Dotnet Architectural Principles

You should architect and design software solutions with maintainability in mind. These principles will guide you towards building applications out of discrete components that are not tightly coupled to other parts of your application, but rather communicate through explicit interfaces or messaging systems.

# Common design principles

## Separation of concerns

A guiding principle when developing is separation of concerns. This principle asserts that software should be separated based on the kinds of work it performs. For instance, consider an application that includes logic for identifying noteworthy items to display to the user, and which formats such items in a particular way to make them more noticeable. The behaviour responsible for choosing which items to format should be kept separate from the behaviour responsible for formatting the items, since these behaviours are separate concerns that are only coincidentally related to one another.

Architecturally, applications can be logically built to follow this principle by separating core business behaviour from infrastructure and user interface logic. Ideally, business rules and logic should reside in a separate project, which should not depend on other projects in the application. This separation helps ensure that the business model is easy to test and can evolve without being tightly coupled to low level implementation details (it also helps if infrastructure concerns depend on abstractions defined in the business layer). Separation of concerns is a key consideration behind the use of layers in application architectures.

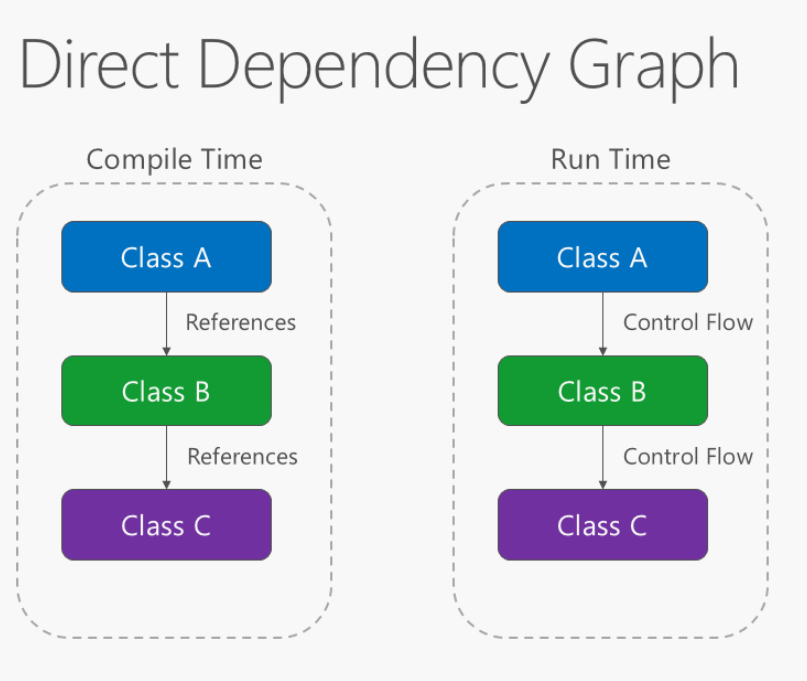
Encapsulation

Different parts of an application should use encapsulation to insulate them from other parts of the application. Application components and layers should be able to adjust their internal implementation without breaking their collaborators if external contracts are not violated. Proper use of encapsulation helps achieve loose coupling and modularity in application designs, since objects and packages can be replaced with alternative implementations if the same interface is maintained.

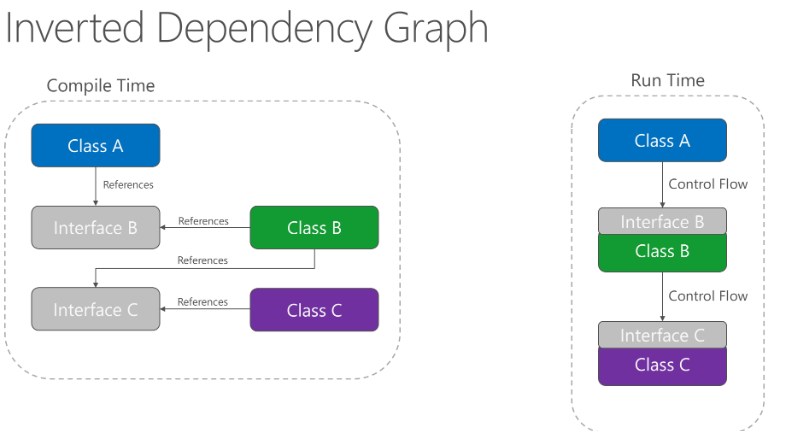
In classes, encapsulation is achieved by limiting outside access to the classes internal state. If an outside actor wants to manipulate the state of the object, it should do so through a well-defined function (or property setter), rather than having direct access to the private state of the object. Likewise, application components and applications themselves should expose well-defined interfaces for their collaborators to use, rather than allowing their state to be modified directly. This approach frees the applications internal design to evolve over time without worrying that doing so will break collaborators, so if the public contracts are maintained.

Mutable global state is antithetical to encapsulation. A value fetched from mutable global state in one function cannot be relied upon to have the same value in another function (or even further in the same function). Understanding concerns with mutable global state is one of the reasons programming languages like C# have support for different scoping rules, which are used everywhere from statements to methods to classes. It’s worth noting that data driven architectures which rely on a central database for integration within and between applications are, themselves, choosing to depend on the mutable global state represented by the database. A key consideration in domain driven design and clean architecture is how to encapsulate access to data, and how to ensure application state is not made invalid by direct access to its persistence format.

## Dependency Inversion

The direction of dependency within the application should be in the direction of abstraction, not implementation details. Most applications are written such that compile-time dependency flows in the direction of runtime execution, producing a direct dependency graph. That is, if class A calls a method of class B and class B calls a method of class C, then at compile time class A will depend on class B, and class B will depend on class C, as shown:  
  


Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at run time, but for B to depend on an interface controlled by A at compile time (thus *inverting* the typical compile-time dependency). At run time, the flow of program execution remains unchanged, but the introduction of interfaces means that different implementations of these interfaces can easily be plugged in. Inverted dependency graph:



Dependency inversion is a key part of building loosely coupled applications, since implementation details can be written to depend on and implement higher-level abstractions, rather than the other way around. The resulting applications are more testable, modular and maintainable as a result. The practice of dependency injection is made possible by following the dependency inversion principle.

## Explicit dependencies

Methods and classes should explicitly require any collaborating objects they need in order to function correctly. It is called the ‘*Explicit Dependencies Principle*’. Class constructors provide an opportunity for classes to identify the things they need in order to be in a valid state and to function properly. If you define classes that can be constructed and called, but that will only function properly if certain global or infrastructure components are in place, these classes are being dishonest with their clients. The constructor contract is telling the client that it only needs the things specified (possibly nothing if the class is just using a parameter less constructor), but then at runtime it turns out the object really did need something else.

By following the explicit dependencies principle, your classes and methods are being honest with their clients about what they need in order to function. Following the principle makes your code more self-documenting and your coding contracts more user friendly, since users will come to trust that if they provide what’s required in the form of method or constructor parameters, the objects they’re working with will behave correctly at run time.

## Single responsibility

The single responsibility principle applies to object-oriented design but can also be considered as an architectural principle similar to separation of concerns. It states that objects should have only one responsibility and that they should have only one reason to change. Specifically, the only situation in which the object should change is if the manner in which it performs its one responsibility must be updated. Following this principle helps to produce more loosely coupled and modular systems, since many kinds of new behaviour can be implemented as new classes, rather than by adding additional responsibility to existing classes. Adding new classes is always safer than changing existing classes, since no code yet depends on the new classes.

In a monolithic application, we can apply the single responsibility principle at a high level to the layers in the application. Presentation responsibility should remain in the UI project, while data access responsibility should be kept within an infrastructure project. Business logic should be kept in the application core project, where it can be easily tested and can evolve independently from other responsibilities.

When this principle is applied to application architecture and taken to its logical endpoint, you get microservices. A given microservice should have a single responsibility. If you need to extend the behaviour of a system, it’s usually better to do it by adding additional microservices, rather than by adding responsibility to an existing one.

## Don’t repeat yourself (DRY)

The application should avoid specifying behaviour related to a particular concept in multiple places as this practice is a frequent source of errors. At some point a change in requirements will require changing this behaviour. It’s likely that at least one instance of the behaviour will fail to be updated, and the system will behave inconsistently.

Rather than duplicating logic, encapsulate it in a programming construct. Make this construct the single authority over this behaviour and have any other part of the application that requires this behaviour use the new construct.

**Note:** avoid binding together behaviour that is only coincidentally repetitive. For example, just because two constants have the same value, that doesn’t mean you should have only one constant, if conceptually they’re referring to different things. Duplication is always preferable to coupling to the wrong abstraction.

## Bounded contexts

Bounded contexts are a central pattern in domain driven design. They provide a way of tackling complexity in large applications or organisations by breaking it up into separate conceptual modules. Each conceptual module then represents a context that is separated from other contexts (hence, bounded), and can evolve independently. Each bounded context should ideally be free to choose its own names for concepts within it and should have exclusive access to its own persistence store.

At minimum, individual web applications should strive to be their own bounded context, with their own persistence store for their business model, rather than sharing a database with other applications. Communication between bounded contexts occurs through programmatic interfaces, rather than through a shared database, which allows for business logic and events to take place in response to changes that take place. Bounded contexts map closely to microservices, which also are ideally implemented as their own bounded contexts.

## Architectural Agility

This principle highlights the importance of designing software architectures that can adapt to changing requirements and conditions. Architects should consider using techniques such as modular design, microservices or other architectural patterns that promote flexibility and adaptability.

The principle of architectural agility, while not as widely recognized as some other principles, is an important concept in modern software architecture. It emphasizes the need to design software architectures that can adapt to changing requirements, conditions or technologies. The term “Architectural Agility” is more of a philosophy that encompasses several practices and pattens that enable a system to evolve more easily. Although the term itself might not be as widely referenced, the underlying idea is echoed in various architectural styles, methodologies and literature. The concepts of architectural agility is closely related to agile software development methodologies, which prioritize flexibility and adaptability in software projects.

Other areas of software development that support architectural agility include:

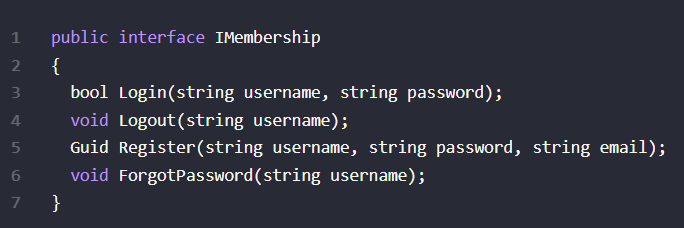
* Agile Software development, whose methodologies focus on iterative development and adapting to change, often allowing for ongoing architectural agility.
* Microservices Architecture, which is designed to break systems into small, loosely coupled services that can be developed, deployed and scaled independently. Each microservice can evolve without requiring the entire system to be impacted.
* Continuous delivery and DevOps practices that emphasize close collaboration between development and operations teams, and promote rapid, frequent reliable delivery of software.

## Interface Segregation Principle

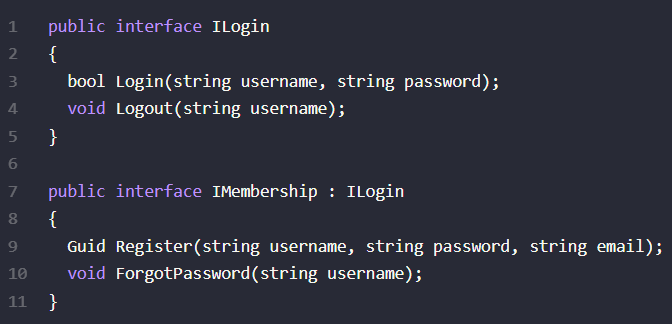
The interface segregation principle (ISP) states that clients should not be forced to depend on methods that they do not use. Interfaces should belong to clients, not to libraries or hierarchies. Application developers should favour thin, focused interfaces to “fat” interfaces that offer more functionality than a particular class or method needs.



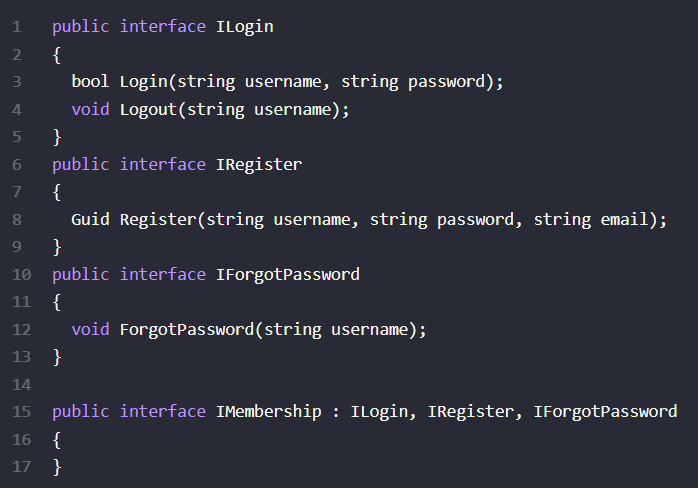
In many languages, such as C# interfaces can inherit from multiple other interfaces. Thus if you need a larger interface in some parts of the application, but not in others, you may be able to compose it from two or more other interfaces. This is also a good approach to keep in mind if you find yourself refactoring a legacy codebase, which already has large interfaces that you can’t break. Consider an interface like this one:



It’s easy to imagine such an interface growing completely out of control and having more functionality than any one class would ever require. To keep, say a login form from having more methods on it than it needs, you could create a login-specific interface and have the existing interface extend from it:



You could extend this further, as required, perhaps ending up with an original “fat” interface that only exists for legacy reasons, and is totally composed of other interfaces:



Ideally your thin interface should be cohesive, meaning they have groups of operations that logically belong together. This will prevent you from ending up with one interface per method most of the time in real world systems (as opposed to the above trivial example).

Another benefit of smaller interfaces is that they are easier to implement fully. They also provide greater flexibility in how you implement functionality, since parts of a larger interface can be implemented in different ways. Consider the repository pattern, which usually includes methods for reading as well as writing. A common performance pattern for database reads it so add a caching layer, but this generally only makes sense for read operations. Likewise, scalability can often be improved by queuing commands (like write operations) rather than executing them immediately, but you wouldn’t queue queries. Thus having an IRepository interface composed of an IReadOnlyRepository and an IWriteRepository would allow base implementations that go against a data store and separate implementations that employ caching and queuing as well.

## SOLID

The SOLID principles of object-oriented design include these five principles:

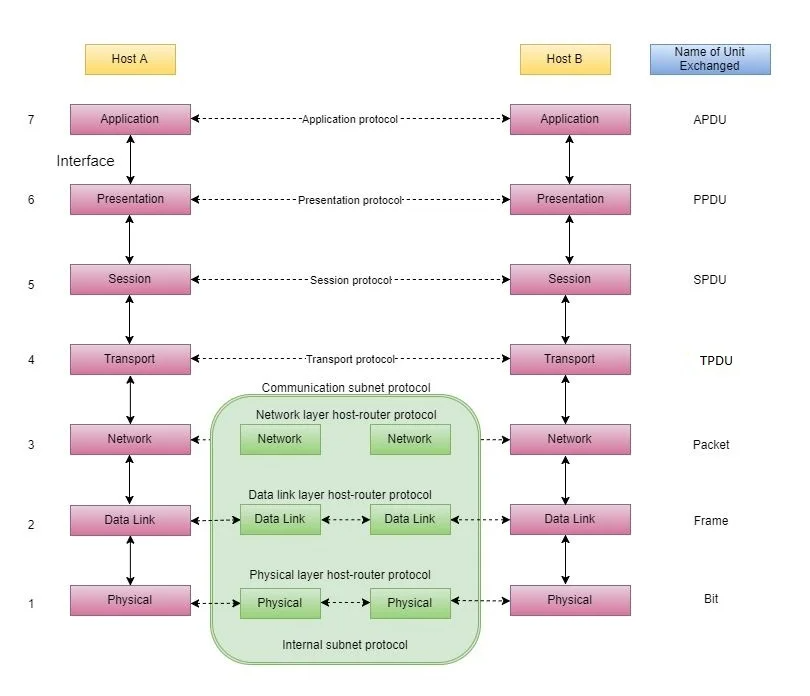
* SRP – single responsibility principle
* OCP – Open/Closed Principle
* LSP – Liskov Substitution principle
* ISP – Interface Segregation Principle
* DIP – Dependency Inversion Principle

Generally, software should be written as simply as possible in order to produce the desired result. However once updating the software becomes painful, the software’s design should be adjusted to eliminate the pain. Often, these principles, in addition to the more general ‘Don’t repeat yourself principle’ can be used as a guide while refactoring the software into a better design.

# The OSI Model – Features, Principles and Layers

There are n numbers of users who use computer network and are located over the world. So to ensure, national and worldwide data communication, systems must be developed which are compatible to communicate with each other ISO has developed a standard. ISO stands for International organization of Standardization. This is called a model for Open System Interconnection (OSI) and is commonly known as OSI model.

The ISO-OSI model is a seven-layer architecture. It defines seven layers or levels in a complete communication system.



## Feature of OSI Model

* Big picture of communication over network is understandable through this OSI model.
* We see how hardware and software work together.
* We can understand new technologies as they are developed.
* Troubleshooting is easier by separate networks.
* Can be used to compare basic functional relationships on different networks.

## Principles of OSI Reference Model

The OSI reference model has 7 layers. The principles that were applied to arrive at the seven layers can be briefly summarized as follows:

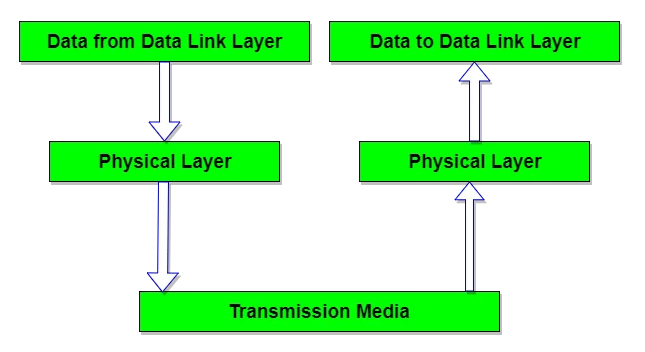
1. A layer should be created where a different abstraction is needed.
2. Each layer should perform a well-defined function.
3. The function of each layer should be chosen with an eye toward defining internationally standardized protocols.
4. The layer boundaries should be chosen to minimize the information flow across the interfaces.
5. The number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity and small enough that architecture does not become unwieldly.

## OSI Model Layer 1: The Physical Layer

Physical layer is the lowest layer of the OSI reference model. It is responsible for sending bits from one computer to another. This layer is not concerned with the meaning of the bits and deals with the setup of physical connection to the network and with transmission and reception of signals.

Following are the various functions performed by the physical layer of the OSI model:

1. Representation of Bits: data in this layer consists of a stream of bits. The bits must be encoded into signals for transmission. It defines the type of encoding i.e., how 0s are changed to signal.
2. Data Rate: this layer defines the rate of transmission which is the number of bits per second.
3. Synchronization: it deals with the synchronization of the transmitter and receiver. The sender and receiver are synchronized at bit level.
4. Interface: The physical layer defines the transmission interface between devices and transmission medium.
5. Line configuration: this layer connects devices with the medium; point to point configuration and multipoint configuration.
6. Topologies: Devices must be connected using the following topologies: Mesh, star, ring and bus.
7. Transmission modes: physical layer defines the direction of transmission between two devices: simplex, half duplex, full duplex.
8. Deals with baseband and broadband transmission.



## Design issues with Physical Layer

* The physical layer is concerned with transmitting raw bits over a communication channel.
* The design issue has to do with making sure that when one side sends a 1 bit, it is received by the other side as a 1 bit and not as a 0 bit.

Typical questions here include:

* How many volts should be used to represent a 1 bit and how many for a 0?
* How many nanoseconds a bit lasts?
* Whether transmission may proceed simultaneously in both directions?
* How many pins the network connector has and what each pin is used for?

The design issues here largely deal with mechanical electrical and timing interfaces, and the physical transmission medium, which lies below the physical layer.

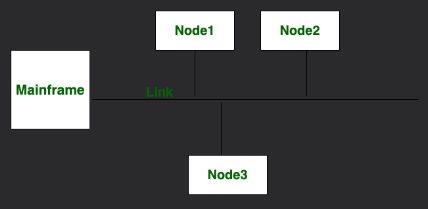
## Multipoint vs Point to Point

|  |  |
| --- | --- |
| Point to point communication | Multipoint Communication |
| Point to point communication means the channel is shared between two devices. | Multipoint communication means the channel is shared among multiple devices or nodes. |
| In this communication, there is dedicated link between nodes. | In this communication, link is provided at all times for sharing the connection among nodes. |
| In this communication the entire capacity is reserved between these connected two devices with the possibility of waste of network bandwidth/resources. | In this communication, the entire capacity isn’t reserved by any two nodes and the network bandwidth is maximally utilized. |
| In this communication there is one transmitter and one receiver | In this communication, there is one transmitter and many receivers. |
| In point-to-point connections the smallest distance is most important to reach the receiver. | In multi-point connections, the smallest distance is not important to reach the receiver. |
| Point to point communication provides security and privacy because communication channel is not shared. | Multipoint communication does not provide security and privacy because communication channel is shared |

### Point to point:



### Multipoint:

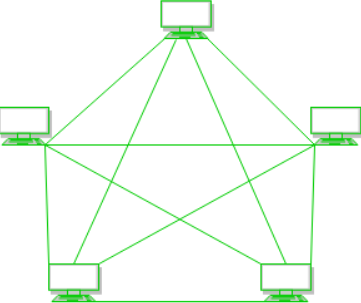


## Topologies

There are different types of topologies, like the following.

### Mesh Topology

In a mesh topology, every device is connected to another device via a particular channel. In Mesh Topology, the protocols used are AHCP (Ad Hoc Configuration Protocols), DHCP (Dynamic Host Configuration Protocol)



Every device is connected to another via dedicated channels. These channels are known as links.

* Suppose the N number of devices are connected with each other in a mesh topology, the total number of ports that are required by each device is N-1. In this figure above there are 5 devices connected to each other, hence the total number of ports required by each device is 4. The total number of ports required = N\*N-1.
* Suppose, N number of devices are connected with each other in a mesh topology, then the number of dedicated links required to connect them is N(N-1)/2. In the figure there are 5 devices connected to each other, hence the total number of links required is 5\* 4/2=10

### Advantages of Mesh Topology

* Communication is very fast between the nodes.
* Mesh Topology is robust.
* The fault is diagnosed easily. Data is reliable because data is transferred among the devices through dedicated channels or links.
* Provides security and privacy.

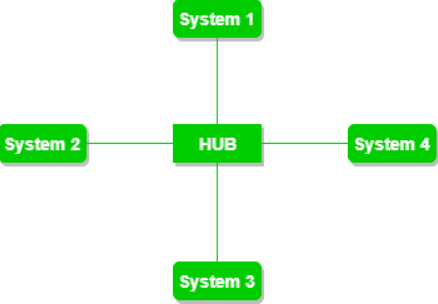
### Disadvantages of Mesh Topology

* Installation and configuration are difficult.
* The cost of cables is high as bulk wiring is required, hence suitable for a smaller number of devices.
* The cost of maintenance is high.

A common example of mesh topology is the internet backbone, where various internet service providers are connected to each other via dedicated channels. This topology is also used in military communication systems and aircraft navigation systems.

### Star Topology

In Star Topology, all the devices are connected to a single hub through a cable. This hub is the central node, and all other nodes are connected to the central node. The hub can be passive in nature i.e., not an intelligent hub such as broadcasting devices, at the same time the hub can be intelligent known as an active hub. Active hubs have repeaters in them. Coaxial cables or RJ-45 cables are used to connect the computers. In Star Topology, many popular Ethernet LAN protocols are used as CD (Collision Detection), CSMA (Carrier Sense Multiple Access).



### Advantages of Star Topology

* If N devices are connected to each other in a star topology, then the number of cables required to connect them is N. So, it is easy to set up.
* Each device requires only 1 port i.e., to connect to the hub, therefore the total number of ports required is N.
* It is Robust. If one link fails only that link will be affected.
* Easy to identify and isolate fault.
* Star topology is cost effective as it uses inexpensive coaxial cable.

### Disadvantages of Star Topology

* If the concentrator (hub) on which the whole topology relies fails, the whole system will crash down.
* The cost of installation is high.
* Performance is based on the single concentrator i.e., hub.

A common example of star topology is a local area network (LAN) in an office where all computers are connected to a central hub. This topology is also used in wireless networks where all devices are connected to a wireless access point.

### Bus Topology

Bus Topology is a network type in which every computer and network device is connected to a single cable. It is bi-directional. It is a multi-point connection and a non-robust topology because if the backbone fails the topology crashes. In Bus Topology, various MAC (Media Access Control) protocols are followed by LAN ethernet connections like TDMA, Pure Aloha, CDMA, Slotted Aloha.



The figure above shows a bus topology with shared backbone cable. The nodes are connected to the channel via drop lines.

### Advantages of Star Topology

* If N devices are connected to each other in a bus topology, then the number of cables required to connect them is 1, known as the backbone cable, and N drop lines are required.
* Coaxial or twisted pair cables are mainly used in bus-based networks that support up to 10 Mbps.
* The cost of the cable is less compared to other topologies, but it is used to build small networks.
* Bus topology is familiar technology as installation and troubleshooting techniques are well known.
* CSMA is the most common method for this type of topology.

### Disadvantages of Star Topology

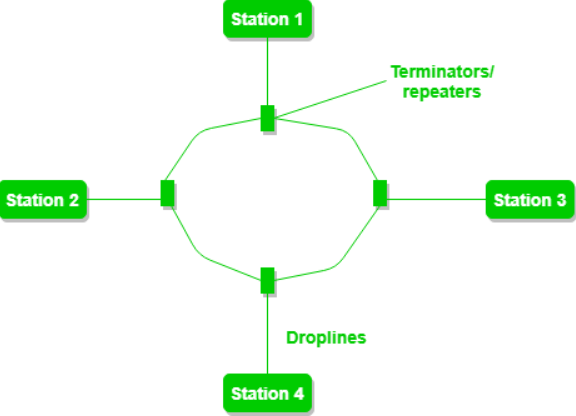
* A bus topology is quite simple, but still requires a lot of cabling.
* If the common cable fails, then the whole system will crash down.
* If the network traffic is heavy, it increases collisions in the network. To avoid this, various protocols are used in the MAC layer known as Pure Aloha, slotted aloha, CSMA/CD etc.
* Adding new devices to the network would slow down networks.
* Security is very low.

A common example of bus topology is the Ethernet LAN, where all devices are connected to a single coaxial cable or twisted pair cable. This topology is also used in television networks.

### Ring Topology

In a ring topology, it forms a ring connecting devices with exactly two neighbouring devices. A number of repeaters are used for Ring topology with a large number of nodes, because if someone wants to send some data to the last node in the ring topology with 100 nodes, then the data will have to pass through 99 nodes to reach the 100th node. Hence to prevent data loss repeaters are used in the network.

The data flows in one direction, i.e., it is unidirectional, but it can be made bidirectional by having 2 connections between each Network Node, it is called Dual Ring Topology. In-Ring Topology, the Token Ring Passing protocol is used by the workstations to transmit the data.



The figure above shows a ring topology comprising of 4 stations connected with each-other forming a ring. The most common access method of ring topology is token passing.

* Token passing: A network access method in which a token is passed from one node to another node.
* Token: A frame that circulates around the network.

### Operations of Ring Topology

1. One station is known as a monitor station which takes all the responsibility for performing the operations.
2. To transmit the data, the station has to hold the token. After the transmission is done, the token is to be released for other stations to use.
3. When no station is transmitting the data, then the token will circulate in the ring.
4. There are two types of token release techniques: Early token release releases the token just after transmitting the data and Delayed token release releases the token after the acknowledgment is received from the receiver.

### Advantages of Ring Topology

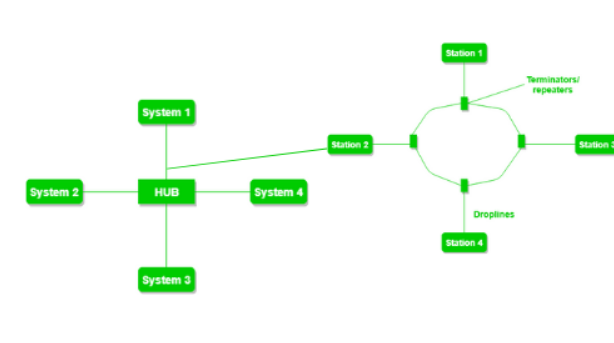
* The data transmission is high speed.
* The possibility of collision is minimum in this type of topology.
* Cheap to install and expand.
* It is less costly than a star topology.

### Disadvantages of Ring Topology

* The failure of a single node in the network can cause the entire network to fail.
* Troubleshooting is difficult in this topology.
* The addition of stations in between or the removal of stations can disturb the whole topology.
* Less secure.

### Hybrid Topology

This topological technology is the combination of all the various types of topologies we have studied above. Hybrid Topology is used when the nodes are free to take any form. It means these can be individuals such as Ring or Star Topology or can be a combination of various types of topologies seen already.



### Advantages of Hybrid Topology

* This topology is very flexible.
* The size of the network can be easily expanded by adding new devices.

### Disadvantages of Hybrid Topology

* It is challenging to design the architecture of the hybrid network.
* Hubs used in this topology are very expensive.
* The infrastructure cost is very high as a hybrid network requires a lot of cabling and network devices.

A common example of a hybrid topology is a university campus network. The network may have a backbone of a star topology, with each building connected to the backbone through a switch or router. Within each building there may be a bus or ring topology connecting the different rooms and offices. The wireless access points also create a mesh topology for wireless devices. This hybrid topology allows for efficient communication between different buildings while providing flexibility and redundancy within each building.

### Terms

### Coaxial Cables

Coaxial cables, sometimes known as coax cables are electrical cables that can transmit electrical signals from one point to another.

This technology has been around since the early 20th century, with these cables mainly used to connect satellite antenna facilities to homes and businesses thanks to their durability and ease of installation.



**What is RG cable?**

There are different types of coaxial cable, which vary by gauge and impedance. Gauge refers to the cables thickness and is measured by the radio guide measurement or RG number. The higher the RG number, the thinner the central conductor code is.

‘

**How do Coaxial Cables work?**

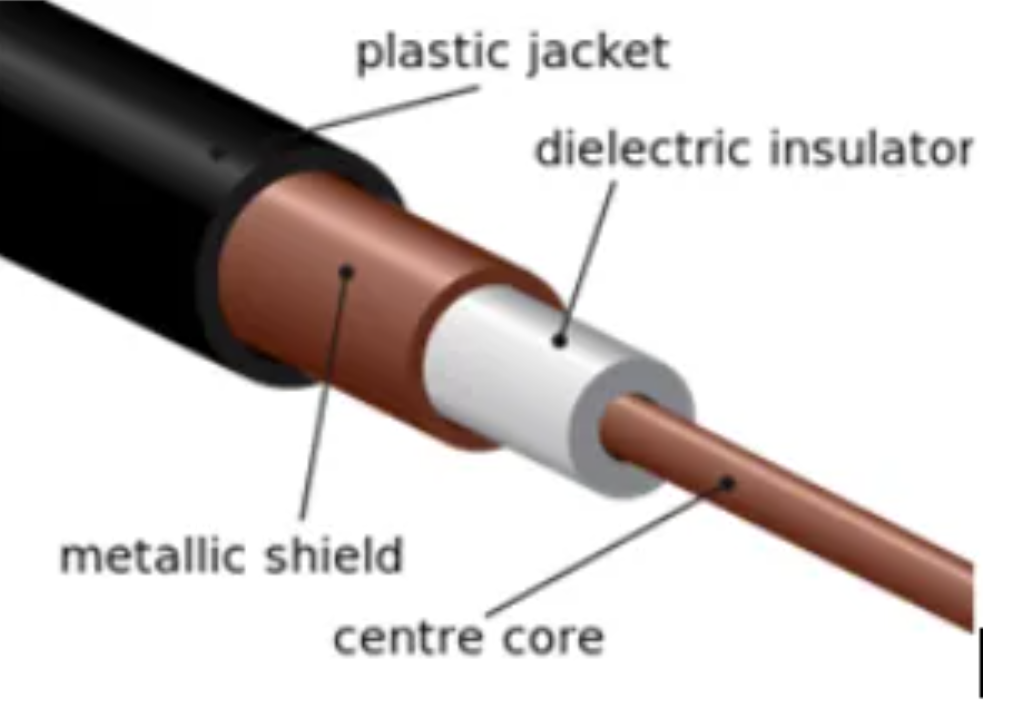
Coaxial cables are a popular choice because their shielded design allows the centre conductor to transmit data quickly while being protected from damage and interference.

Coaxial cables are mainly built up of these four different layers:

1. A centre conductor which is usually a copper wire, which data and video travels through.
2. Surrounding the copper wire is a dielectric plastic insulator.
3. A braided mesh made from copper then helps to shield the cable from electromagnetic interference (EMI).
4. The external layer is a plastic coating which protects the internal layers from damage.

Coaxial cable works by carrying data in the centre conductor, while the surrounding layers of shielding stop any signal loss (also called attenuation loss) and help reduce EMI.

The first layer, called the dielectric, provides distance between the core conductor and the outer layers, as well as some insulation. The next layers collectively referred to as the shield, keep electrical impulses and radio transmissions out. The different layers of a coaxial cable are shown in the image below:



The transmission speed of a coaxial cable is 10Mbps (megabits per second).

**Advantages of coaxial cable**

* Inexpensive
* Easy to wire and install
* Easy to expand
* Good resistance to EMI
* Up to 10Mbps capacity
* Durable

Another benefit of coaxial cable is the electromagnetic field carrying the signal exists only in the space between the inner and outer conductors. This means coaxial cable can be installed next to metal objects without losing power, unlike other types of transmission lines.

**Disadvantages of coaxial cable**

The main disadvantage of using coaxial cable is that single cable failure can take down an entire network.

## Fibre Optic Cables

Fibre optics, or optical fibres are long, thin strands of carefully drawn glass about the diameter of human hair. These strands are arranged in bundles called fibre optic cables. We rely on them to transmit light signals over long distances.

Optical fibre cabling is used to transfer information via pulses of light, which pass along one or more transparent plastic or glass pipes. In some cases, this can be more than several hundred pipes. Each of these strands is little wider than an average hair and is normally surrounded by a further layer of cladding which is also in plastic or glass but constructed at a different density to the main inner strand. A sheath made up of several layers of insulated casing is also wrapped around the cladded fibers. This usually comprises of a protective wrapper, known as a buffer tube, followed by a final outer jacket designed to protect the entire multi-stranded cable.

**Optical Fibre Internet and Networking**

Fibre optic internet cable is increasingly popular. This is due to the higher speeds and bandwidth it can provide compared to standard ethernet or Wi-Fi signals delivered by coaxial or even copper wire from street level exchanges. This means that fibre networking is a far better choice where high speeds are advantageous or for particularly intensive data transfer needs. Much of this is also true for fibre optic phone lines.

Fibre optic bandwidth is usually significantly higher

Bandwidth: the maximum amount of data your connection can handle at any moment

Speed: the maximum rate at which you can transmit data