In these cases, we explain the role of each node in the system in detail.

### Parts of a distributed system

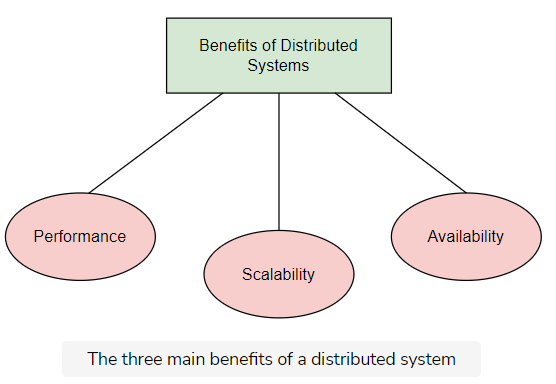
There are two categories of the central parts that help distributed systems function:

* The various parts that compose a distributed system: These are located remotely and are separated by a network
* The network that separates the various parts of a distributed system: It acts as a communication mechanism that lets them exchange messages.

We will see the individual parts in detail later in this course.

## Why we need a distributed system

There are three main benefits of distributed systems, as shown in the illustration below.



Let’s explain each one separately.

### Performance

According to Mohan et al., “**Performance** is the degree to which a software system or component meets its objectives for timeliness.”

#### Problem with a single computer

The physical constraints of its hardware impose certain limits on the performance of a single computer. Moreover, it is extremely expensive to improve a single computer’s performance after a certain point.

#### Solution

We can achieve the same performance with two or more low-spec computers as with a single, high-end computer. So, distributed systems allow us to achieve better performance at a lower cost.

Note that better performance can translate to different things depending on the context, such as lower latency per request, higher throughput, etc.

### Scalability

According to Bondi et al., “Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.”

#### Problem with a single computer

Data storage and processing are responsible for most of the value that software systems impart in the real world. As a system’s customer base grows, the system needs to handle more traffic and store larger amounts of data. However, a system that comprises a single computer can only scale up to a certain point, as explained earlier.

#### Solution

If we build a distributed system, we can split and store the data in multiple computers, and distribute the processing work.

**Vertical scaling** refers to the approach of scaling a system by adding resources (memory, CPU, disk, etc.) to a single node. Meanwhile, **horizontal scaling** refers to the approach of scaling by adding more nodes to the system.

As a result of this, we can scale our systems to sizes that we could not imagine with a single computer system.

### Availability

In the context of software systems, **availability** is the probability of a system to work as required, when required, during a mission.

#### Problem with a single computer

Nowadays, most online services need to operate all the time (also known as “24/7 service”), which is a huge challenge. When a service states that it has five-nine availability, it usually operates 99.999% of the time. This implies that it can be down for only 5 minutes at most per year to satisfy this guarantee.

If we consider how unreliable hardware can be, we can easily understand how big an undertaking this is. Of course, it would be infeasible to provide this kind of guarantee with a single computer.

#### Solution

**Redundancy** is one of the widely used mechanisms to achieve higher availability. It refers to storing data into multiple, redundant computers. So, when one computer fails, we can efficiently switch to another one. This way, we’ll prevent our customers from experiencing this failure.

Given that data are stored now in multiple computers, we end up with a distributed system!

If we leverage a distributed system, we get all of the above benefits. However, as we will see later on, there is tension between them and several other properties. So, in most cases, we have to make a trade-off. To do this, we must understand the basic constraints and limitations of distributed systems. The first part of this course will help us with this.

**Fallacies of Distributed Computing**

Let's see what false assumptions developers make while developing software for the distributed systems.

**We'll cover the following**

* [The difference in developing software for distributed systems](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#The-difference-in-developing-software-for-distributed-systems)
* [Fallacies](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#Fallacies)
  + [The network is reliable](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#The-network-is-reliable)
  + [Latency is zero](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#Latency-is-zero)
  + [Bandwidth is infinite](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#Bandwidth-is-infinite)
  + [The network is secure](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#The-network-is-secure)
  + [Topology doesn’t change](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#Topology-doesn%E2%80%99t-change)
  + [Transport cost is zero](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#Transport-cost-is-zero)
* [The global clock fallacy](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5757325994622976#The-global-clock-fallacy)

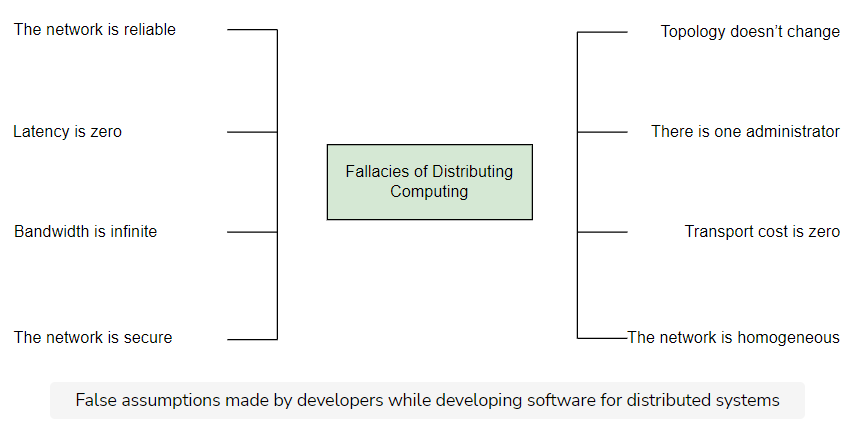
**The difference in developing software for distributed systems**

Distributed systems are subject to many more constraints than software systems that run on a single computer. As a result, the development of software for distributed systems is also very different. However, those who are new to distributed systems make assumptions based on their experience with software development for systems that run on a single computer. Of course, this creates a lot of problems down the road in the systems they build.

To eliminate this confusion and help people build better systems, L Peter Deutsch and others at Sun Microsystems created a collection of these false assumptions. These are the [fallacies of distributed computing](https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing).

**Fallacies**

There are eight such fallacies of distributed computing. The following illustration lists them.



As you progress through the course, you’ll gain a deeper understanding of why these statements are fallacious.

However, we’ll give you a sneak preview here by quickly going over them and explaining where they fall short.

### The network is reliable

The abstractions developers learn from various technologies and protocols often enforce this common fallacy. As we will see in a later chapter, networking protocols like [TCP](https://www.educative.io/collection/page/10370001/4891237377638400/5278678164963328#tcp) can make us believe that the network is reliable and never fails. However, this is just an illusion with significant repercussions. Also, we build network connections on top of hardware that will also fail at some point. Hence, we should design our systems accordingly.

### Latency is zero

Libraries that attempt to model remote procedure calls as local calls, such as [gRPC](https://grpc.io/" \t "_blank) or [Thrift](https://thrift.apache.org/), enforce this assumption. We should always remember that there’s a large difference (from milliseconds to nanoseconds) in latency between a call to a remote system and that to local memory access. This gets even worse when we consider calls between data centers on different continents. Thus, this is another thing to keep in mind when deciding how to geo-distribute our system.

### Bandwidth is infinite

This fallacy is weaker nowadays. This is because the bandwidth we can achieve has significantly improved in the last few decades. For instance, we can now build high-bandwidth connections in our own data centers. However, this does not mean we can use all of it if our traffic needs to cross the Internet. This is important to consider when we make decisions about our distributed system’s topology, and when requests travel through the Internet.

### The network is secure

This fallacy shows that the wider network used by two nodes to communicate is not necessarily under their control. Thus, we should consider it insecure.



The course dedicates a portion to security, where it explains the various techniques we can use to securely utilize an insecure network.

### Topology doesn’t change

Network also comprises many different parts that different organizations may manage with different hardware. Moreover, failures in some parts of this network may require us to change its topology to keep it functional. This also highlights the other two fallacies i.e **there is one administrator** and **the network is homogeneous**

### Transport cost is zero

The transportation of data between two points incurs financial costs. We should factor this in when we build a distributed system.

## The global clock fallacy

There’s one fallacy that’s not a part of the above set, but still often causes confusion amongst people new to distributed systems. . If we follow the same style as above, we can phrase this fallacy as:

“Distributed systems have a global clock, which we can use to identify when events happen.”

This assumption is quite deceiving since it’s somewhat intuitive and holds true even in non-distributed systems. For instance, an application that runs on a single computer can use the computer’s local clock to decide when events happen, and in what order. However, this is not true in a distributed system, where every node in the system has its own local clock that runs at a unique rate.

While there are ways to keep the clocks in sync, some are very expensive and don’t completely eliminate these differences. [Physical laws](https://en.wikipedia.org/wiki/Time_dilation) also bind this limitation. An example of this is the TrueTime API built by Google, which exposes the clock uncertainty explicitly as a first-class citizen.

However, as we’ll see in the upcoming lessons that discuss cause and effects, there are other ways to reason about time using logical clocks.

# Difficulties Designing Distributed Systems

Let's see what makes distributed systems hard to design.

**We'll cover the following**

* [Why distributed systems are hard to design](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4633409322483712#Why-distributed-systems-are-hard-to-design)
  + [Properties that make distributed systems challenging](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4633409322483712#Properties-that-make-distributed-systems-challenging)
    - [Network asynchrony](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4633409322483712#Network-asynchrony)
    - [Partial failures](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4633409322483712#Partial-failures)
    - [Concurrency](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4633409322483712#Concurrency)

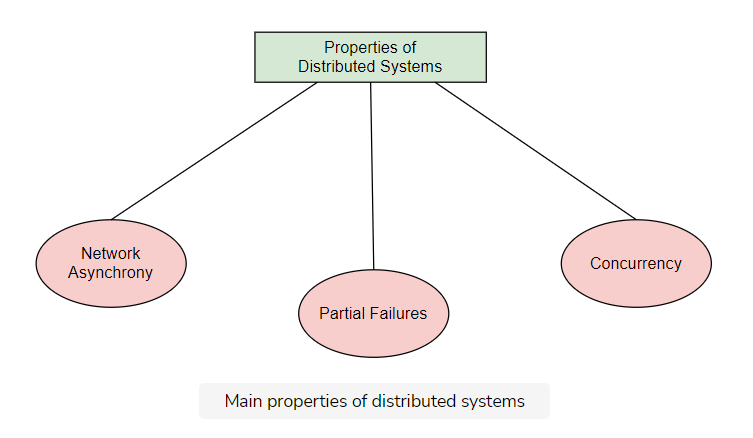
## Why distributed systems are hard to design

In general, distributed systems are hard to design, build, and reason about. This increases the risk of error.

It’s worth questioning this: why are distributed systems so hard to design? The answer to this question will help us eliminate our blind spots, and provide guidance on some aspects we should pay attention to.

### Properties that make distributed systems challenging

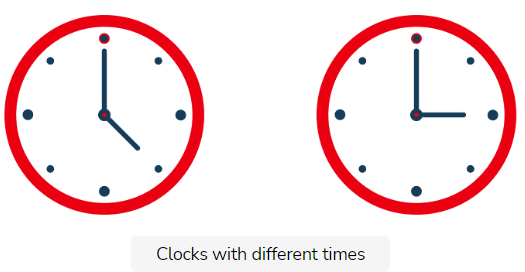
The following illustration shows the main properties that make distributed systems challenging to reason about.



Let’s look at each property.

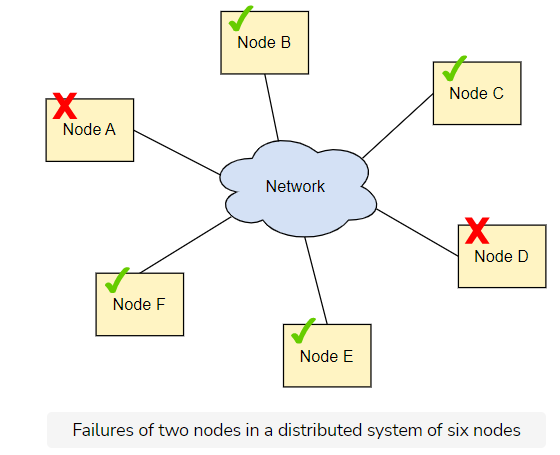
#### Network asynchrony

**Network asynchrony** is a property of communication networks that cannot provide strong guarantees around delivering events, e.g., a maximum amount of time a message requires for delivery. This can create a lot of counter-intuitive behaviors that are not present in non-distributed systems. This contrasts to memory operations that provide much [stricter guarantees](https://en.wikipedia.org/wiki/CAS_latency). For instance, messages might take extremely long to deliver in a distributed system. They may even deliver out of order—or not at all.



#### Partial failures

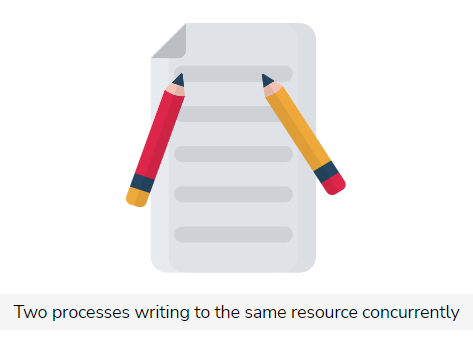
**Partial failures** are the cases where only some components of a distributed system fail. This behavior can contrast with certain kinds of applications a single server deploys. These applications work under the assumption that either everything is working fine, or there has been a server crash. It introduces significant complexity when it requires atomicity across components in a distributed system. Thus, we must ensure that we either apply an operation to all the nodes of a system, or to none of them.



The chapter about [achieving atomicity](https://www.educative.io/collection/page/10370001/4891237377638400/6261976206934016) analyses this problem.

#### Concurrency

**Concurrency** is the execution of multiple computations at the same time, and potentially on the same piece of data. These computations interleave with each other. This introduces additional complexity since these computations can interfere with each other and create unexpected behaviors. This is, again, in contrast to simplistic applications with no concurrency, where the program runs in the order the sequence of commands in the source code defined.



The chapter that talks about [isolation](https://www.educative.io/collection/page/10370001/4891237377638400/4543682114486272) explains the various types of problematic behaviors that arise from concurrency.

Network asynchrony, partial failures, and concurrency are the major contributors to complexity in the field of distributed systems. So, we should keep them in mind when we build distributed systems in real life. Doing so would help us anticipate edge cases and handle them appropriately.

# Measures of Correctness in Distributed Systems

Let's see how to measure the correctness of a distributed system.

**We'll cover the following**

* [Correctness](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6510300937584640#Correctness)
* [Measures of Correctness](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6510300937584640#Measures-of-Correctness)
  + [Safety](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6510300937584640#Safety)
  + [Liveness](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6510300937584640#Liveness)
* [Example of a correct system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6510300937584640#Example-of-a-correct-system)

## Correctness

We can define the correctness of a system in terms of the properties it must satisfy.

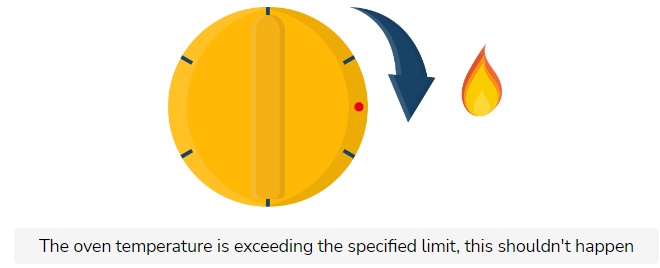
## Measures of Correctness

The correctness measures for distributed systems are the two properties they must satisfy. These are the following:

* Safety property
* Liveness property

### Safety

A safety property defines something that must never happen in a correct system.



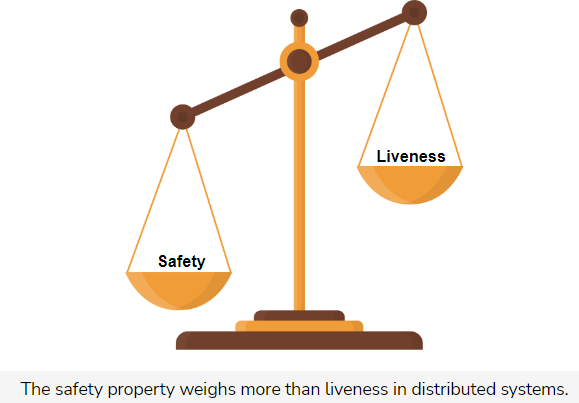
### Liveness

A liveness property defines something that must eventually happen in a correct system.

## Example of a correct system

If we consider the correct properties of an oven, we can say that “the oven not exceeding a maximum temperature threshold” is a safety property. The property of “the oven eventually reaching the temperature we specified via the button” is a liveness property.

Similar to this example, it’s usually more important in distributed systems to ensure the system satisfies the safety property than the liveness one.



Throughout this course, it will become clear that there is an inherent tension between safety and liveness properties. Actually, as we will see later in this course, there are some problems that make it physically impossible to satisfy both kinds of properties. So, we need to compromise some liveness properties to maintain safety.

# System Models

Let's see the distributed system models.

**We'll cover the following**

* [Nature of real-life distributed systems](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Nature-of-real-life-distributed-systems)
* [Making a generic model](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Making-a-generic-model)
  + [Properties each system follows](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Properties-each-system-follows)
* [Categories of distributed systems](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Categories-of-distributed-systems)
  + [Synchronous system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Synchronous-system)
  + [Asynchronous system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6091640678907904#Asynchronous-system)

## Nature of real-life distributed systems

Real-life distributed systems can differ drastically in many dimensions. These differences depend on factors like the network that deploys them, the hardware they run on, etc.

Thus, we need a common framework to solve problems generically. This way, we don’t need to repeat the reasoning for the different variations of these systems.

## Making a generic model

To create a model of a distributed system, we must define several properties it must satisfy. If we prove an algorithm is correct for this model, we can be sure that it’ll also be correct for all the systems that satisfy these properties.

### Properties each system follows

The main important properties in a distributed system concern the following:

* How the nodes of a distributed system interact with each other
* How the nodes of a distributed system can fail

## Categories of distributed systems

There are two main categories of distributed systems that depend on the nature of communication:

1. Synchronous systems
2. Asynchronous systems

### Synchronous system

A **synchronous system** is one where each node has an accurate clock, and there is a known upper bound on the message transmission delay and processing time. As a result, the execution is split into rounds. This way, every node sends a message to another node, the messages deliver, and every node computes based on the messages it receives. During this, all nodes run in lock-step.

### Asynchronous system

An **asynchronous system** is one where there is no fixed upper bound on how long it takes for a node to deliver a message, or how much time elapses between consecutive steps of a node. The system nodes do not have a common notion of time and, thus, run at independent rates.

The [previous lesson](https://www.educative.io/collection/page/10370001/4891237377638400/4624746937843712) discussed the challenges arising from network asynchrony.

So, it should be clear by now that the synchronous model is much easier to describe, program, and reason about. However, the asynchronous model is closer to real-life distributed systems, such as the Internet, where we cannot control all the components they involve. Also, there are minimal guarantees on the time it takes to send a message between two places.

As a result, most of the algorithms we look at in this course assume an asynchronous system model.

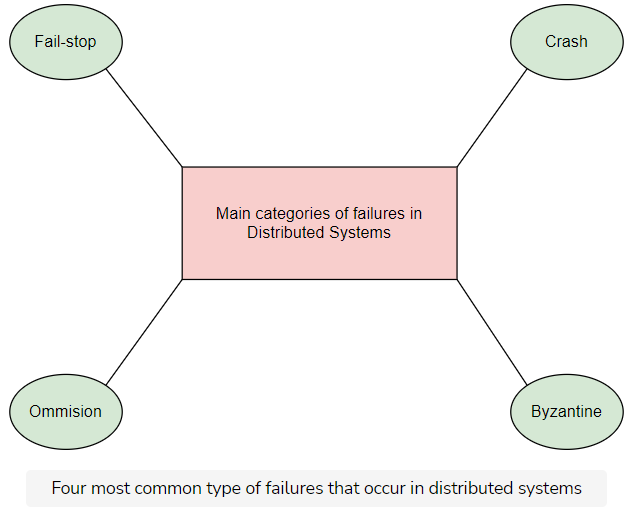
**Types of Failures**

Let's see the four basic types of failures.

**We'll cover the following**

* [Fail-stop](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5409778985861120#Fail-stop)
* [Crash](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5409778985861120#Crash)
* [Omission](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5409778985861120#Omission)
* [Byzantine](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/5409778985861120#Byzantine)

There are several different types of failures. The following illustration shows the most basic categories.



Let’s see these failures one by one.

## Fail-stop

A node halts and remains halted permanently. Other nodes can detect that the node has failed (i.e., by communicating with it).

## Crash

A node halts, but silently. So, other nodes may not be able to detect this state. They can only assume its failure by being able to communicate with it.

## Omission

A node fails to respond to incoming requests.

## Byzantine

A node exhibits arbitrary behavior: it may transmit arbitrary messages at arbitrary times, take incorrect steps, or stop.

Byzantine failures occur when a node does not behave according to its specific protocol or algorithm. This usually happens when a malicious actor or a software bug compromises the node.

To cope with these failures, we need complex solutions. However, most companies deploy distributed systems in environments that they assume to be private and secure.

Fail-stop failures are the simplest and the most convenient ones from the perspective of someone that builds distributed systems. However, they are not very realistic. This is because there are many cases in real-life systems where it’s not easy for us to identify whether another node crashes or not.

Most of the algorithms we analyze in this course work under the assumption of crash failures.

**The Tale of Exactly-Once Semantics**

Know the story behind the exactly-once semantics.

**We'll cover the following**

* [Multiple deliveries of a message](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Multiple-deliveries-of-a-message)
  + [Example consequence](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Example-consequence)
* [Avoiding multiple deliveries of a message](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Avoiding-multiple-deliveries-of-a-message)
  + [Idempotent operations approach](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Idempotent-operations-approach)
    - [Example of idempotent operation](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Example-of-idempotent-operation)
    - [Example of non-idempotent operation](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Example-of-non-idempotent-operation)
  + [De-duplication approach](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#De-duplication-approach)
    - [Example](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Example)
* [Difference between delivery and processing](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Difference-between-delivery-and-processing)
* [Other delivery semantics](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/4791081028026368#Other-delivery-semantics)

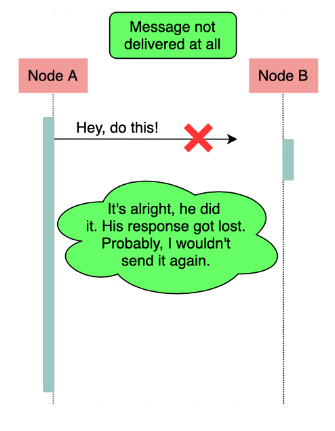
**Multiple deliveries of a message**

Various nodes of a distributed system communicate with each other through the exchange of messages.

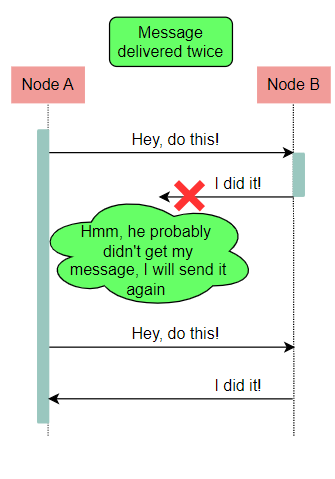
As the network is not reliable, these messages might get lost. Of course, to cope with this, nodes can retry with the hope that the network will recover at some point and deliver the message.

However, this means that the nodes may deliver messages multiple times because the sender can’t know what really happens.

The following illustration shows what happens when a node doesn’t deliver a message at all.



The following illustration shows a message that a node delivers twice.



This duplicate delivery of a message can create disastrous side effects.

### Example consequence

Think about what would happen if the message is supposed to signal the transfer of money between two bank accounts as part of a purchase. The bank may charge a customer twice for a product.

## Avoiding multiple deliveries of a message

To handle scenarios like the one above, we can take multiple approaches to ensure that the nodes only process a message once, even though it may be delivered multiple times. Let’s see these approaches.

### Idempotent operations approach

**Idempotent** is an operation we can apply multiple times without changing the result beyond the initial application.

#### Example of idempotent operation

An example of an idempotent operation is to add a value in a set of values. Even if we apply this operation multiple times, the operations that run after the first will have no effect, since the value will already be added to the set. Of course, we assume here that other operations cannot remove values from the set. Otherwise, the retried operation may add a value that was removed.

#### Example of non-idempotent operation

An example of a non-idempotent operation is to increase a counter by one, where the operation will have additional side effects every time it’s applied.

By using idempotent operations, we can have the guarantee that even if a node delivers a message multiple times and repeats the operation, the result will be the same.

However, idempotent operations commonly impose tight constraints on the system. So, in many cases, we cannot build our system so that all operations are idempotent by nature. In these cases, we can use another approach: the de-duplication approach.

### De-duplication approach

In the de-duplication approach, we give every message a unique identifier, and every retried message contains the same identifier as the original. In this way, the recipient can remember the set of identifiers it received and executed already. It will also avoid executing operations that are executed.

It is important to note that in order to do this, we must have control on both sides of the system: sender and receiver. This is because the ID generation occurs on the sender side, but the de-duplication process occurs on the receiver side.

#### Example

Imagine a scenario where an application sends emails as part of an operation. To send an email is not an idempotent operation. If the email protocol does not support de-duplication on the receiver side, we can’t be sure that every email displays exactly once to the recipient.

## Difference between delivery and processing

When we think about **exactly-once semantics**, it’s useful to distinguish between the notions of delivery and processing.

In the context of the above discussion, let’s consider **delivery** to be the arrival of the message at the destination node, at the hardware level.

Then, we consider **processing** to be the handling of this message from the software application layer of the node.

In most cases, we care more about how many times a node processes a message, than about how many times it delivers it. For instance, in our previous email example, we were mainly interested in whether the application would display the same email twice, and not whether it would receive it twice.

As the previous examples demonstrated, it’s impossible to have exactly-once delivery in a distributed system. However, it’s still sometimes possible to have exactly-once processing.

In the end, it’s important for us to understand the difference between these two notions, and clarify what we refer to when we talk about exactly-once semantics.

## Other delivery semantics

As a last note, it’s easy to see that we can easily implement **at-most-once** delivery semantics and **at-least-once** delivery semantics.

We can achieve the at-most-once delivery when we send every message only one time, no matter what happens. Meanwhile, we can achieve the at-least-once delivery when we send a message continuously until we get an acknowledgment from the recipient.

# Failure in the World of Distributed Systems

Let's see why failures occur in distributed systems, and how we can detect them.

**We'll cover the following**

* [One reason for failure](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#One-reason-for-failure)
* [One mechanism to detect failure](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#One-mechanism-to-detect-failure)
  + [Trade-off for the small timeout value](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#Trade-off-for-the-small-timeout-value)
  + [Trade-off for the large timeout value](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#Trade-off-for-the-large-timeout-value)
* [Failure detector](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#Failure-detector)
  + [Properties that categorize failure detectors](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#Properties-that-categorize-failure-detectors)
  + [A perfect failure detector](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6737253988827136#A-perfect-failure-detector)

We should understand that it is challenging to identify failure because of all the characteristics of a distributed system that the [Difficulties Designing Distributed Systems](https://www.educative.io/collection/page/10370001/4891237377638400/6155524821745664) lesson described. One of them is the asynchronous nature of the network.

## One reason for failure

The asynchronous nature of the network in a distributed system can make it very hard for us to differentiate between a crashed node and a node that is just really slow to respond to requests.

## One mechanism to detect failure

**Timeouts** is the main mechanism we can use to detect failures in distributed systems. Since an asynchronous network can infinitely delay messages, timeouts impose an artificial upper bound on these delays. As a result, we can assume that a node fails when it is slower than this bound. This is useful because otherwise, the assumption that the nodes are extremely slow would block the system is waiting for the nodes that crashed.

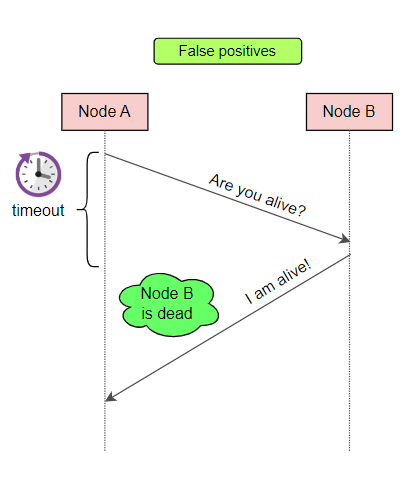
However, a timeout does not represent an actual limit. Thus, it creates the following trade-off.

### Trade-off for the small timeout value

If we select a smaller value for the timeout, our system will waste less time waiting for the nodes that have crashed.

At the same time, the system might declare some nodes that have not crashed dead, while they are actually just a bit slower than expected.

The following illustration shows this trade-off phenomenon.

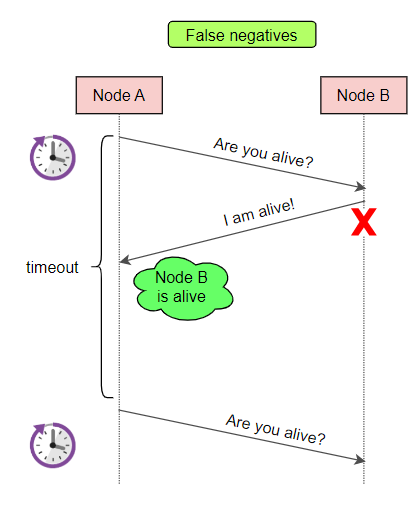


### Trade-off for the large timeout value

If we select a larger value for the timeout, the system will be more lenient with slow nodes.

At the same time, the system will be slower in identifying crashed nodes, in some cases wasting time while waiting for them.

The following illustration shows this trade-off phenomenon.



This is a fundamental problem in the field of distributed systems.

## Failure detector

A failure detector is the component of a node that we can use to identify other nodes that have failed.

This component is essential for various algorithms that need to make progress in the presence of failures. There has been extensive research about failure detectors.

### Properties that categorize failure detectors

We can distinguish the different categories of failure detectors through two basic properties that reflect the trade-off:

1. Completeness

**Completeness** corresponds to the percentage of crashed nodes a failure detector successfully identifies in a certain period.

1. Accuracy

**Accuracy** corresponds to the number of mistakes a failure detector makes in a certain period.

### A perfect failure detector

A perfect failure detector is the one with the strongest form of completeness and accuracy. That is, it is one that successfully detects every faulty process without ever assuming a node has crashed before it actually does.

As expected, it is impossible to build a perfect failure detector in purely asynchronous systems. Still, we can even use imperfect failure detectors to solve difficult problems. One such example is the problem of consensus.

# Stateless and Stateful Systems

Let's see the difference between stateless and stateful systems.

**We'll cover the following**

* [Stateless system](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Stateless-system)
  + [Examples](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Examples)
* [Stateful systems](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Stateful-systems)
  + [Example](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Example)
* [Some interesting observations](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Some-interesting-observations)
  + [Benefits of stateless systems over stateful systems](https://www.educative.io/module/page/lOn30BIA1wV52NDAg/10370001/4527677663084544/6035680073613312#Benefits-of-stateless-systems-over-stateful-systems)

We can say that a system belongs in one of the two following categories:

* Stateless systems
* Stateful systems

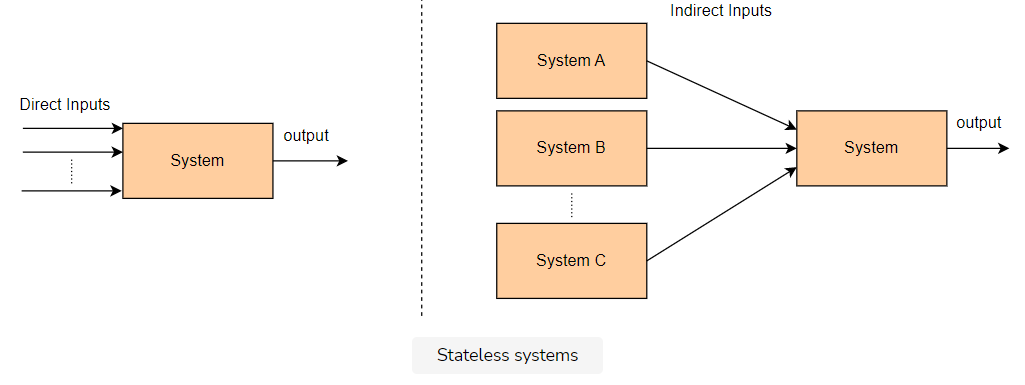
## Stateless system

A stateless system maintains no state of what happened in the past and performs its capabilities purely based on the inputs we provide to it.

### Examples

A contrived stateless system receives a set of numbers as input, calculates their maximum, and returns it as a result. These inputs are either direct or indirect. Direct inputs are inputs that are included in the request, while indirect inputs are inputs that are potentially received from other systems to fulfill the request.

Imagine a service that calculates the price of a specific product by retrieving its initial price and any currently available discounts from some other services, and then performing the necessary calculations with this data. This service is still stateless.

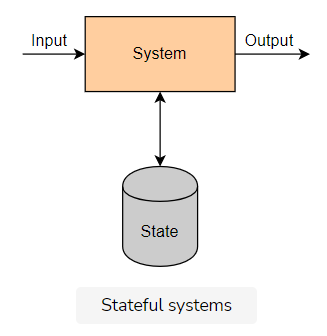


## Stateful systems

Stateful systems are responsible for maintaining and mutating a state. Their results depend on this state.

### Example

Imagine a system that stores the ages of all the employees of a company, and we can ask it the maximum age. This system is stateful since the result depends on the employees we register in it.



## Some interesting observations

* Stateful systems are beneficial in real life because computers are much more capable than humans of storing and processing data.
* Maintaining state involves additional complexity. For example, we must decide what’s the most efficient way to store and process it, how to perform back-ups, etc.
* As a result, it’s usually wise to create an architecture that contains clear boundaries between stateless components (which perform business capabilities) and stateful components (which handle data).

### Benefits of stateless systems over stateful systems

Stateless distributed systems are much easier to design, build and scale, compared to stateful ones.

The main reason for this is that we consider all the nodes (e.g., servers) of a stateless system identical. This makes it a lot easier for us to balance traffic between them, and scale by adding or removing servers.

However, stateful systems present many more challenges. As different nodes can hold different pieces of data, they require additional work. They need to direct traffic to the right place and ensure each instance is in sync with the others.

As a result, some of the course’s examples include stateless systems. However, the most challenging problems we cover in this course mainly concern stateful systems.