Network software

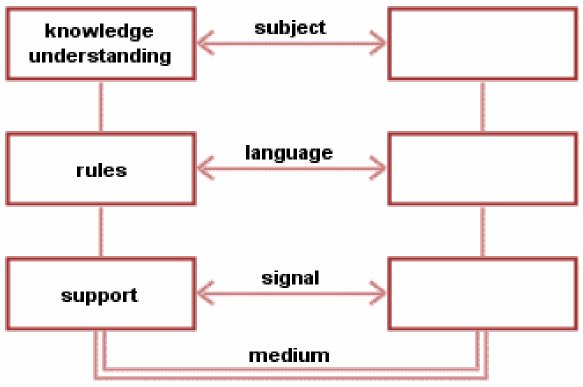
Notion of protocol

In the **previous chapiter**, it was said that the communication system was ir charge of the information transfer and the right delivery to the final user. In this context, it is necessary to set up rules so that both sender and receiver can understand each other. In the same way, we saw it is necessary for users to agree about the meaning of sent messages. In a more general way, the set of transmission processes are managed by mechanisms called **protocols**.

Simple example of protocol: an ordinary conversation between two friends Actually, a given number of implicitly established rules regulates the conversation For instance, these rules are: don't speak both at the same time and speak about the same subject. These rules form a protocol.

Layer structure and protocol hierarchies

The layer structure already appears in the previous example: communication rules can be stacked as shown on the following outline:



Communications over networks work exactely the same way. In order to reduce the design complexity of network software and obtain the independance between software and hardware, we carry out a functional cutting of the whole communication process into **layers or abstraction levels**. A layer corresponds to a set of functions or processes coherent with each other and providing a given global function. Two adjacent layers have independant functions but they interconnect through what we call an **interface**: layers exchange data through interfaces. Let's take a concrete example: a washing machine! Let's consider the washing machine as a layer (its function is "to wash"). It is connected to two other layers: the electricity layer and the water layer. Interfaces are the socket and plug system and the water tube.

Protocols correspond to a given implementation of a layer. This model is actually based on the famous saying: "divide to reign". The set of layers and protocols is called **network architecture**. The set of protocols used by a system with one protocol per layer is called the **stack of protocols** (example of protocol stack

TCP/IP).

To work, each layer uses the previous one. Each layer must provide some services to the previous ones, without these latter to "know" the details of the implementation of these services. This guarantee that the whole system is modular and that implementations are independant with each other. In such a system, the layer *N* of a machine is in charge of the communication with the layer *N* of another machine using one or several protocols and underlying layers.

As we saw, 2 adjacent layers are connected together by an **interface**. This interface defines the services a lower layer provides to the upper layer. When designing a network architecture, and still in order to respect the independance of implementations, it is very important to define these interfaces. Layers must therefore carry out a very well defined set of functions. Later, we will see interfaces are developed around **primitives**.

Principles of layer design

As a machine may run several processes (or programs) that may use the network resources, and as a machine may communicate with many other processes, it is essential that every layer has a mechanism ti identify senders (local processes) and receivers (remote processes).

It is also advisable to think about data transmission rules. A communication may indeed be carried out in many ways:

- **simplex communication**: the data is carried only in one direction,
- **semiduplex communication**: the data can be carried in both directions, but not at the same time,
- **duplex communication**: the data can be carried in both directions at the same time.

Then we must consider the quality of transmission: a layer must be able to detect errors due to physical communication circuits. There are a lot of error detection and correction systems, but both ends have to agree about the system to use. Besides, ends must have a system that allows them to send to the other an acknowledgment message to tell that the message has been correctly received. There is actually another quality issue: preserving of the order of messages. The sender must make possible for the receiver to reorder messages, for instance by numbering messages, but this does not explain how to order messages effectively.

Still another important issue: some processes are not able to handle message of any size. We must therefore provide deassembling, transmission and reassembling mechanisms. Such mechanisms may also be useful for the opposite problem: in order to improve the transmission rate, we may need to split a message into smaller parts. The receiving end must then be able to reassemble the small message into the big first one.

These issues are the most important we must consider when developing layer. We may however consider some others: receiving end congestion (this happens for instance when the sender is too fast), rational rule-based message routing, multiplexing problems (especially for the physical layer) when the underlying layer has a limited amount of transmission channels...

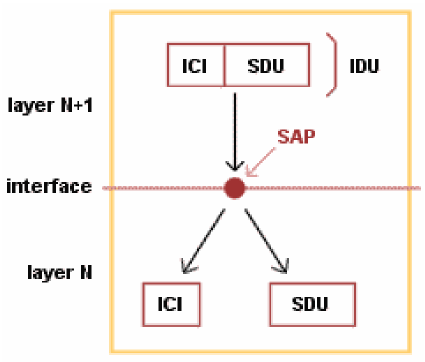
Services

Interfaces and services

The goal of every layer is to provide services to the upper layer. In this case, we say the underlying layer is a **service provider** to the upper layer, which is a **service user**. Active elements of a layer are called **entities**. These entities can be software entities (a process for instance) or hardware entities (a chip for instance). Entities from the same layer *N* of two different machines are called **peer entities**.

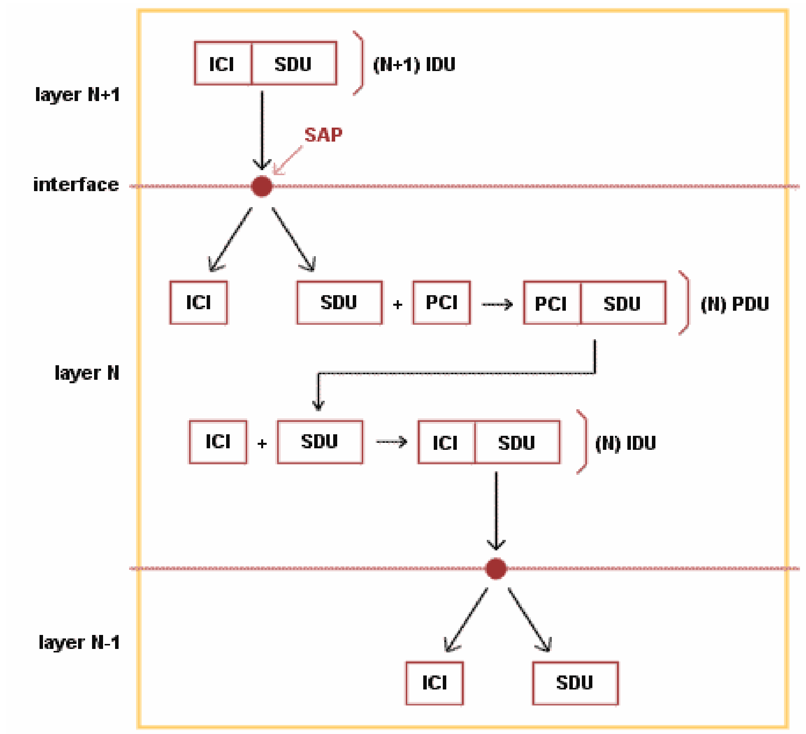
Services of a layer *N* can be reach using **service access points** (SAP). Each SAP is uniquely identified by an address. Typically, SAPs of the phone network are phone sockets and addresses are simply phone numbers.

In order to make two adjacent layers to communicate, some rules must be set up for the interface. An entity from the layer *N+1* gives to an entity of the layer *N* an **Interface Data Unit** (IDU) through the SAP. The IDU is actually made up with 2 things: a **Service Data Unit** (SDU) and some **Interface Control Information** (ICI):



The SDU is the data that 2 peer entities exchange, but it is also what the layer *N+1* of the sender gives to the layer *N*. The control information is used to help the lower layer. It is for example the number of bytes contained in the SDU (this can be used in the function that controls data integrity).

To send a SDU, a layer *N* might need to split it into several parts. When exchanging this data with its peer entity, every piece of data is fitted with a **Protocol Control Information** (PCI) contained in a **header**, and then sent as a **Protocol Data Unit** (PDU). Headers are used by peer entities to carry their peer protocol. This PDU then becomes the SDU for the layer *N* and is going to be sent to the layer *N-1* via the SAP. Here is how all this works:



Finally, all messages fit together. This mechanism is often described as a **wrapping** mechanism.

Service primitives

A service is formally by a set of **primitives** or operations a user or other entities can invoke to access the service. That is what materializes an interface. We commonly classify service primitives into 4 classes:

primitive	meaning
request	an entity is requesting a service (we are requesting a connection to a remote computer)
indication	an entity is informed of an event (the receiver has just received a connection request)
response	an entity is responding to an event (the receiver is sending the permission to connect)
confirm	an entity acknowledges the response to its request (the sender acknoledge the permission to connect to the remote host)

Most primitives need parameters. For instance, parameters of a CONNECT.request (used to query a connection) are the machine you want to connect to, the service you want to use (FTP, telnet...) and the maximum size of exchanged packets.

A **acknowledged service** is a service that requires a request, an indication, a response and a confirm. A **unacknowledged service** is a service that requires only a request and an indication. Typically, the service that establishes a connection is an acknowledged service because the peer entity must agree to set the connection. On the other hand, data transmission may be an unacknowledged service, whether we want an acknowledgement or not.

On an implementation point of view, primitives correspond to functions we can use in a program to access a given service.

services/protocols relations

A service is a set of primitives a layer provides to the upper layer. The service defines the operations a layer may realize, but it does not tell how these operations are really realized. The most characteristic element of a service is the interface between two adjacent layers.

Conversely, a protocol is a set of rules that applies to the meaning and format of messages exchanged between two peer entities. Entities uses protocols to **implement** service specifications. A service may therefore remain the same with two different protocols.

Protocols and services are different, but they are close to each other. We must not confuse. A service is rather an abstract notion, although the protocol corresponds to what really happens. This distinction actually answers to modern programming and implementation requirements. It is equivalent to making the distinction between an algorithm and its implementation.

connection-oriented services and connectionless services

These services are simply those we tackled in the **previous chapiter**. It is then just a matter of reminding the main issues.

The **connection-oriented service** requires a connectio to be set between two points. The receiver then expects the sender to transmit data. At the end of the transmission, the connection is stopped. Such a service is for example the telephone: to use it, we must first take of the hook, and dial a number. The called person picks up the phone and the connection is then set. The two speakers converse until they hang up.

The **connectionless service** is characterized by the independance of transmitted messages. Someone can receive a message without being aware of it. Messages can then follow different routes. The consequence is that we can receive messages in an inverted order. The typical example for this kind of service is the mail system: someone writes a letter and sends it without warning the addressee. This letter may then arrive after a second letter the same guy may have sent to the same addressee. Routes followed by these two letters may be different.

printable format

concepts **the OSI model**