Computer Networks I

Cleto Martín

Apr 17, 2024

CONTENTS

This is the *private* documentation for the subject *Computer Networks I*, given in English language during of the 1st course of Degree in Computer Engineering at ESI (UCLM).

You can get the slides for the subject presentation here.

This documentation has been created with the following purposes:

- Be a helpful guide for somebody that needs to teach this subject.
- Unify the different pieces received from different people, mainly from previous teachers and the Spanish version.
- Keep the documentation in format that is easy to update and maintain.

The target audience of this document is specifically teachers. I hasn't been released publicly because it contains notes and resolved exercises and students are not meant to access them.

CHAPTER

ONE

INTRODUCTION

This lesson is an introduction to computer network concepts and the Internet. You can find the slides here.

1.1 What's a computer network?

A computer network is a set of *interconnected* computers that can *exchange* information. A computer network is usually represented like a graph where:

- The *nodes* may represent a *host* (like a PC or smartphone) or a *connecting device* (like a router or a switch). Nodes are considered *terminal* devices while the connecting devices are typically used to interconnect other nodes or other networks.
- The *links* represent a connection between the nodes. These links may be wired or through wireless signals. We will see different types of connections during this course.

Note: A network can be as small as 2 computers connected through a wired connection and it can be as large as millions of nodes.

A network can be measured with the following *criteria*:

- *Performance*: the network performance can be measured with in a different ways:
  - *Transit time*: the time needed by a message to travel between 2 nodes.
  - *Response time*: the time elapsed between a request and the arrival of its response.

  Note: *Throughput* (the amount of information during a period of time) and *delay* (the excessed time for a particular event) are often contradictory.

- *Reliability*: typically measured by the frequency of *failures* in the network and its *robustness* during failures.
- *Security*: whether the network is capable of protecting *unauthorised access*, *data loss* and *data manipulation*.

The information in a network flows between the nodes through the links. These information flows can be:

- *Simplex*: the information can only flow in one direction, from one node to another. For example: keyboards, TV signal.
- *Half-duplex*: the information can flow in both directions although only one node can transmit at certain time. For example: walkie-talkies.
- *Full-duplex*: the information can flow in both directions and both nodes can transmit at the same time. For example: telephone.

1.2 Network classification

There are different criteria on which one could classify a computer network.

1.2.1 By type of connection

The link is the connection between two nodes of the network. This link can be of two types:

- *Point-to-point*: the link is established between 2 nodes of the network and can only be used by them. They are typically wired although they can be wireless (e.g. the TV remote controller).
- *Multipoint*: the link is established between nodes and it is shared across all of them. Since this is a shared medium, it has some problems that need to be addressed that are not present in a point-to-point link:
  - Sharing the transmission medium and its bandwidth (*spatially* or *timely* shared).
  - Addressing: in order to send information, we will have to use some *addressing mechanism* so each node will be identified with an address. This mechanism may support the following variations:
    1. *Unicast*: the address identifies a single node in thet network.
    2. *Multicast*: the address identifies a subset of nodes of the network.
    3. *Broadcast*: the address identifies all nodes of the network.

1.2.2 By physical topology

There is also another way to classify a network using the concept of *physical topology*. It describes they way the network is laid out physically, how the links are arranged and connect different set nodes to each other. The topology can be seen as a *geometric representation* of the links and the linking devices to one another.

- *Mesh*: there is a point-to-point link from every node of the network to the rest of nodes. This means that for a network of `n` nodes, there will be `n ( n - 1) / 2` full-duplex links.
  - Higher robustness: if a link or node fails, there are alternative paths so the network can still function.
  - Higher security: communication happens in dedicated channels.
  - Higher cost: requires lot of links
- *Star*: there is a point-to-point link between the nodes to a centralised hub. This hub will be used as a shared linking device and will interconnect all nodes. So for a network with `n` nodes, there will be exactly `n` full-duplex links, each of them going from a node to the hub.
  - Lower cost: less complexity in terms of links required.
  - Lower robustness: the hub is a *single point of failure*.
  - Higher security: communication still happens in dedicated channels.
- *Bus*: there is a multipoint connection between nodes, a shared transmission medium (e.g. a cable called *backbone*) to which the nodes are attached (tapped).
  - Lower cost: less complexity in terms of links.
  - Limitation of the amount of nodes that can be attached.
  - Requires some kind of control on how to use the shared media (addressing, access, etc.)
  - In case of failure, localising the error can be challenging.
  - Security concerns about unauthorised access.

  - The shared medium is a single point of failure.

- *Ring*: each node of the network has 2 point-to-point connection to other 2 nodes on either side of it, forming a ring between all of them. The information flows in one direction reaching its destination jumping from one node to the following one in turn.
  - Easy and cheap to implement.
  - Each node can be now a potential single point of failure but the failure would be very easy to identify.
  - The unidirectional flow makes them slow.

1.2.3 By size

In terms of the network size, we can distinguish:

- *IPN*: InterPlanetary y Delay Tolerant Networks (DTN).
- Global (Internet).
- *WAN*: Wide Area Network – country or continents.
- *MAN*: Metropolitan Area Network – towns or neighbourhoods.
- *LAN*: Local Area Network – buildings or departments.
- *PAN*: Personal Area Network – computes or desktops.
- *SAN*: System Area Network – embedded systems.
- *NoC*: Network On Chip – networks in a integrated circuit.

We are going to focus in LAN and WAN.

LAN

A Local Area Network usually consists on hosts connected to one another. They can use a shared bus (multipoing is not really common nowdays) or having connecting device where each node has a point-to-point connection to it (e.g. a switch or router). Logically, it represents a network of nodes that can send information between them and node addresses identify each of them uniquely within the network. It is common that today's LANs can reach transmission rates of 10Gbps and typically use a *simple topology*.

LANs are not rarely used in isolation. Instead, they are connected to other LANs using WAN connections.

WAN

A Wide Area Network brings together connecting devices like routers and their associated hosts. The links of a WAN are typically point-to-point between network devices (not hosts) and it is used for connecting larger entities like organisations or countries.

There are 2 types of WANs:

- *Point-to-point WAN*: where 2 connecting devices are connected to bring 2 different networks together.
- *Switched WAN*: when we need to connect multiple point-to-point WANs together then we will need to use switches between the connecting devices. As we will see soon, this is the structure of the backbone of global communication today.

1.3 Inter-networks

This mix of LANs and WANs is very common today. It creates the notion of *inter-network* (or *internet*) where two or more networks are connected each other. Each network can use different technology and, as long they use the same communication rules, information can flow from and to different networks with no issues.

Note: Note the we are talking about an internet (with lowercase i) and as abstract concept.

As we introduced before, switched WANs are required when we need to connect multiple networks. It is required to form a *switched network* between networks in other to create an internet. In a switched network, data must be forwarded between networks somehow. It can be done using:

- *Circuit-switched* networks: the switches activate (or deactivate) connections between ends. This connections are established once and are kept active during all the data transmission until it is closed. This connection is called *circuit*.

  An example of this type of switched WAN is the "old" telephone system. The idea is that telephone terminals are connected to the switch and switches are connected with high-capacity links between them. Unless all telephone circutis are used at the same time, the high-capacity link will be not used at full capacity.

- *Packet-switched* networks: instead of opening connections between communication ends, the ideas is to break down the data into small *packets* that are transmited individually. This means that the packets can now be stored and sent later and organise the transmission differently than in a continous communication scenario.

  Computer networks (and also telephone systems nowdays) work using this approach. A *router* can receive packets from the hosts, queue/store them and send them to the other router individually. Even if the link between routers is at full capacity, the fact each packets can be stored and queued by the routers make the network still functional, although some *delays* might be introduced.

1.3.1 The Internet

The most notable internet is the Internet (with uppercase I) and it is composed of thousands of interconnected networks. The structure can be described as following:

- *Customer networks*: these are the end users that pay for a broadband connection for a home or an office.
- *Provider networks*: a user pay to an Internet Service Provider (ISP) to connect their customer network to the provider network. These providers (usually known as *regional ISPs*) pay for connecting their provider networks to the backbone networks of the Internet.
- *Backbone networks*: these are managed by some ISP (usually known as *international ISPs*) and they are interconnected using *peering points*, making that a message can be routed through the entire structure from one network to another.

There are different ways in which you could *access the Internet*:

- Using *telephone lines*: although not really common these days for end users, the telephone infrastructure can be used for accessing the Internet using a *modem* or a *DSL* line.
- Using *dedicated lines*: like fibre channel or TV cable, which is the most common way for end user's. It provides high-speed access compared to the telephone lines.
- Using *wireless lines*: there are remote locations in which case it is difficult to have wire-based technology. For these cases wireless lines like satellite links can be used to access the internet.
- Using *direct connection to the Internet*: for big organisations it is possible get a high-speed connection with a regional ISPs and connect its internets to the Internet.

Standards and administration

The Internet is currently organised and managed by the following bodies:

- *Internet Society* (ISOC): is the main Internet organisation that provides support for the Internet standards and procedures.
- *Internet Architecture Board* (IAB): is the technical advisor of ISOC, focused on the technical and researching matters of the Internet community. It has two main components:
  - *Internet Engineering Task Force* (IETF): is a forum of working group, managed by the *Internet Engineering Steering Group* (IESG) responsible of detecting problems and provide solutions to them. It is divided by *areas* like applications, protocols, architecture, etc. There are *working groups* on each area for developing solutions to specific topics.
  - *Internet Research Task Force* (IRTF): is a forum of working group, managed by the _Internet Research Steering Group (IRSG) focused on long-term research topics about new technologies, architectures, applications, etc.

The *Internet standards* are a very thoroughly tested specification for a particular functionality of feature of the Internet. These standards will be followed by the Internet community so users and services can interoperate.

Before of having an Internet standard fully approved, it must go through a review process:

1. An *Internet draft* is created in the first place. This in a work-in-progress, non-official document with a 6-month lifetime.
2. If authorities recommend it, the Internet draft is published as a *Request For Comments* (RFC) document. A number is assigned to them and the RFCs will be assigned with a *maturity level*:
   - *Proposed Standard*: at this level, the RFC has generated enough interest so some groups have started to test and implement it.
   - *Draft Standard*: if two or more independent and interoperable implementations have been done, the RFC will be elevated to this level. Typically the original RFC will be modified as problems are expected to be found.
   - *Internet Standard*: a draft standard reaches this level after a demonstration of successful implementation.
   - *Historic*: typically reserved for those RFC with historical interest like superseded features.
   - *Experimental*: RFCs that should not implemented on any Internet service in production.
   - *Informational*: RFCs usually created by vendors that contain general or tutorial information.

A brief history of the Internet

The Internet evolved from a private network to a global one in about 40 years. These some of the milestones of the Internet's history:

- **Before 1960**: during these years, computer networks were limited to some terminals connected to a mainframe by leased lines. The project RAND (Research ANd Development) started on 1947 with the aim of promoting the scientific knowledge for helping US security homeland. Some of the research results of this project will have impact on the birth of the Internet years later.
- **1961**: Leonard Kleinrock proposes packet switching.
- **1962**: J.C.R. Licklider published “On-Line Man Computer Communication”.
- **1962**: Licklider was hired by the Defense Advanced Research Projects Agency (DARPA) to “interconnect the Department of Defense (DoD) main computers at Cheyenne Mountain, Pentagon and SAC” (ARPAnet). Three network terminals were installed: Santa Monica, Berkeley and MIT.

- **1967**: Creation of ARPANET: a small network whose hosts would have an *Interface Message Processor* (IMP) to communicate with other hosts. The IMPs would be able to send and receive messages from other IMPs. Leonard Kleinrock proposes packet switching.
- **1968**: Douglas Engelbart and the Augmentation Research Center (ARC) present the oN-Line Ssystem (NLS), a computer collaboration system where multiple users could interact. That event was known later as "The mother of all demos". The NLS was the first system including graphic interface, desktop, icons, mouse, windows, hyperlinks and video-conference.
- **1969**: RFC 1 "Host software", specifying what is in the IMPs and what in hosts.
- **1969**: First ARPANET: 4 nodes in University of California at Los Angeles, University of California at Santa Barbara, Stanford Research Institute, and University of Utah.
- **1971**: ARPANET grows to 15 nodes. It worked with the Network Control Protocol (NCP).
- **1971**: Ray Tomlinson sends the first email.
- **1971**: File Transfer Protocol (FTP) is defined.
- **1973**: Robert E. Kahn and Vinton Cerf work in a common internetwork protocol, as an improvement of NCP.
- **1974**: RFC 675 "Specification of Internet Transmission Control Program", by Vinton Cerf.
- **1977**: An internet of 3 different networks (ARPANET, packet radio, and packet satellite) was demonstrated.
- **1981**: ARPANET has 213 nodes.
- **1981**: RFCs 791-793 define the basics of TCP/IP.
- **1983**: 1 January was "the flag day": TCP/IP replaced all earlier protocols in ARPANET.
- **1983**: Paul Mockapetris proposes the protocol Domain Name System (DNS).
- **1983**: Berkeley Unix 4.2BSD includes the socket API.
- **1984**: 4 Berkeley students write Berkeley Internet Name Domain (BIND), the DNS Unix first implementation.
- **1985**: IETF is created.
- **1989**: First ISP "TheWorld.com" in the US.
- **1991**: Tim Berners-Lee (at CERN) develop the first network based hypertext implementation and the HTTP protocol in the "WorldWideWeb" project.
- **1993**: University of Illinois creates the Mosaic graphical web browser.
- **1994**: Classless Inter-Domain Routing.
- **1998**: IPv6.
- **2008**: 1500 M-users.
- **2023**: 5200 M-users (90% in developed countries and 57% in developing countries)

Economy around the Internet

Nowdays the biggest and most-valued companies in the world has grown based on the existence of the Internet. These are a few examples of this type of companies:

- *Amazon*: founded in 1994, initially based on the e-commerce bussiness, runs Amazon Web Services (AWS) which is used by companies and developers for running lots of Internet services.
  - Revenue (2022): $513.98 billion
  - Employees (2022): 1,112,555
- *Alphabet*: founded in 2015 for re-structuring Google, the famous web search engine.
  - Revenue (2022): $282.8 billion
  - Employees (2022): 181,798
- *Meta*: founded in 2004 and originally as the social media Facebook, is an Internet-based company focused on communication and marketing research. It is also the owner of Instagram and WhatsApp, among others.
  - Revenue (2022): $116.61 billion
  - Employees (2023): 66,185
- *Microsoft*: founded in 1975 and they are currently focusing on its cloud service called Azure.
  - Revenue (2023): $211.9 billion
  - Employees (2023): 238,000
- *ByteDance*: founded in 2012, they are the owners of TikTok.
  - Revenue (2022): $85.2 billion
  - Employees (2023): 150,000

CHAPTER

TWO

NETWORK ARCHITECTURE

This lesson introduces the concept of protocol layering, showing ISO and TCP/IP models. You can find the slides here.

2.1 Introduction

A network architecture describes how a network is organised, its components and its *protocols* or communication rule set.

When the communication is simple we may need a simple *protocol* to reach an effective exchange of information between peers. However, automating this communication is usually a complex process that requires the implementation of specific rules (protocols) in order to be achieved. In fact, even a simple human conversation can have hidden complexity underneath:

- They should talk a language that both can understand.
- They should respect the other's turn before talking.
- It is expected to follow good manners. For example, say hello, ask for things kindly, farewell before leaving, etc.

This examples gets even more complicated if we introduce new variables:

- The speakers do not share a common language so they need some translation mechanism.
- The speakers are located apart from one another.

If we want to deal with complex communication problems, we will have to divide each problem into simpler functions or tasks that will collaborate each other in order to get the overall communication done. This is an intuitive idea of *protocol layering*: divide the communication problem into different phases which solve *a particular issue/problem* of the overall communication process and *delegate* the rest to other phases to other layers.

Note: The Philosopher's analogy is an illustrative example. See that each layer of the communication use a information useful at its own level. For example, the translator has a field called `L` for the language and the sender uses the fax number.

Protocol layering organises the communication mechanism in layers that:

- Are sorted by *abstraction level*, from the end user to the transmission medium.
- Require functionality from upper layers and provide it to the lower ones.
- At both ends, each layer can be seen as they maintain a virtual conversation that makes sense at the layer's level.

Note: Two principles of protocol layering:

1. Each layer should be able to perform two opposite tasks: for example send/receive, or encrypt/decrypt.

2. Each layer should share the same object at both ends of the communication. For example, an encryption layer should have a chiphertext at both ends.

2.2 Reference models

In this section two main reference protocol models are shown. They describe how different protocol layers can cooperate so a computer network can be implemented. Depending of the defined details, the model can be *more specific* by describing the detailed behaviour of each layer and the protocols used, or it can be *more general* defining the model conceptually.

2.2.1 OSI

The Open System Interconnection (OSI) model is a reference model created by the ISO by the late 1970s. This is a general and conceptual model that can be used for describing other computer communication models too.

This model defines 3 key components for each layer:

1. **Service**: the functionality that a layer provides.
2. **Interface**: the mechanism on how to get the service from a layer.
3. **Protocol**: the communication rule set between the communication ends of a layer.

From bottom to top, the layers of this model are:

1. **Physical**: it is in charged of transforming bits into electrical/mechanical signals and its reverse operation. This layer is very dependant of the transmission medium. Both ends of this layer send and receive *bits*.
2. **Data link**: provides data transmission between nodes that are directly connected (cable, air, etc.). The data transmission is grouped in set of bytes called *frames*. At this level nodes need to have some addressing mechanism and a common one, used in multiple types of networks, is the *Media Access Control* (MAC) address.
3. **Network**: provides data transmission between nodes that are not directly connected. At this level, a source node can send data to a target node and some intermediate nodes (*routers*) will help on retransmitting the message until it reaches its destination. The data is grouped in *datagrams* or *packets* and this layer also needs an addressing mechanism. Nodes will have a *logical address* and the most common one is the IP address.
4. **Transport**: provides a logical connection between two programs of different nodes. Both nodes can maintain multiple logical connections. The data is grouped in *segments* or *user datagrams* depending of the protocol used and it also requires an addressing mechanism which usually is a *port* number.
5. **Session**: provides a way to establish a session for a logical connection. The session provides the mechanism for a *reliable* communication and synchronisation between the nodes.
6. **Presentation**: it mainly provides the *data conversion* (also known *serialisation*) of complex data structures that need to be sent. It can also provide data encryption/decryption or compression/decompression.
7. **Application**: provides the highest-level communication between 2 programs and it is specific of the programs themselves (email, web browsers, remote commands, etc.)

2.2.2 TCP/IP

The TCP/IP model is published a few years after the first implementations where already in used so it was an attempt to formalise its use and development. It is a more specific model as it describes what protocols should be used at some levels although other layers are not fully defined. The semantics of each layer are very similar to the OSI one.

The layers of the TCP/IP model, from top to bottom:

1. **Application**: any protocol used by the 2 programs involved in the communication.
2. **Transport**: defines 2 possible transport protocols:
   - *Transport Control Protocol* (TCP)
   - *User Datagram Protocol* (UDP)
3. **Inter-network**: defines the *Internet Protocol* (IP) used for transmitting data between nodes across different networks.
4. **Host to net**: allow IP datagrams to be delivered (or rejected) to another host of the same network.

2.2.3 Hybrid model

The OSI and TCP/IP models are very similar. In fact, as the OSI model is so general, it can be simplified so it is adjusted to the TCP/IP model's layers. For terminology simplicity, in this course we are going to follow an hybrid model with 5 layers:

- Application
- Transport
- Network
- Data link
- Physical

2.3 Encapsulation and decapsulation

As we have seen intuitively in the previons sections, each layer groups data in different types. These groups are known as *Protocol Data Units* (PDUs). PDUs are built by *encapsulating* the data to be sent with extra information appended to the data. These extra information might be *headers* (when they are placed before the data) and *tails* (when they are placed after the data).

In our hybrid model, there are the following PDUs:

Layer	PDU	Header?	Tail?
Physical	Bits	N/A	N/A
Data Link	Frame	Yes	Yes
Network	Datagram	Yes	No
Transport	Segment	Yes	No
Application	Message	N/A	N/A

From top to bottom, the process of *encapsulation* is as follows:

1. The program builds the *message* it needs to send.

2. The message is split in *segments* each one with a transport header for sending them. This header will contain the transport address, usually port numbers (source and destination).
3. Each segments is split in network *datagrams* that will contain the logical addresses (source and destination) in a new header appended to it.
4. For each network datagram, a set of data link *frames* will be built and they will includy another header with the hardware adresses (source and destination). At this level, a tail is usually appended so the frame is built with a fixed size.
5. Each frame is analysed and each *bit* is transformed into electric signals to be sent over the transmission medium.

Note: It is time to show the example of sending `Redes I` message via the stack UDP/IP/Ethernet. As all values are shown in hexadecimal, it is important to show the ASCII table so the message's bytes can be understood.

2.4 Addressing fundamentals

We have seen 2 main addressing mechanisms: *physical* and *logical* addresses. The main differences are:

- The physical addresses are closer to the lower-level layers. The logical address are usually at the higher-level layers.
- The physical addresses are globally unique. The logical address is only unique in the network it belongs.
- The physical addresses is not meant to be changed at any time. The logical address can change frequently.

2.4.1 Physical address

For simplicity, in this course we are going to use *physical address*, *MAC address*, and *Ethernet address* indistinctly. All of them is going to reference to the physical address that identifies nodes that are *directly connected* (data-link layer). It is important to note that this addressing mechanism allows data exchange between nodes that are directly connected without any intermediate node or router.

Note: The example in the slides is a simplified data-link frame that is going to be sent to host `87` from the host `10`. All nodes are connected to the shared communication medium and only the destination host receives the frame.

2.4.2 Logical address

For simplicity, in this course we are going to use *logical address*, *network addresses* and *IP address* indistinctly. Both references to the address that identifies a node *within a network* (network layer). It is important to note that this addressing mechanism allows data exchange between nodes that are not connected directly and networking devices (routers) will help on routing datagrams through different networks.

Note: The example in the slides is a 3-LAN network with simplified physical and network addresses.

- The `A` node wants to send a datagram to node `P`.
- The frame is sent to the first router as the `20` physical address is used.
- The first router sees that the datagram is for `P` and, thanks to its routing tables, determines that it has to send it to the second router (physicall address `33`). It also changes the source physical address to its own.

- The second router sees that the datagram is for `P` and it knows that it belongs to its own network, so it just needs to use `95` as a destination address. It also changes the source physical address to its own.

2.5 Exercises

TBD

CHAPTER

THREE

APPLICATION LAYER

This lesson describes in depth the application layer and describes multiple application-layer protocols. You can find the slides here.

As it has already mentioned in the *architecture* chapter, the each protocol layer provides a certain service to the upper-level layer and requires functionality from the lower one. In this case, the application layer is at the top of the stack so:

1. It *provides* service to the user: from the user point of view, there is a *logical connection* between its application and the remote one as they were directly connected sharing messages. Obviously, this connection is actually performed by the lower-level layers down to the physical layer.
2. It *uses* the transport layer to send its *messages*: depending on the selected transport protocol the message deliveries could be guaranteed or not.

There are *public-domain* protocols, defined in RFCs, like HTTP or DNS, and *proprietary* protocols like Skype or WhatsApp. Since the public-domain ones are standardised and publicly available they are typically adopted for a better interoperability between different manufacturers.

The application-level protocol defines mainly the following features:

- **Messages**: the valid syntax and structure of messages (fields and values), the meaning of the messages (semantics), and the types of messages (e.g. requests, reponses, control, etc.)
- **Message processing**: those rules about how and when messages should be processed.

3.1 Paradigms

The application layer communicates two or more applications. The role of the applications that take during the data exchange defines the *paradigm* of the communication. There are 3 paradigms:

1. **Client-Server**: this is the most traditional one where the client contacts the server which is always on-line waiting for clients to connect. The client *initiates the connection* to server in order to perform a *request* and the server replies to that operation with a *response*:

    - Multiple clients can connect to the same server.
    - Clients cannot connect directly to each other.
    - The server has to be always reachable by clients.

    Applications using this paradigm usually provides 2 different programs: the client application and the server application.

2. **Peer-to-Peer**: the main problem with the client-server paradigm is that the server side becomes a *single-point-of-failure* (SPOF) and it often needs to be replicated. This replication might not be easy to implement in some cases.

The alternative paradigm is to have a network of *peers*, i.e. nodes that can *request* and *response* messages. This means that:

- There is not a central entity that needs to be always available.
- The system can grow just by adding nodes to the network but they are difficult to manage and control.

P2P systems are used in some limited context.

3. **Mixed**: some applications used both client-server and P2P paradigms at the same time, depending of the operation they are performing. For example, the lookup operations of available peers can be done with a client-server approach whereas the connection between peers would be done via P2P.

3.2 Transport requirements

The application layer requires service from the transport layer. The kind of service and the performance needed is determined by the own nature of the application. The following metrics are usually relevant for most of applications:

- *Data loss*: the amount of data that can be lost at different points of the communication path. Some applications tolerate some degree of data loss (e.g. video transmission) whereas others require 100% reliable data transfer (e.g. file transfer).
- *Bandwidth*: the amount of data transferred per time unit. Some applications requires a minimum of bandwidth to be "usable" (e.g. videoconferencing) whereas others can adapt themselves to changes in available bandwidth (e.g. modern video players).
- *Timing* or *delay*: the amount of time between a request and its response. Some applications requires a very limited delay to be "usable" (e.g. games) whereas others might tolerate higher delays (e.g. messaging).

Note: Show the table with different types of applications and their requirements of the transport layer

The Internet protocol stack provides 2 protocols: TCP and UDP. These protocols provide the following services:

- *TCP service*: provided by the TCP protocol and has the following features:
  1. *Connection-oriented*: a virtual connection is established between client and server. This connection is created before client and server start to exchange transport messages.
  2. *Reliable*: there is some level of guarantee that messages will arrive to destination.
  3. *Flow control*: so the sender cannot overwhelm the receiver.
  4. *Congestion control*: so the data flow is adapted by if the network is overloaded.

  Note that TCP does not resolve some communication issues like *timing* or *guaranteed bandwidth*.
- *UDP service*: it does *not* provide any of the TCP service features, so it is not reliable.

We will study them in depth in the *transport* chapter.

Note: Show the table with different protocols and the transport layer associated to them.

3.3 Implementation example

One common way to implement applications that require communication is using the *socket interface*. Sockets are an abstraction provided by the operating system for sending and receiving data using by using the transport layer. The developer can choose what type of transport layer to use and many other parameters of the communication.

Note: You can show a trivial client and server applications. Students do not need to understand everything, just the overall process and get in contact with the concept of socket.

Firstly run:

python server.py

And in a different terminal:

python client.py

3.4 Web and HTTP

The World Wide Web (WWW or Web) is a distributed service that provides access to documents called web pages. The web page content has evolved from plain text to multimedia content like videos, audios, etc. These two features have made the web to be highly scalable:

- Highly *distributed*: they can be placed in different physical locations. The place where web pages are located is called *web sites*.
- The are *linked*: they can refer to each other, even though they are not placed is the same server. The concept from 1963 of *hypertext*, where a document can refer to another and the reader can access to it directly, has evolved to *hypermedia* because the linked objects are not only text anymore (video, audio, etc.)

It was originally created by Tim Berners-Lee in 1989 at CERN for allowing researchers to access scientific content from other researchers.

3.4.1 Web browser

In order to get access to web pages and their content a special application is needed: a *web browser* or *web client*. This program is able to request and receive content that is shown to the user. It also handles the user input and transform it into requests.

Note: Show the 3 components of the web browser. They work as follows:

1. The *controller* handles the user input and transform it into specific actions of the client programs.
2. Each client program knows how to access a specific *protocol*, so if the user requested a document via HTTP, the HTTP client will be used to get it.
3. Once the document is retrieved, the controller uses the *interpreters* to render that document on the screen.

Web browsers handle 3 types of web pages:

- *Static* pages: these documents that are stored in the server and the client gets a copy as they are. Typical static files are HTML, XML, XSL, XHTML, etc.

Note: Good things:

- They are simple and fast.
- Low workload for the server.

Bad things:

- No dynamic content which limits application scopes.
- Hard to maintain as the web site grows.

- *Dynamic* pages: when they are requested by the client, the server *generates* the content that is returned to the user. Each request might generate different content.

Note: Good things:

- Content can be programmatically generated.

Bad things:

- High workload for the server.
- Slower than static files.

- *Active* pages: when they are requested by the client, the server responds with a program that will be executed by the web browser. This program will finally generate the web page that will be show to the user.

Note:

- Good things:
  1. Content can be programmatically generated.
  2. Low workload for the server.
- Bad things:
  1. Web browsers need to run programs (become interpreters).
  2. Security and privacy concerns.

Web pages can be referenced in the web browser using a Uniform Resource Locator (URL). A URL is a general mechanism for referencing objects in a structured way. This is its format:

```
<method>://<host>[:<port>][/<path>]
```

Where:

- `method` is the protocol to use. For example, `http` or `ftp`.
- `host` is the IP or the host name of the server. For example, `google.com` or `www.hola.es`.
- `port` is the destination port of the transport communication. By default is `80` for `http`.
- `path` is the place of the target object. If not specified, it will take the root document (typically `index.html` in `http`).

For example:

```
http://www.someschool.edu/someDept/pic.gif
ftp://my-server:631/contacts.csv
```

3.4.2 The HTTP protocol

The HyperText Transfer Protocol (HTTP) is a protocol that origianlly was created for retrieving web pages. It defines how to a client and a server have to be implemented in order to provide all the required operations. This protocol uses:

- TCP as transport protocol.
- The HTTP server will be listen at port 80.

The clients sends *HTTP requests* to the server and this returns *HTTP responses*. For example, a web browser exhanges HTTP messages with a web server to retrieve web pages.

The protocol HTTP is *stateless* meaning the server will not maintain information about past requests from clients. HTTP can be configured to use TCP in two different modes:

1. **Nonpersistent HTTP**: for each web page object, there will be a new TCP connection. This means that each HTTP request-response will generate a new TCP connection each time.

   Note: The example is a file HTML that includes an image. There are 2 different TCP connections due to it is not persistent. Note that each TCP connection needs the 3-way handshake to be established, which is expensive if there are lot of objects.

2. **Persistent HTTP**: multiple web page objects can be sent over the same TCP connection. This reduces the amount of overhead required for each connection but might increases the amount of connections that a single server holds for a period of time.

   Note: The example now is much simpler than the nonpersistent one

The HTTP procotol defines 2 types of messages:

- **Requests**: messages sent from clients to the servers. They have 4 fields:
  1. *Request line*:
     - Method: we will see them in depth a bit later.
     - URL: the targeted object of the request.
     - Version: the HTTP protocol version to use.
  2. *Header lines*: headers are used to provide options and more context to the request. It can be empty.
  3. *Blank line*: just a line break made of 2 characters `CR` and `LF`.
  4. *Body*: if the requests require to have data to be sent to the server, the body may be the place for them.

  Note: Show the example of the HTTP request parts. The `Connection` header indicates that this is a nonpersistent request.

  The main request methods are:

  1. `GET`: retrieve a specific object or resource. Its body is body is empty but it can pass some information to the server using:

- The headers.
- The URL parameters e.g. `http://uclm.es/info?edificio=1&campus=CR`

2. `HEAD`: similar to `GET` but only retrieve metadata about the resource, like the last time the resource was modified.
3. `PUT`: sends a document to the server which is the reverse of `GET`.
4. `POST`: sends some information that the server should add or manipulate, like form data.

Note: Show the table of HTTP request headers. Mention that `Cookie` will be seen in depth.

Also comment that the request headers shown in the slides are just of a subset of the standard ones. There can be custom headers defined by the application themselves.

Apart from the request headers we have seen, The headers might also be

- **Responses**: messages sent by the servers to the clients.

  They have 4 fields:

  1. *Response line*:
     - Version: the HTTP protocol version to use.
     - Status code: an integer representing the response type. We will see them in depth a bit later.
     - Phrase: a string version of the status code.
  2. *Header lines*: headers are used to provide more context to the client in the response. It can be empty.
  3. *Blank line*: just a line break made of 2 characters `CR` and `LF`.
  4. *Body*: if the reponse includes data the body may be the place for them.

  The status codes are 3-digit integers whose meaning is divided in groups:

  - `2XX`: successful group. `200` means the request was successful.
  - `3XX`: redirection group. `301` means the requested object was moved to another location, specified by the `Location` reponse header.
  - `4XX`: client-side error group: `400` means the request was not valid and `404` means the requested object was not found.
  - `5XX`: server-side error group: `500` means a general server error happened and `505` means the server does not support the specified HTTTP version.

Note: Comment that response headers shown in the slides are just of a subset of the standard ones. There can be custom headers defined by the application themselves.

About the following examples of the slides:

1. The HTTP GET example shows a GET request for the object `/usr/bin/image1` using HTTP version 1.1. The tells to the server that it accepts both `gif` and `jpeg` images. The server's reponse contains the image with a few more metadata: the server that relied, the image size, the format in which the image is encoded, etc.
2. `httpbin.org` is a dummy HTTP server that provides information about the requests it receives. It can be useful for testing HTTP request development. In this example we are using it to create a request using *netcat*. `nc` will create a TCP connection to port 80 and send the data we type on the console. When a blank line is introduced, the GET request will be made.

Note that the connection is kept open. If you use `Connection: close` as header, after the request the connection will be closed.

3. `curl` is a command-line tool for interacting with HTTP servers. It helps to create debug and create scripts that need to access resources via HTTP (download a file, request for a service, etc.). In this example `curl` recevies a `301` meaning that `google.es` is actually `www.google.es`, `curl` creates another request to the new URL automatically.

   You can show this step by step if you firstly don't use `-L` flag. That will make `curl` not following the new location automatically.

4. In the HTTP PUT example the client requests the execution of a CGI `/cgi-bin/doc.pl` (a Perl script). In this case, the request does contain input data (50 bytes) for the script.

   The server runs the script with the input data and generates a response with a dynamic object that goes in the response body (2000 bytes).

5. Again, `nc` can be used for creating a PUT request. Note that if `Content-Length` is not provided, the server will response as soon as the first blank line is sent.

3.4.3 Cookies

As we previously said, the HTTP protocol is stateless. This means that the communication parts need to track any context or state related to the communication. For example, the server does not recall that a certain client already made a related request before.

The use of HTTP in many different types of applications has forced the introduction to keep some context or state. For example, a shopping cart or authenticated users of a given web page are examples that having some kind of context will improve notabily the user experience.

A way to implement this stateful mechanism is using *cookies*. A cookie is usually a string value that encondes information about the user and their context. For example it can include the domain name, user name, timestamps, etc. It is *created by the server* and passed to the client for them to store it (in their file system) during the client's first request. The idea the client will send the cookie along their requests os the server can identify them and get the previous context.

Note: Explain the toy store example: the shopping cart is implemented using a cookie. Note that the headers used are `Set-Cookie` and `Cookie`.

You can also show a cookie analyser like this. Search for `http://uclm.es` there and you will see the cookies stored by the user.

3.4.4 Proxies and web caches

We have seen a direct communication bewteen clients and servers so far. However, it is possible to have intermmediate HTTP nodes that will act on behalf of the clients to contact the servers. These nodes are called *proxies* and they are useful in multiple scenarios:

- *Security and audit*: some work environments require to be secure and confidential. The use of proxies is useful for auditing the activity of web browsing as well for applying policies for access restrictions (forbidden web sites, user access control, etc.).

  Note: Show the `curl` example with `api.ipify.org`. The returned IP is the public IP from where we requested the HTTP GET. When we use a proxy with `http_proxy` envarionment variable, the returned IP will be the proxy one instead of the previous one. If the example proxy IP does not work you can get different one from this list.

- *Caching content*: which improves the overall user experience and reduces the shared bandwidth (typically at WAN links).

Note: Show the diagram where 2 clients use a proxy as a web cache. The first one requests an object that is *not* cached, so the proxy has to retrieve it from source. The second one requests an object that is cached so the response is much faster by saving an external call.

The main reasons for using a web cache are:

1. Improve client's experience as the response time is reduce overall.
2. Reduce the use of the *access link*, i.e. the WAN connection that is typically shared across the organisation to access the Internet.
3. Helps to those content providers with lower resources to effectively deliver content.

Note: Explain the numeric example of how a web cache improves the user experience without increasing the cost too much:

1. The response time for this case is `4.01` seconds.
2. Increasing the bandwidth of the access link will reduce the response time to `3.1` seconds. This is usually an expensive upgrade.
3. Using a web cache with a 40% chance of hits and no access link upgrade, the average response time will be `3.6`.

3.5 TLS and HTTPS

Nowadays, HTTP is used in very different environments. With more application being capable of providing services on the Internet, the need of having *secure communication* between clients and servers have been important in the recent years.

Think about the following applications:

- A bank's web application where users can make payments.
- Transport-related applications like a place for buying flight tickets.
- Email web portals where people can receive and send emails from their accounts.

These applications need to be protected from authorised use and HTTP nor TCP provide any help for this. This is where protocols like Transport Layer Secure (TLS) or Secure Sockets Layer (SSL) are designed for. The first one is the most recent version of the latter. It can be used by any application protocol but when it is used in conjunction with HTTP it is known as HTTPS. If a server supports HTTP with TLS, the URL changes from `http://` to `https://` and the protocol changes from `80` to `443`.

TLS provides the following services:

- *Fragmentation*: application messages are divided in fragments of 2^{14} bytes.
- *Compression* (optional): fragments are compressed with the algorithm negotiated by client and server.
- *Integrity*: ensures that data is not modified on-transit.

- *Confidentiality*: data is encrypted so unauthorised access is not possible.
- *Framing*: a header is added to the encrypted payload creating a *frame* which is passed to the transport layer.

These are important given the shared nature of the Internet: there are many places where information can be accessed by unauthorised actors.

TLS defines 4 protocols to achieve all these services:

1. *Handshake protocol*: client and servers need to agree on TLS communication details like what algorithms to use, what type of encryption, etc. Both also need to authenticate each other. They have to make sure that the other part is actually a legit one.
2. *ChangeCipherSpec protocol*: this protocol is used for signalling to client and server that when it is time to use for real the parameters and secrets exchanged during the handshake.
3. *Alert protocol*: when an error is detected, client and server will use this protocol.
4. *Record protocol*: all messages from the previous protocols will be processed by this layer that will transform the TLS messages into encrypted and signed frames.

Note: The example is just a TLS handshake to `google.es`. The real output is longer because it includes the certificates from the server. We do not need to explain them in detail, just mention that it is a way to authenticate that the server we are hitting is actually a Google's one.

The result of the TLS handshake is correct and some of the encryption parameters are shown.

3.6 DNS

One important part of an URL is the *host name* or *host address*. This can be the IP of the server but human users are not good at remembering sequences of 4 bytes. The Domain Name Service (DNS) is what helps in this situation: it provides a mapping between a host name made of characters to the corresponding IP addresses. In this context, we will say that *a host name is resolved to its IP* when a host name is translated to its corresponding IP address. For example `google.es -> 142.250.200.131`.

DNS can be seen as a *distributed and hierarchical database* which is accessible via *name servers*. These name servers communicate with clients and routers using the *DNS protocol* for resolving names. The service is distributed because it cannot be centralised for the following reasons:

- Avoid single points of faiure.
- If used globally, the amount of traffic will be huge to be handled by a centralised service.
- Distance between clients and the central server will be different depending on the client's location.

Apart from name resolving, DNS provides the following services too:

- *Host aliases*: a server can be accessed by an *alias* of a *canonical* name.
- *Mail*: it contains information about mail servers and its aliases.
- *Load balancing*: there can be multiple IPs assigned to the same name.

3.6.1 Hierarchical structure

DNS uses a *hierarchical* name space. This means that DNS names are structured in a specific way so each part of the structure corresponds to a certain level of the DNS server structure.

The structure of DNS names can be seen like a *tree*:

- Each node in the tree has a *label* no longer than 63 characters.
- The root node has an empty label.
- A node can be referenced by its *domain name*, the path of labels separated by dots. If the domain name goes from the target node to the root node is known as *fully qualified domain name* (FQDN).

Note: In the example of `pc1.dept.uni.edu.`, we can show that:

- That's a FQDN.
- We have 5 labels: `pc1`, `dept`, `uni`, `edu` and the empty label of the root node.
- We have 4 domain names: `pc1.`, `dept.`, `uni.` and `edu.`.

It is good to say that a *domain* is any subtree of the name space. For example: `dept.uni.edu.` is a domain.

The DNS servers are structured in a very similar way so they can be scalable to a large distributed system. Each node of the tree might have a corresponding server that provides the information of all the nodes under that domain name. This is known as *zone* and the servers responsible of a zone create a *zone file* with a list of all nodes under that domain.

Note: In the example, the `com` zone is huge so if the com DNS servers need to create the list of all nodes under this domain would not be scalable. For that reason, subdomains like `yahoo.com` or `amazom.com` will hold the zone file for their own domains. That way the `com` server can refer to any of the subdomains if more detailed information is needed.

3.6.2 DNS servers

- *Root servers*: there are 13 root name servers worldwide, each of them named after a letter (from A to M). They do not really store too much information as the details are delegated to other subdomain servers.
- *Top-Level Domain servers*: these are placed at the following level of the root servers and they are responsible of domains like `com`, `edu`, etc. and also top-level country domains like `es`, `uk`, etc. They usually do not hold detailed information about other domains of their zone.
- *Authoritative DNS servers*: these servers are contacted by TLD ones as they are delegated to managed a subdomain and know all the hosts within their zone. For example, `google.com.` has `ns1.google.com` as one of the authoritative servers for their domain.
- *Local name server*: it is the name server that DNS clients will contact directly when they need to create a DNS query for resolving names. It can be maintained by an ISP or an organisation.

3.6.3 DNS resolution mechanism

When a DNS client creates a query for a specific DNS name, it will firstly contact to its local name server and, if it does *not* know the answer, it will trigger the query to the rest of the DNS network. At that point, two strategies are possible: *recursive* or *iterative*.

Note: The example shows two ways on how the name resolution of `gaia.cs.umass.edu` can take place:

1. *Recursive*: the local DNS server contacts a root server. The root server does not know about `gaia.cs.umass.edu` and asks for more information to `.edu` TLD server. The TLD server does not know about `gaia.cs.umass.edu` but it knows that for `cs.umass.edu` can contact to `dns.cs.umass.edu` authoritative server so it asks for it. This final authoritative server knows the answer for `gaia.cs.umass.edu` so the response goes back through all the previous path.
2. *Iterative*: the order of the DNS queries are the same but it is performed by the local name server instead by each DNS server in turn. For that to be possible, those servers that do not know the answer for `gaia.cs.umass.edu` will have to provide a server to ask instead.

Recursive queries clearly *simplifies the implementation* of the local server but puts a *burden* on the root and TLD servers. Iterative queries makes the local name server to work more on each query and remove the pressure from the root and TLD servers. This approach is more interesting for a global service like DNS.

In both cases, for DNS to scale and function properly in a global context, it is important that responses can be *cached*. Caching responses for a period of time known as time-to-live (TTL) will reduce the amount of queries in the DNS network. For example, the TLD servers are usually cached by the local name servers so the root name servers are not queried very often.

3.6.4 DNS records

As we said before, DNS is a distributed database. This database defines different types of *records* that will hold information about the nodes. They are stored as *resource records* (RR) with the following format:

```
(domain name, type, class, ttl, value)
```

Where:

- *Domain name*: identifies the resource record.
- *Type*: the way the value needs to be interpreted (we will see more later).
- *Class*: can be different values but we are only interested in `IN` (means Internet).
- *Value*: the value of the record.

These are some types:

- `A` record: defines the IP for a given hostname,
- `NS` record: defines the authoritative name server for a given domain.
- `CNAME` record: defines an alias for a canonical name.
- `MX` record: defines the mail server for a given name.
- `SOA` record: provides information of a zone and domain.

3.6.5 DNS messages

The messages can be *queries* or *replies*. Both use exactly the same message format:

- *Header*: 6 fields of 2 bytes each (12 bytes):
  - `identification`: an 2-byte integer that will be generated during the query and will be used in the replies to identify the original query.
  - `flags`: a set of bits representing different conditions of the message. For example:
    - *Is this a query or reply?*
    - *Is this reply authoritative?*
    - Error status.
  - The rest of the fields depend on if we send a query or receive a reply and point to the next variable section.
- *Content*: these fields might include the questions for queries and the responses for the replies.

3.6.6 Examples

Note: The following examples will be used for a better understanding:

1. Adding a new DNS record like `redes1-esi.es` requires that an entity that is authorised for it (a *registrar*) updates the TLD servers with new information. In this example we use a domain provider like IONOS to:
   - Register our domain `redes1-esi.es`.
   - Point `www.redes1-esi.es` to our web server at `10.20.30.40`.

   Once it is done, people could access our new web site by doing the normal resolve mechanism:
   - `www.redes1-esi.es` will contact to the `.es` TLD server.
   - The TLD server will say it does not know about it but we should contact `ns1029.ui-dns.de` for getting more information.
   - The DNS client will contact this new name server and it will return the answer thanks to the A record it holds.
2. Show `/etc/resvol.conf` and `host` command.
3. Show `dig` command. The `PTR` record provides the domain name associated to an IP.

3.7 SSH

The Secure SHell (SSH) protocol is was originally created for replacing the insecure protocol TELNET. The main use of this protocol is to open terminal sessions on remote servers (commands `ssh` or `PuTTy`). This is very useful, for example, for maintaining infrastructure remotely without the need of being physcailly present. However, this protocol is general enough so it can be used for other purposes:

- File transfer: `sftp` and `scp` are applications that can transfer files securely over an insecure transport.
- Port forwarding: this is a generalisation of sending and receiving data over a secure channel. If we want to communicate 2 applications and provide a secure transport between them we can:
  1. Create an SSH tunnel between both nodes.
  2. Configure the end of each tunnel so the data received on through it will be redirected to the application ports.

This way, it is possible to encrypt any type of traffic (HTTP, DNS, etc.).

The SSH stack defines 3 protocols:

1. *SSH-TRANS*: this layer is closed to the transport layer and is in charge of creating a secured transport layer. It will provide:
   - *Confidentiality* and *integrity* of the data transmitted.
   - *Compression* which improves the performance of the communication.
   - *Server authentication* that allows the client to ensure the server is actually who claims it is.
2. *SSH-AUTH*: once the secure transport is in place, SSH-AUTH allows client and server authenticate each other. Client initiates its authentication and based on the protocol negotiated it may or may not get access to the server. The mechanism is very similar to TLS.
3. *SSH-CONN*: once the client and server are authenticated and ready for starting to exchange data, this layer helps the client to create communication *channels*. Each channel can be used for one purpose (remote command execution, file transfer, etc.). This layer multiplexes data between the available channels.

Note: The SSH example requires to run a local SSH server. It can be done using the following command:

docker run –rm -e PASSWORD_ACCESS=true -e USER_PASSWORD=redes1 -e USER_NAME=redes1 -ti linuxserver/openssh-server

It would be good to explain briefly:

- The first run just request the user name and password.
- The second run show the verbose output where some part of the negotations between client and server is shown.
- If we stop the SSH server and restart it again, the host key has changed so SSH will warn us that the server is not the same as we firstly contacted.

3.8 P2P file sharing

Peer-to-peer is a different paradigm of application communication. Instead of having a clearly separated role like client-server, each node of the network can act as a client or a server *at any time*. This is the reason because the nodes are known as *peers*.

P2P is used in multiple applications these days:

- File sharing.
- Voice communication.
- Blockchain.

File sharing services are commonly implemented using a P2P.

Note: Show the P2P example about Alice and Bod file sharing. The key points to show are:

1. All nodes can be servers so this type of communication can be *highly scalable* and provide *high availiability*.
2. The problem with this approach is the quality of the content distributed by the peers. It might be the case that some files are not fully available as there are parts none of the peers provide.

When peers ask for an object to the network, they need to know something how to locate other peers. A possible solution is to use a *centralised directory* (as Napster was implemented originally). The idea is each peer will inform to a central server about how to be contacted and the content they can server. The problem with this approach is that the central directory is now a *single point of failure*.

Another approach is to have multiple ways to maintain this peer directory. This is the case of BitTorrent, where people can distribute metadata files (`.torrent`) that hold information about peers and their content. On top of this, there are also multiple *tracker servers* that keep track of the peers as well. This approach avoids the single point of failure and provide high availability to the peer directory.

BitTorrent also improves the P2P file sharing service by making the following assumptions:

1. All files are broken in blocks of 256KB. Shorter sharing blocks allow peers to enter and leave the network more often without causing too much interruption to others.
2. Those blocks that are not common are requested first in order to increase their availability. That way, clients will balance the availability of the content evenly.
3. The distribution of the metadata files is not managed by the P2P network itself. It needs to be done with some other mechanism.

   The metadata file usually contains:

   - The name of the file.
   - The total size of the file.
   - The URL to a tracker.
   - The length of each piece (usually 256KB).
   - A list of hashes: one hash per piece.

In order to encourage clients to share files, BitTorrent defines a client relationship policy *tit-for-tat*: at first, the client is collaborative but later it will replicate to others what others do to it. If the client shares lot of files, it will become a *seeder* which will have better reputation across the network. However if the client do not share much and only downloads from others, it will become a *leech* which will decrease its reputation.

3.9 Exercises

TBD

CHAPTER

FOUR

TRANSPORT LAYER

This lesson describes in depth the transport layer and describes some protocols for reliable data transmission. You can find the slides here.

The transport layer provides a way to communicate *processes* that are located in different places. These means that two applications can send and receive messages as they were directly connected. The transport layer uses the network layer to transmit information between hosts.

Note: It is important to remark the difference between the transport and network layer: the network layer's domain is the hosts whereas the transport layer's domain is the processes (applications) that run on these hosts.

In this section we are going to study the services provided by the transport layer to the application layer.

4.1 Services

In this section some general concepts and transport-layer services are described. These services are provided to the application layer.

4.1.1 Port addressing and sockets

As it has been already mentioned, the transport layer communicates processes running on hosts. Because an application could send data to one or more applications on another host, some kind of process addressing would be required. The transport layer uses *port numbers* for distinguish applications within a server.

Using the client-server paradigm:

- A server application uses a *fixed* port so all clients can contact to it. It is usually said that *an application is listening on port X* to describe
- A client application uses an *ephemeral* port: its existence only lasts while communication takes place and can be changed the next time.

Although we have not get into the *network layer*, every hosts has its own network address (usually an IP address). The port number is a *higher-level* address as it points to an application running on the host:

- The IP address selects the host.
- The port number selects the application of the host.

The combination of IP address + port number is known as *socket address*.

Note: This is what we used in our example of socket programming: `s.connect((HOST, PORT))`.

The standard organisation ICANN has defined the following port ranges:

Name	Range	Description
Well-known	0-1023	Controlled by IANA
Registered	1024-49151	Registered by IANA but not controlled
Dynamic	49152-65535	Temporary and private ports

Note: Explore `/etc/services` and show the different applications associated with the port numbers. Also mention that there can be 2 ports for a speific applications (TCP and UDP).

Note: Explain the example of using `nc` for creating a local TCP server and listen on port 12345. With `netstat` you can explore the processes that are currently listening:

- `-l`: shows only those processes in `LISTEN` status.
- `-p`: shows the program names.
- `-t`: shows only TCP.
- `-n`: shows IPs instead of hostnames.

4.1.2 Encapsulation and decapsulation

As any intermediate layer, the transport layer receives messages from the application layer and from the network layer:

- *Encapsulation*: application messages are transformed into *segments* or *datagrams* (depending on the transport protocol we use). The application message (the *payload*) is appended with a *transport header* that describes its content and has useful information for the receiver to consume. This process happens on the *sender* side.
- *Decapsulation*: the transport layer processes each segments or datagrams coming from the network layer. It removes the transport header and passes the payload to the application network. This process happens on the *receiver* side.

4.1.3 Multiplexing and demultiplexing

Multiple processes can be communicating all at the same time, exchanging information with different remote processes. The transport layer needs to be able to perform the following operations in order to achieve this:

- *Multiplexing*: *many* processes will use the transport layer as *one* logical channel. This is performed at the sender side.
- *Demultiplexing*: the information that comes from the transport layer's logical channel needs to be deliver to the right process. This is performed at the receiver side.

4.1.4 Flow control

It is important to control the amount of data a sender can *produce* and a receiver can *consume*. There needs to be a *balance* between them:

- If the sender produces too few data, the receiver might be idle.
- If the sender produces too much data, the receiver can be overwhelmed.
- Circumstances might change as communication progresses (e.g. the receiver might get slower because of it is busy with something else).

In general, a process may deliver data in 2 ways:

- *Pull*: the receiver *requests* for more data to the sender. In this case, there is no need of flow control mechanism as the receiver can request data as needed.
- *Push*: the sender delivers the data *as it is produced*. This mechanism requires some kind of control as the receiver can be overwhelmed or being idle.

This pull and push model is used differently across the components of the transport layer:

- *Between the sender application and the transport layer*: the application *pushes* data to the transport layer, so there is a flow control mechanism between them.
- *Between the sender's transport layer and the receiver's transport layer*: when segments/datagrams are ready, the sender's transport layer *pushes* them to the receiver's one. Another flow control mechanism is needed here too.
- *Between the receiver's transport layer and the receiver application*: the receiver application *pulls* data from the its transport layer. In this case, a flow control mechanism is not required as the application consumes data on-demand.

The transport layer provides this flow control mechanism as we will see by the end of this chapter.

4.1.5 Error control

The transport layer might work on top of an *unreliable network layer*. That's the case, for example, for the TCP/IP stack. For this reason, the transport layer provides a way to detect errors of the data transmitted. Specifically, it has to deal with:

1. *Corruption detection*:data might be modified during their trip across the network. The transport layer provides a mechanism for detecting this type of data and discard them.
2. *Loss detection*: data packets might be lost so the transport layer will provide a way to detect this loss and request for being resent.
3. *Duplicate detection*: data packets might arrive duplicated for multiple reasons so the transport layer will discard any duplicated packet.
4. *Out-of-order detection*: data packets might arrive in different order due to network conditions so the transport layer will detect it and re-order them properly.

All these are achieved using these mechanisms:

- *Checksums*: numerical values generated from the sent data. They can be used by the receiver for checking that the data haven't been altered.
- *Sequence numbers*: packets are tagged with a increasing number so the order can be checked by the receiver.
- *Acknowledgement*: when packets are received, the receiver will have to acknowledge its arrival to the sender. Thus, the sender knows which packets arrived and which ones did not.
- *Timers*: as the network is not reliable, sender and receiver will use different timers that will expire if data are not acknowledged on time or expected replies never arrive.

4.1.6 Congestion control

Congestion may happen on any system shared by different users that try to make use of it at the same time. A congested network is a network where nodes *need to wait* for transmitting data.

The networking nodes (e.g. routers) use queues internally for dispatching packets to different routes. These queues can be full if the networking nodes cannot consume packets at higher rate than they arrive. The problem usually happens at network level and it manifests at transport layer too.

4.1.7 Connectionless and connection-oriented protocols

Since the transport layer might work on different context and applications, it is possible that not all features might be required because another layer is already providing them or they are not required for a particular context. For these reason, the transport layer might be:

- *Connectionless*: the data is sent in *atomic chunks*. These chunks, known as *datagrams*, do not have any kind of relationship between each other. They are individual packets that the receiver will treat as separated units. The datagrams are sent in order but the transport layer *will not* guarantee that the packets arrive in order to the application or prevent their loss.

  This type of transport-layer protocols just provide a simple mechanism of sending packets from one application to another, without really providing extra services. This is the case of UDP.

  Note: In the example, the packets arrive out of order and they are passed to the receiver's application as they arrive, without any flwo or error control

- *Connection-oriented*: the data is sent through a *logical connection* between the sender and receiver. The connection should be *established* before sending data and has to be *closed* afterwards. The packets transported through a specific connection are related each other: they belong to the same context of communication.

  This type of transport-layer protocols provide a more sophisticated set of services and can be used over unreliable layers like the network layer as flow and error controls are implemented. This is the case of TCP.

4.2 Transport-layer protocols in TCP/IP suite

TCP/IP protocol stack defines 3 transport protocols that can be used:

- UDP: User Datagram Procotol (connectionless)
- TCP: Transmission Control Protocol (connection-oriented)
- SCTP: Stream Control Transmission Protocol (connection-oriented that supports multiple redundant paths).

We are going to focus on UDP and TCP. SCTP offers more complex services than TCP and it is used only in few context.

4.2.1 UDP

The User Datagram Protocol (UDP) is the *connectionless* transport protocol included in the TCP/IP protocol suite. It should be consider *unreliable* so it is up to the application layer to provide reliability if it is required. This protocol is useful when a minimal overhead is required (e.g. real-time applications) and reliability/control is not strongly required (e.g. video/audio transmission).

UDP uses *user datagrams* (or just *datagrams*) as packets. It has a fixed header of 8 bytes:

- 2 bytes for the *source port*.
- 2 bytes for the *destination port*.
- 2 bytes for the *total length* of the data. This means that a datagram can carry up-to 64KB (2^{16} bytes).
- 2 bytes for an *optional checksum*:
  - If checksum is not desired, then it will be filled with 0s.
  - Checksum is calculated by adding all together in chunks of 16-bits the following:
    1. A *pseudo-header* of source IP, destination IP, protocol type and UDP length.
    2. The UDP header.
    3. The UDP data.

In essence, UDP provides no more than process-to-process communication on top of of the network layer:

- It is connectionless.
- No flow control.
- No error control although the optional checksum might be used by the application layer.
- No congestion control.

When an application requires a UDP port, the operating system creates 2 queues: *incoming* and *outgoing*.

Note: Show some of the well-known UDP protocols.

In Debian, you can install `inetsim` which includes many fake servers for different UDP protocols:

```
sudo apt install inetsim
```

Make sure that the Echo UDP service is running:

```
sudo service inetsim status | grep echo
```

In one terminal, you can show the log of `inetsim` so people can see the actions taken by each server:

```
sudo tail -f /var/log/inetsim/service.log
```

Then you can use the following to show it working:

```
$ nc -u 127.0.0.1 7 Hello! Hello!
```

Another interesting example is Daytime. Show RFC867 at UDP datagram. You can run the following:

```
$ nc -u 127.0.0.1 13 type whatever Sat Dec 16 16:15:40 2023
```

Also show the DNS example. Just run it with:

```
$ python src/simple-dns/main.py
```

4.2.2 TCP

The Transmission Control Protocol (TCP) is a *connection-oriented* protocol that provides a reliable packet transport to the application layer:

- It is based on a combination of *Go Back N* and _Selective-Repeat_algorithms. Lost and corrupted packets are retransmitted.
- It uses mandatory *checksums* for error control.
- Use of control messages like ACKs for ensuring packets are received.

TCP is a complex protocol where client *establishes a connection* before exchanging data with the server. At the end of the communication, the connection is closed.

TCP can be described as a *Finate State Machine* (FSM). The machine is really a program receiving/sending data and jumping between states. The next state will depend on:

- The information received
- The specific state where the machine is in.

Note: Show the state diagram. The machine starts in `LISTEN` state. In that state, if a `SYN` message is sent, the machine changes to `SYN_SENT` state.

TCP provides a *stream delivery service*. This means that the information is transmitted as a stream of bytes from the client to the server. From the client and server points of view, there is an imaginary pipe *connected* to both ends where bytes arrive in order to the other side.

TCP packets are called *segments* and the segments of a specific connection are related each other. This means that the fields in the headers are logically connected to previous and following segments so they can be reassembled when received.

Note: We will see this more in depth in the following sections but it is worth mention the *circular buffers*. This technique is used by TCP for providing reliability and flow control:

- On the sender side, they keep two types of segments:
  1. The ones ready to be sent: these segments are built straighaway from the data coming from the application layer.
  2. The ones already sent: when segments are already sent, the buffer can rotate and move on.
- On the receirver side, they keep two types of segments:
  1. The ones just received but not read: those are waiting to be read by the applicatio.
  2. The ones read by the application: when segments are read, the buffer can rotate and move on.

One of the important header fields are the *sequence number* and the *acknowledgement number*. They both define the *numbering system* of TCP that allows:

- Reliability: as they provide confirmation of the reception.
- Flow control: they define the amount of data the receiver is able to digest.

TCP assigns numbers to the data. This *byte number* is:

- Different for each direction of the communication.
- Initially generated randomly. It does not need to correlate to the data that is meant to be sent.

Note: For example: we want to sent 6000 bytes of data. If the first byte is numbered as 5432 (generated randomly) the last byte will be $5432 + 6000 - 1 = 11431$

TCP also assigns numbers to each segment. This is known as *sequence number* and it is generated as follows:

- The *initial sequence number* (ISN) for the first segment is generated randomly. It will be typically the first byte number (also generated randomly).
- The following sequence number will be the previous segment's sequence number plus the number of bytes carried by this previous segment.

Note: The example of the slide is the transmission of a file of 5000 bytes in 5 segments. If the first byte was labeled as 10001, the:

- The first sequence segment will be 10001 and it will contain 1000 bytes. The range of the byte numbers will be from 10001 to $10001 + 1000 - 1 = 11000$

TCP segments can carry data and/or control information. Sequence numbers are *only* use when data is carried. For those segments that only carries control information, the meaning of their sequence number is different: they are not related with the data transmission. As we will see later, for the future sequence numbers, these segments just consume 1 sequence number (as the carried 1 byte).

On top of the sequence number, TCP header can also include an *acknowledgement number*. The value of this number is the next byte number of the last one received. In other words, *it is the number of the next byte expected*. For example: if `ack = 5643` it means that all bytes until `5642` has been recieved and we are waiting for the next one. Of course, it does *not* mean the total bytes received are 5642 because the initial byte number is random and it is likely it was not 0.

The TCP segment

A segment has two parts:

- A header that can be between 20-60 bytes.
- The payload of the TCP segment.

The TCP header has the following structure:

- *Source port address*
- *Destination port address*
- *Sequence number*: if needed, it is the value of the first byte number carried by the segment. Initially it takes a random ISN as value.
- *Acknowledgement number*: if needed, it is the value of the byte number expected. The previous byte number has been received properly.
- *Header length*: measured in 4-byte words. This field is 4-bit long and since the header can be from 20 to 60 bytes, the values of this field might be from 5 to 15.
- *Control bits*: a 6-bit field where each bit categorise the segment:
  - `URG`: the segment carries urgent data so the urgent pointer field is valid.
  - `ACK`: the segment is an acknowledgement.
  - `PSH`: the segment data should be pushed to the reciever application without waiting for buffers or windows.
  - `RST`: the segment requires the connection to be reset.

  - SYN: the segment requests the synchronisation of sequence numbers.
  - FIN: the segment requests the termination of the connection.
- *Window size*: gives the information to the sender of the the maximum amount of data that the receiver wants to receive. Thus, the sender will have to respect this value.
- *Checksum*: it is mandatory in TCP and it is calculated very similarly to UDP's one (it adds the pseudo-header on top of the TCP segment).
- *Urgent pointer*: if URG was set, this field points to the last byte of the urgent data within the segment. The urgent data is placed at the beginning and this pointer helps to know when it ends.
- *Options and padding*: this optional field might incorporate extra data up to 40 bytes.

The TCP connection establishment

TCP is a connection-oriented protocol where a *bidirectional connection* is established between sender and reciever before any data is exchanged. Once this connection is established, all segments will be logically related to this connection.

The TCP connection establishment is known as the *three-way handshaking* and it works like this (values come from the example of the slides):

1. The server starts to listen for incoming connections. We say the server has a *passive* role during connection openings because it waits for the client to start them.
2. The client starts creation of a connection to the server. We say the client has an *active* role during connection openings because it initiates them.

   The first one is SYN segment which requires to the server to synchronise the sequence numbers. The *ISN* in the example is 8000.
3. The server receives the SYN, so it sends a segment of 2 types:
   - SYN: requesting the client to synchronise the sequence number. The server's ISN is 15000.
   - ACK: acknowledging the previous client's SYN. So the ACK number is 8001.
4. The server recieves the SYN-ACK segment and responses with an ACK segment. This segment acknowledges the SYN request in the previous message so:
   - The sequence number is 8001 as it is the following one to 8000.
   - The ACK number is 15001 since we have recieved 15000 before.

   Tip: The client could also send data along this ACK segment. That way it can take advantage of this ACK segment and start sending data instead of doing it in a different segment. This is known as *piggy-backing*. If the client does *not* piggyback data within this segment, the sequence number will *not* be increased later.
5. The connection is now established.

This connection establishment procedure has an intrinsic problem: the server has to acknowledge the initial SYN sent by the client. A malicious actor can send lots of SYN segments with *spoofed* source IP. This will make the server to send lots of SYN-ACK segments to clients that do not exist. Since they do not exist, the server will never get a response so it will wait for the ACK untile the timeout expires. If there are lots of SYN segments it is possible that the server waits for so many clients that it reaches its limit. Thus, any *new and legit* SYN request will not be replied. This is a classic *Denial of Service* (DoS) attack.

The TCP data transfer

Once the connection is established, then the bidirectional data exchange can happen:

1. The client sends 1000 bytes:
    - The `SEQ` number is again 8001 because the previous `ACK` (the last segment of the connection establishment) did not contain data.
    - The `ACK` number is again 15001 because it is the expected byte number from the server.
    - The `PSH` is on.
    - Note the byte numbers go from 8001 to 9000
2. The client sends another 1000 bytes:
    - The `SEQ` number is now 9001 (8001 + 1000).
    - The `ACK` is still 15001
    - The `PSH` is on.
    - The byte numbers go from 9001 to 10000
3. The server has recieved both segments and it replies with an `ACK` segment piggybacked with data:
    - The `SEQ` number is 15001. It is still the same sequence number as the server has not sent data until now.
    - The `ACK` number is 10001, meaning that both chunks of 1000 bytes have been received.
    - The `rwnd` is set to 3000, telling the client not to send more than 3000 bytes in a segment.
    - 2000 bytes of data, so the byte numbers will go from 15001 to 17000.
4. The client recieves the segment and creates another `ACK` segment, this time with no data:
    - The `SEQ` number is 10001, as it comes after 1000 bytes previously sent.
    - The `ACK` is 17001, telling the server that it has recieved the last 2000 bytes.
    - The `rwnd` is set to 10000, telling the server not to send more than 10000 bytes in a segment.

The TCP connection termination

After the data is exchanged, the connection can be terminated at any time. The closure can be done in 2 ways:

- *Three-way handshaking*: this will fully terminate the connection.
- *Half-close*: this will allow the server to send data to the client before closing the connection.

The three-way handshaking connection termination is as follows:

1. The client sends a `FIN` segment. The `SEQ` and `ACK` number values would depend on the previous conversation, so let's suppose `x` and `y`.

    Tip: This `FIN` segment may also piggyback data, in which case it will affect the following sequence numbers.

2. The server sends a `FIN-ACK` segment, acknowledging the previous `FIN` request.
3. Finally, the client sends an `ACK` segment previous server's `FIN` segment.

The half-close connection termination is as follows:

1. The client sends the `FIN` segment.

2. The server sends an `ACK` segment this time, meaning that the `FIN` has been recieved.. However, the server still can send data to the client as a `FIN` was not sent. All the recieved data in this time period will be acknowledged by the client.
3. When the server finishes the sending data, its `SEQ` number would be `z`, so a `FIN` is sent to the client with this sequence number.
4. Finally, the client sends an `ACK` segment for `z + 1`.

4.2.3 Segment loss

In a normal operation, `ACK` numbers are always recieved in order.

Note: In the example, a client and a server exchange data:

- 200 bytes to the server
- 1000 bytes to the client
- Client acknowledges.
- Server sends 2 segments of 1000 bytes each.
- Client acknowledges both.

However, some segments might get lost. This is detected because client and server keep a set timers and, if the expire, segments are retransmitted.

Note: In the example, a client and a server exchange data:

- The client sends 2 segments of 100 bytes each.
- The server acknowledges both.
- The client sends another 2 segments of 100 bytes each but the first one is lost.
- The server sends `701` as `ACK` number because, even though it has recieved the second chunk, the first one was not.
- The client's timer expired for the chunk 701, so the segment is re-sent.
- The server now recieves it so it can now acknowledge both (701 and 901).

4.3 Reliable protocols

As we have mentioned, TCP is a complex protocol that uses different techniques for achieving flow, error and congestion control. In this section we are going to explore some of the protocols that provide reliability.

4.3.1 Simple protocol

The simplest protocol that provides reliability is the one that works on ideal conditions:

- No packet is lost.
- There is no corrupted packets along the way.
- All packets arrive in order.

Under this cirscunstancies, the protocol can be connectionless and there is no need of flow/error/congestion control. However these are ideal conditions that almost never happen in reality.

4.3.2 Stop and wait

This protocol is the simplest way to provide flow and error control to an unreliable network. It uses a *sliding window* of size 1, which means that the sender has to wait for the ACK of a packet before sending a new one. With this simple approach, the flow control is provided (the reciever only sends the ACK when it is ready to receive more).

In this protocol, the following mechanisms are used:

- *ACK for each packet*: which means a sliding window of size 1.
- *Sequence numbers*: the amount of sequence numbers depends on the size of the sliding window. It uses $mod 2^m$. This is the same as saying that it goes from 0 to $2^m - 1$. In this case, the sequence numbers will be 0 and 1.
- *Checksum*.
- *Timers*: for each packet sent. If no ACK is recieved and timeout expires the packet will be retransmitted.

In general, the sliding window size N depends of a variable m: $N = 2^m - 1$. The top and wait protocol uses $m = 1$, so the window size is $2^1 - 1 = 1$. The window size can be seen as the buffer that controls what has been sent or recieved.

Note: In the diagram, you should show that window size is the sum of:

- Packets already sent but not ACK'd
- Packets ready to be sent.

When an `ACK` is recieved, the window slides the amount of positions that the `ACK` includes.

Note: The example flow diagram works like this.:

1. Since we are in stop and wait protocol, both sender and receiver use $m = 1$ so both have a window size of length 1. The sequence numbers are 0 or 1.
2. The sender starts sending the packet 0. It starts a timer associated to this packet.
3. The receiver moves its window one position and sends an `ACK`. Note that `ACK 1` confirms the reception of packet 0.
4. The sender gets the ACK and moves its window one position further. The timer is stopped.
5. The sender sends the packet 1. It starts a timer again.
6. The packet 1 gets lost.
7. Since nothing happens, the timer expires.
8. The sender resend the packet 1.
9. The receiver gets packet 1, move its window one position forward and sends the ACK.

10. The sender receives the ACK and moves its window one position ahead. The time is stopped now.
11. The sender tries to send packet 0 now.
12. The receiver gets it and moves its window one position more. Note that the receiver now waits for packet 1, so it sends `ACK 1`.
13. The ACK 1 gets lost.
14. Since the sender did not get any ACK, it resends packet 0.
15. The receiver gets packet 0 but it is wainting for packet 1, so it discards it considering it as duplicated.

4.3.3 Go back N

The stop and wait protocol is very inefficient. The communication channel might be free to be used but if the sender is waiting for an ACK then it will not be used.

Go back N is a generalisation and improvement of stop and wait:

- More than just one packet might be sent.
- Multiple packets might be pending to be ACK'd at a time.
- If errors are detected it will be required to go back N packets and retransmit them.

Note: The example shows the reason why the window size should be always lower then 2^m. It is because the sequence numbers never get confused by the receiver even if all ACKs are lost.

The first case (the right one) the window size is 3 which implies $m = 2$ and 0 to 3 sequence numbers. All packets are received but all ACKs are lost. This means that the sender has to go back to the first packet 0 and resend it. The receiver is waiting for packet 3 though, so it will discard packet 0 correctly.

The second case (the wrong one) $m = 2$ but the window size is 4 (instead of being lower than 4). This means that the last request is packet 3. Again, all ACKs are lost so we have to retransmit from packet 0. In this case, the receiver had received packet 3 so it is waiting packet 0 (as the top sequence number is 3). The receiver can not tell if the packet 0 it has just received is a re-transmission or a new packet 0.

Note: The next example is $m = 3$ so the window size in the sender is 7, and the sequence numbers will go from 0 to 7:

1. The sender sends packet 0.
2. The receicer acks it.
3. The sender gets the `ACK` and moves the window one position.
4. The sender sends 3 packets: 1, 2, 3.
5. The receiver gets all of them but the ACK for the first one is lost.
6. The sender gets the ACK for packet 2. The window slides 2 position (packet 1 is assumed). The ACK for packet 3 is yet to arrive, so it restarts the timer.
7. The sender gets ACK for packet 3, so the window slides 1 position and timer is stopped.

Note: The last example is similar to the previous one. Same initial conditions:

1. The sender sends packet 0.

2. The receicer acks it.
3. The sender gets the `ACK` and moves the window one position.
4. The sender sends 3 packets: 1, 2, 3. This time packet 1 is lost.
5. Packet 2 arrives to reciever but it is expecting 1 so it is discarded. The receiver sends an ACK waiting for 1.
6. Packet 3 arrives to reciever but it is expecting 1 so it is discarded. The receiver sends an ACK waiting for 1.
7. The ACKs arrive to the sender. They tell that packet 0 has been received but packet 0 is no longer in the sender's sliding window. This ACK does not make sense and it is discarded.
8. After a while, the time expires and the sender goes back to the first sent-but-not-ack'd position in the window and starts to resend.
9. Now all packets arrive and their associated ACKs.

4.4 Exercises

TBD

CHAPTER

FIVE

NETWORK LAYER

This lesson describes in depth the network layer. It descripts the IP addressing and routing mechanisms too. You can find the slides here.

Under the transport layer, the network layer is used mainly for delivering data between hosts as long as the hosts are connected through a set of inter-networks. The following are the main features of the network layer:

- *Payload encapsulation*: data from the transport layer is encapsulated and delivered to any point of the internet.
- *Routing*: the data packets are sent *host-to-host* although they can travel through other hosts or networks. What route a packet takes is a decision made with the help of *routing protocols* and it might the case that packets of the same transport flow go through different routes (*connectionless*).
- *Forwarding*: the *routing nodes* of the network layer are known *routers*. They perform packet forwarding: decide what is the next hop based on the *routing table*.
- *Best-effort delivery*: the network layer is *unreliable*. There is no guarantee that a packet has arrived to destination nor it arrived in the right order.

Note: In the network example:

- The `Link` nodes are devices that only operates at data-link layer. They just connect nodes directly.
- The `Intermediate system` are nodes that can connect networks.
- A datagram going from the node `A` to any of the `End system` follows the encapsulation/decapsulation trace at the bottom of the image. Each node of the network uses up-to network layer.

Note: In this example, the *connectionless* concepto is shown clearly. Packets 1, 2, 3 and 4 take different routes and arrive out of order. Note that this is a packet-switched network where information is transmitted as individual datagrams.

5.1 IP protocol

The Inter-network Protocol (IP) is the standard network protocol of the Internet. It can be used to interconnect different types of networks (made out of different technologies) as long as the nodes understand IP messages.

In this section we are going the dive into the format of the IP datagrams, the fragmentation mechinism, how IP addresses work and how they can be strucuture using subnetting.

5.1.1 Format

The IP datagram has a maximum length of $2^1 6$ bytes:

- The minimum header length is 20 bytes. The header can be extended 40 bytes more with options.
- The rest is reserved to the payload.

These are the fields of the IP header:

- *Version*: 4 bits for specifying the IP version. `4` for IPv4 and `6` for IPv6.
- *Internet Header Length (IHL)*: 4 bits for specifying the length of the IP header in 4-byte words.
- *Type of Service*: 8 bits for helping to service differentiation (also known as *quality of service*)
- *Total Length*: 16 bits for specifying the total length of the IP datagram, including the header.
- *Identification*: 16 bits for identifying the datagram if it belongs to a fragmented one. If a datagram is fragmented, this field is the same for all the fragments.
- *Don't Fragment (DF)*: 1 bit to tell routers the datagram should not be fragmented.
- *More Fragments (MF)*: 1 bit to tell routers this datagram is not the last one of a serie of fragmented datagrams.
- *Offset*: 13 bits to tell the position of this datagram in the series of fragmented datagrams. It is expressed as a value of 8-byte words, so:
  - There can only be 2^8 fragments.
  - The size of a fragment must be mutiple of 8, except for the last one. The first fragment has this value always set to 0.
- *Time-to-live (TTL)*: 8 bits to tell the router if a datagram needs to be discarded. Everytime a router forwards a datagram, it decreases the TTL value. If the value is set to 0, the packet is discarded by the router.
- *Protocol*: 8 bits to tell what protocol is encapsulated:

Value	Procotol
1	ICMP
2	IGMP
4	IPv4
6	TCP
17	UDP
89	OSPF

- *Checksum*: 16 bits for detecting errors of the header. It needs to be checked on every hop of the route. The payload is not included and it is left to the upper layers to perform their own check.

  The field is computed by dividing the hearder in 2-byte fields, sum them all and complement the sum. If the final sum overflows, it is wrapped.
- *Source IP*: 32 bits for the source IP address.
- *Destination IP*: 32 bits for the destination IP address.
- *Options*: up-to 40 bytes is reserved for options that are used in specific scenarios and debugging. Although it is not mandatory to use options in the IP protocol, all IP devices should handle them if they are present. An option has the following structure:
  - *Code*: 8 bits for indicating the option
    - *Copy*: 1 bit for indicating whether the option needs to be copy or not to all the fragments.

    - ∗ *Class*: 2 bits for indicating the general category of the option.
    - ∗ *Number*: 5 bits for identifying the option. There are 2^5 options avaiable but only a few of them are frequently used.
  - *Length*: 8 bits for specifiying the length of the data of this option.
  - *Data*: variable-length field containing the data of the option.

5.1.2 Fragmentation

All networks are not the same. Each network may have its own features and capabilities and an important one is the *Maximum Transfer Unit* (MTU). This is the maximum size of a frame that can be transmitted through a link. This value might be different from one network to another, so routers will need to adapt the IP datagram size to it.

This process of routers slicing IP datagrams is known as *fragmentation* and the IP protocol provides support to handle the problems associated with it. At destination, the fragments are reassembled so the original IP datagram is built.

Note: The fragmentation can only happen at destination because intermediate routers might not see all the fragments passing through them. Since IP datagrams might take different routes, because they are independent datagrams, only the destination is capable of reassembling them.

Note: The fragmentation example should be explained as follows:

- A datagram is going to be sent from `source` to `destination`.
- The router `R1` will have to fragment it in 3 fragments ($4020/1420 = 2.5$ so we need 3 fragments).
- `f1` and `f3` will no need more fragmentation because the MTU will be enough for the rest of the hops.
- `f2` requires one more fragmentation step in `R4` as it goes to a smaller MTU.
- In total, 5 fragments will arrive.

In the following slide, you should explain how the data is distributed across the different fragments:

- An important aspect to take into account is that if the IP header is 20 bytes, these 20 bytes must be *always* included. So `R1` will only be able to send 1400 bytes (plus 20 bytes of the IP header) through the bottom network. This is why the `f1` contains 1400 bytes (from 0 to 1399), `f2` contains 1400 bytes (from 1400 to 2799) and `f3` contains 1200 bytes (from 2800 to 3999).

  The identification field for all these fragments are the same as the original datagram (14567).

  The total length of each fragment changes to reflect the new size of the datagrams: 1420 (1400 + 20 for the header) for `f1` and `f2`, and 1220 for `f3`.

  The `M` bit is set to 1 for `f1` and `f2` but is set to 0 for `f3`. This bit indicates if there is more fragments to come. `f1` and `f2` has more fragments to come but `f3` is the last one.

  The last field that needs to be kept into account is the *fragmentation offset*. This field is calculated based on the first byte number of data of each fragment divided by 8:

  - `f1`: $0/8 = 0$
  - `f2`: $1400/8 = 175$
  - `f3`: $2800/8 = 350$

- `f2` is split in 2 more fragments: `f2.1` and `f2.2`. This would be a good exercise to calculate the different fields with the students.

  The original size of the datagram is 1400 bytes (plus 20 bytes for the header). The MTU is 820, so we can split this datagram is 2: `f2.1` of size 800 bytes (from 1400 to 2199) and `f2.2` of 600 bytes (from 2200 to 2799).

  The idenfication field remains the same.

  The total length now is set to 820 and 620 respectively.

  The fragment offset remains the same for `f2.1` (1400/8 was already calculated) but it is different for `f2.2` (2200/8 = 275).

  An important detail is that the `M` bit is 1 too all of them. `f2.2` is *not* the last fragment of the original IP datagram, it is just another fragment within

Note: The DIY examples:

- We use `ping` for sending 1 ICMP packet with 4000 bytes of data. Note that the MTU is 1500 bytes. `tshark` is used for capturing traffic. It shows that the datagram is split in 3 fragments.

  `ping` is actually sending 4028 bytes: 4000 bytes of data, 20 bytes for the IP header and 8 bytes corresponding to the ICMP header.

- In this case, the ICMP packet contains 3992 bytes of data, thus it is sent 4020 bytes (including the IP and ICMP headers).
- Here we use a Python library called scapy for generating an IP datagram. We add a payload of 7000 bytes and this generates 5 fragments. You can find the code here.

5.1.3 Addressing

The IP protocol addresses are 32-bit integers. Each node of an IP network will have, at least, one IP address that will identify it *uniquely* within the network. Note that a node can belong to multiple networks (like a router) so it could have multiple IP addresses associated to it.

The addresses are hierarchical, similar to a postal address (e.g. country/city/street). The 32 bits that make the IP address are divided in 2 groups:

- *Prefix*: that identifies the *network*.
- *Suffix*: that identifies the *host* within that network.

Note: For example, the address `192.168.1.128` with a prefix of n=24, the network is `192.168.1` and the host is `128` within that network. However, if n=8 the network is `192` and the host would be `168.1.128`.

There are a few special addresses:

- `0.0.0.0`: indicates *this host*. Its meaning might vary depending the context it is used. For example, it might indicate that a service can listen on *any* interface of the host. In DHCP, as we will see, refers to the host that is requesting an IP address.
- `255.255.255.255`: is the broadcast address. Any message send to this direction will be received by all hosts.
- `X.X...0000`: is the network address. It is the first address of the range defined by the prefix.
- `X.X...1111`: is the broadcast address within the network. It is the last address of the range defined by the prefix.

- `0.0...X`: identifies a host within the network.
- `127.0.0.0`: the loopback network used for testing network applications without the need of using real Internet connection.

Originally, the Internet was divided in classes:

- *Class A*: start with `0` and uses 8 bits for the prefix.
- *Class B*: start with `10` and uses 16 bits for the prefix.
- *Class C*: start with `110` and uses 24 bits for the prefix.
- *Class D*: start with `1110` and use for multicast addresses.
- *Class E*: start with `1111` and reserved for future use.

Note: Classful addressing is now *obsolete* because it couldn't fit well to address demand. The class A addresses provide a ver large range of hosts that can be in the network ($2^2$4) but only 128 networks can be defined. The class C only provides 256 host addresses but the next step (the class B) provides $2^1$6. Organisations and companies that need to go from C to D might not make a good use of all that range.

In general terms, classful addressing makes the address ranges to be not well used and do not adapt to the real demand.

Currently, the Internet uses classless addressing, where we just need to fix the prefix lentgh *n* to fully define a network address. We can define this *n* as we wish, based on our needs, so very little amount of addresses will be wasted. The notation of an address will be: `XXX.XXX.XXX.XXX/n`.

Note: In the example of `167.199.170.82/27`:

- There allows a block of 2^5 addresses.
- The first one is the network address: `167.199.170.64`
- The last one is the network broadcast address: `167.199.170.95`
- Note that `167.199.170.64` and `167.199.170.95` are *not* used for hosts. So for addressing hosts in that range we would have $2^5 - 2$ addresses.
- `167.199.170.82` is the address of the host and, for example, `167.199.170.195` do not belong to this network.

Routers forward datagrams from one network to another, depending on the destination IP. This is an operation that must be quick because they will be doing *a lot*. In order to know what is the network address of a given IP can use the concept of *network mask*. The mask is a 32-bit number where the first *n* bits (the length of the prefix) are set to 1 and the rest to 0. For example: `167.199.170.82/27` has a mask of `11111111.11111111.11111111.11100000`. By using this mask, it is only required an `AND` operation to get the network address:

```
167.199.170.82   -  10100111.11000111.10101010.01010010
255.255.255.224  -  11111111.11111111.11111111.11100000
-------------------------------------------------------
                    10100111.11000111.10101010.01000000  - 167.199.170.64
```

In classless addressing, the IP must be provided along its mask. That way, the host is fully defined in the Internet. `IP/mask` format is known as *CIDR notation*. Classless Inter-Domain Routing (CIDR) is a method of allocating IP addresses which was created for replacing the classful approach of the Internet. IANA delegates the assignation of the IP addresses world-wide by providing different ranges to local entities like ISPs or similar companies. As we have seen, CIDR is based on *VLSM* where the network mask can be of different length to provide accurate ranges for different needs without wasting too many IPs.

There has been defined a few IP addresses for private use. These means that these IP ranges will not go through the public Internet and can be used within an organisation:

- `10/8`
- `172.16/12`
- `192.168/16`

There are also the *link-local addresses* that are used for automatic configuration of the hosts in a network: hosts without an IP address select an IP address from the range `169.254/16`. Although these addresses will never go to the Internet, they can be useful for contacting neighbours though.

5.1.4 Subnetting

Subnetting is the process of subdividing a network into smaller networks so the IPs provided by the original range can be better used. If you ever get assigned to a class A or B, it is clear that the amount of hosts available in that range could be excessive for your needs. Using subnetting you can split the available space into sub-networks. Each sub-network could also be split in other sub-sub-networks and so on. This way, it is possible to organise a big range of IPs and make better use of the available addresses for hosts on each network.

Note: In the example:

- The orginal n is 16 (a class B network).
- We use 4 bits more for creating subnets. This means we can have up-to 16 subnets (2^4).
- Each sub-net will be able to address $2^{32-20} - 2$ hosts. Note we substract 2 because we need network and broadcast addresses.

Because we *extend* the prefix length to create subnets, the number of possible subnets will be always power of 2. I can be applied to any block as long as it is not used already.

Note: In the example we need to build 4 subnetworks for the network 141.14.0.0/24. This means that we will need to use 2 bits more, so the resulting subnets will be `/26` but with different values of course.

- 141.14.0.`00 000000`/26
  - 141.14.0.`00 000000` -> 141.14.0.0/26 net
  - 141.14.0.`00 111111` -> 141.14.0.63/26 brd
- 141.14.0.`01 000000`/26
  - 141.14.0.`01 000000` -> 141.14.0.64/26 net
  - 141.14.0.`01 111111` -> 141.14.0.127/26 brd
- 141.14.0.`10 000000`/26
  - 141.14.0.`10 000000` -> 141.14.0.128/26 net
  - 141.14.0.`10 111111` -> 141.14.0.191/26 brd
- 141.14.0.`11 000000`/26
  - 141.14.0.`11 000000` -> 141.14.0.192/26 net
  - 141.14.0.`11 111111` -> 141.14.0.255/26 brd

5.2 Configuration and management protocols

In this section we are going to explain two important protocols that help the network layer in some sense.

5.2.1 ICMP

IP is a network protocol that delivers datagrams. This delivery, even it is not reliable, requires some basic mechanisms to detect errors, network congestions, etc. The Internet Control Message Protocol (ICMP) works on top of IP to provide:

- *Error detection* at multiple levels.
- *Information* about the network status (hosts, routers, etc).

Each ICMP message is encapsulated in an IP datagram and the message format is as follows:

- *ICMP Header*: the header is 8-byte long.
  - *Type*: 1 byte for the type of the ICMP message. An ICMP message can contain different information based on the type, a *request*, an *error*, a *reply*, etc. Depending of the value of this field it would be one or the other.

Type	Description
0	Echo Reply
8	Echo Request
3	Destination Unreachable

  - *Code*: 1 byte for the subtype of the ICMP message. For example code `1` of type `3` means the destination *port* was unreachable, whereas code `1` of the same type means destination *host* was unreachable.
  - *Checksum*: 2 bytes for the checksum that is calculated like the IP's checksum field.
  - *Rest of header*: 4 bytes more whose value will depend on the specific ICMP message.
- *ICMP Data*: the content of the message.

Errors

The ICMP errors are used for signalling problems. They are always reported from the place they are generated *back to the source host*. There are some special cases in which errors are not reported:

- An error of an ICMP message.
- IP datagrams failures with special addresses like 127.0.0.1 or multicast.
- A failure of an IP fragment that is not the first one.

Note: In all these cases, it is possible to generate lots of error messages.

For reference, the ICMP error messages have the following values in the ICMP Data field:

1. The IP header of the affected datagram, and
2. the first 8 bytes of the affected datagram's payload.

Note: In the RFC 792 we can read:

Internet Header + 64 bits of Data Datagram

```
The internet header plus the first 64 bits of the original
datagram's data.  This data is used by the host to match the
message to the appropriate process.  If a higher level protocol
uses port numbers, they are assumed to be in the first 64 data
bits of the original datagram's data.
```

Some examples of ICMP errors:

- *Destination unreachable*: sent by the router or host that detects the datagram cannot be delivered to its destination. The code provides more specific reasons.
- *Source quench*: a very basic congestion/flow control mechanism. Requests to the source to reduce the data rate.
- *Time exceeded*: sent by the router to indicate that found a TTL set to 0 or by a destination host where a fragmentation timeout expired (e.g. some piece is missing).
- *Parameter problem*: sent by the router or host indicating that a datagram was malformed or has missing fields.
- *Redirect*: typically sent by routers to hosts to inform them that there are better routes for an IP datagram. The host could update its routing table after receiving that information. For example, for a specific datagram, there might be a better route than the one used by a host.

Requests

Some examples of ICMP requests and replies:

- *Echo*: used to verify connectivity between 2 nodes. An Echo Request is replied by an Echo Reply.
- *Timestamp*: used to measure latency between 2 nodes.
- *Address mask*: used by a host to query the network address mask to use to a router.
- *Router Solicitation*: used by a host to identify what local routers are available around it.

Note: The example of `ping google.com` shows how to measure the round trip time (RTT). This is the time between the Echo Request is sent and the Echo Reply is received.

Note: See the code of ping in Scapy here.

5.2.2 DHCP

Dynamic Host Configuration Protocol (DHCP) is an application-layer protocol for assigning IP addresses to hosts automatically. It follows the *client-server* paradigm where the hosts (clients) request to the DHCP servers to be assigned with a configuration. This configuration usually includes:

- An IP address.
- A network mask.
- A default gateway.
- A set of DNS name servers.

Less fequently the configuration can also include:

- A hostname.
- Date and time.

The format of a DHCP message is based on BOOTP as it was already widely used at the time DHCP was desgined:

- *Operation code*: 1 byte for telling if the message is a request or a reply.
- *HW type*: 1 byte to specify the hardware transmission medium.
- *HW address length*: 1 byte to specify how many bytes the hardware address is long. For Ethernet, this values is 6.
- *Hop count*: 1 byte to control the number of forwardings done by DHCP relays. Very similat to IP's TTL.
- *Transaction ID*: 4 bytes for an ID of the transaction. Randomly initialised by the client and repeated by the server.
- *Time elapsed*: 2 bytes to inform the number seconds since the client started.
- *Flags*: 2 bytes for flags. The first bit indicates if the message is unicast or multicast.
- *Client IP address*: the client IP. If the client does not know it, then `0.0.0.0` is used.
- *Your IP address*: the assigned client IP. This is set by the server.
- *Server IP address*: the server IP. If the client does not know it, then `255.255.255.255` is used.
- *Gateway IP address*: the default gateway associated to the network the client should use.

Note: The example of the DHCP operations can be explained as follows:

1. The client creates a `DHCPDISCOVER`. It does not know what DHCP servers are around. The message is sent to the broadcast address. Note that the source port and the destination port are fixec (68 and 67). We will see later why.
2. Every server that receieved the `DHCPDISCOVER` might reply with a `DHCPOFFER`. This message gives the client a possible configuration to use. Since the client does not have an IP yet, this message is sent to the broadcast address too.
3. The client might receive many `DHCPOFFER`. After that, it would select one of them and create a `DHCPREQUEST`. This request goes straight to the server that gave the offer to make sure that the configuration provided can be used for up-to `3600` seconds.
4. The server replies with a `DHCPACK` meaning that agrees with client's configuration request.
5. When the `DHCPACK` is received by the client, it can assume that it will be *bound* for that IP address for the leased time period.

DHCP uses the fixed source port 68 in order to prevent other UDP applications to receive DHCP messages from servers. Suppose an ephemeral port 12345 is used instead of 68. If there is an UDP application running somewhere in the network at the same port, the `DHCPOFFER` messages will arrive to this application too because they are all sent to the broadcast address.

Note: The state diagram shows how DHCP works. It is worth noting that the IP addresses are leased for a certain period of time. It's client's reponsibility to renew these leases on time.

Note: You can run `sudo dhclient -v` to see how the DHCP client starts the renewal of the configuration. Note that it will go to all interfaces of the computer.

5.3 IP routing

In this section we are going to focus on how the hosts and routers manage the IP datagrams so they can reach their destination.

5.3.1 Delivery methods

Every host of an IP network keeps a *routing table*. This table contains the required information for sending IP datagrams based on their destination address. The format of this table is as follows:

Net Address	Mask	Delivery Method
11.11.11.0	24	10.0.0.5
10.0.0.0	8	direct

In this example, there are 2 entries that can be read like this:

1. Any datagram sent to the 11.11.11.0/24 network (for example, to 11.11.11.45) should be sent to 10.0.0.5. This is likely to be a router that knows how to reach the destination host.
2. Any datagram sent to the 10.0.0.0/8 network (for example, to 10.10.0.1), it should be delivered directly. This means that the destination host is in the same network, so no need to send it to a router.

More formally, there are 2 delivery methods:

- *Direct*: if a datagram's destination is in the same network of the processing node then it is direct delivery.
- *Indirect*: the rest. This uses a *next-hop* address where datagrams will be sent.

5.3.2 Forwarding

Forwarding is the mechanismin where an IP datagram *is moved from one path to another* during a route from its source to destation. This requires:

- A *router*: which will move the datagram from one network to another.
- A *routing table*: which contains the information about the possible routes. Routers will query the routing table to make a decision about a specific datagram based on its destination address.

Methods

There are multiple methods that can implement a forwarding mechanism. In this section we are going to compare some of them, listing the its pros and cons.

- *Route method* vs *Next-hop method*: for each destination we could store:
  - The entire route: meaning that `A` needs to know `R1`, `R2` and finally `B`.
  - Just the next hop: meaning that `A` only needs to know `R1`. Note that `R2`'s next hop for reaching `B` is null because it would be direct delivery.

  Obviously, the next-hop method simplifies the routing tables that routers need to keep, so it seems a much better option. However, there is something good of having the entire route in the routing table: the context that the router has is bigger and could make smarter decisions (like foreseeing problematic branches). In any case, route method requires so much space and complexity to keep the routes up-to-date that it is not worth to implement.

- *Host-specific* vs *Network-specific*: the destinations can be:
    - All the possible hosts: this means that `S` will have to store one row for `A`, `B`, `C`, and `D`.
    - Just the destination network: meaning that `S` will only need to store the destination network address to know how to each *any* node from that network.

  Network-specific approach is better because simplifies the routing table and the process of updating them.

- *Default route*: a node or a router can be connected to a network that provide access to multiple inter-networks (like the Internet). We cannot store all the possible destination networks in our table. For that reason, the *default* route can be placed at the end of the forwarding table. If the route for a given destination address is not found then the datagram will be forwarded to the next hop that pointed by the default route.

Note: We can define a forwarding mechanism in pseudo-code:

1. Get the destination address of the datagram.
2. Loop through each row of the routing table.
3. Perform an `AND` operation between the destination address and the network mask specified in the row.
4. If the result is equals to destination value of the row, then forward the datagram to the next hop.
5. If no row matched and there is a default route, then follow the default route.
6. If there is no a default route, then return an ICMP error stating that there is not a route for the given datagram.

5.3.3 Routing tables

Given the previous procedure, each row in the routing table should contain the following fields:

- *Destination network address*: required to know what network address the route goes to.
- *Destination network mask*: required so the `AND` operation can be performed.
- *Next-hop address*: required to know what node will handle a matched datagram for this specific route.
- *Interface*: required to know through what interface the forwarded datagram should be sent.

The next-hop address and the interface is used downstream by ARP, a link-layer protocol that translates an IP address to a link-layer address. The link-layer address can be used by the link-layer to send frames containing the IP datagram. We will study ARP in more detail in the following chapter.

Note: The example shows the routing table of the second router that has 2 interfaces: `eth0` and `eth1`. `eth0` is connected to 40.0.0.0/8 network and its assigned IP is 40.0.0.8. `eth1` is connected to 128.1.0.0/16 network and its assigned IP is 128.1.0.8.

Any datagram to any of the these networks will be considered for *direct delibery* because the router is connected to them. However datagrams to 30.0.0.0/8 network will need to be sent to 40.0.0.7 which has access to that network. Datagrams to 192.4.10.0/24 will need to be sent to 128.1.0.9 for the same reason.

It is important to note that any other destination address, like 8.8.8.8, will generate an ICMP error because there is no route for it.

Note: The next example shows the routing table of R1 which has 3 interfaces. In this case, a default route is configured.

Note: Based on the previous example, let's explain what it would happen if a datagram arrives to R1 with destionation address 180.70.65.140:

1. The first row's mask would be applied to the destination address:

   180.70.65.140 & 255.255.255.192 = 180.70.65.128

   It does not match the first row's destination field.

2. The second row's mask would be applied to the destination address:

   180.70.65.140 & 255.255.255.128 = 180.70.65.128

   In this case, it matches so the next-hop address and the interface is passed to ARP.

Note: Based on the previous example, let's explain what it would happen if a datagram arrives to R1 with destionation address 201.4.22.35:

1. The first row's mask would be applied to the destination address:

   201.4.22.35 & 255.255.255.192 = 201.4.22.0

   It does not match the first row's destination field.

2. The second row's mask would be applied to the destination address:

   201.4.22.35 & 255.255.255.128 = 201.4.22.0

   It does not match the first row's destination field.

3. The third row's mask would be applied to the destination address:

   201.4.22.35 & 255.255.255.0 = 201.4.22.0

   In this case, the result matches so the next-hop address and the interface is passed to ARP.

Note: Based on the previous example, let's explain what it would happen if a datagram arrives to R1 with destionation address 18.24.32.78:

All rows in the table will be queried and no one will provide a match except the latest one: the default route.

Now that each destination network address and mask need to be kept on each row, one can think that in a real environment with lots of networks, routers might need to keep lot of destination networks if neighbours are connected to multiple networks.

However, routers can use *address aggregation* to simplify their routing tables. Many network addresses can be collapsed into just one that includes many of them. That way, only one table row is required to reach multiple networks.

Note: In the example, R1 is connected to 4 networks, all of them /26. R2 does not need to know all those networks, it only needs one row for 140.24.7.0/24.

And it is /24 because R1 uses 2 bits for subnetting (/24 + 2 = /26).

However, address aggregation introduces a new problem. What if one of the 4 networks is removed from R1 (for example, 140.24.7.192/26) and now it is placed in a separated router. Now R2 seems like it will not be able to use 140.24.7.0/24 in just one row. It will need 2 rows: one for 140.24.7.0/24 and one for 140.24.7.192/26 because each of them are connected to different interfaces. But one includes the other so both will match.

In order to disambiguate what row to pick the *longest mask matching* criteria will be followed. This means that the row in the routing table whose mask is the longest will be selected in case there are others shorter. In our case, `/26` is longer than `/24` so both rows can coexist without any problem in R2.

Note: To see how this address aggregation provides a hierarchical routing, let's consider the following example:

- A regional ISP is assigned with 120.14.64.0/18. This means that it has 2^{16} addresses.
- This ISP divides this block in 4 to give each block to local 3 ISPs It needs 2 bits for subnetting. One block will not be used but the other 3 will be assigned to the local ISPs:
  1. 120.14.64.0/20: this local ISP subdivide this block in 8 subnetworks that will be assigned to small providers. That requires 3 bits so the result in /23 networks:
     - 120.14.64.0/23, to
     - 120.14.78.0/23

     And each of these small ISPs divide them into 128 blocks that will be assigned to households. That requires 7 bits, so the resulting networks will be /30:
     - 120.14.64.0/30 - …, to
     - 120.14.78.0/30 - …
  2. 120.14.80.0/20: not assigned and reserved for future use.
  3. 120.14.96.0/20: this local ISP breaks down the network in 4 subnetworks, resulting in /22.
  4. 120.14.112.0/20: this local ISP breaks down the network in 16 subnetworks, resulting in /24.

In GNU/Linux there are tools that can be used for querying the routing table and the host configuration.

- `ip route`: shows the routing table. This can be also done with `route` or `netstat -rn`.
- `ip addr show`: shows the interface configuration (its IP address, network mask and much more). This can be also done with `ifconfig`.

5.3.4 Dynamic routing

Each row of the routing tables represent a route that can be reachable from the host or router where the routing table lives. The routes can be added and removed manually from the routing table. However, that will not provide a good user experience as the Internet is constantly changes. There are nodes that are not available anymore and new routes that can be used very frequently.

In order to update the routes automatically, we will need a *routing protocol* between routing nodes so they can configure themselves. With this type of protocols, routing nodes can discover new routes from their neighbours, announce a faulty path, etc.

As mentioned, these routing protocols are used between routing nodes. Specifically, they are effective at the level of *autonomous systems* (AS). The Internet is built out of autonomous systems can be a single big network of a set of networks of networks. The main point of an AS is that it defines a single routing policy and owns a set of IPs (its IP address space). An *AS number* (ASN) is assigned to each AS that is connected to the Internet.

Note: In the slide there is 4 AS that are interconnected. Some routers within the AS can connect to external AS routers.

The routing protocols can be:

- *Intra-domain*: they are used within an AS (domain). The most common one are:

  - *RIP*: *Routing Information Protocol* is based on the distance vector algorithms that tries to get the best route to a destination based on the number hops and other metrics.x
  - *OSPF*: *Open Shortest Path First* builds a database of the link status with a *cost* metric.

- *Inter-domain*: they are used between AS (domains). Here it is used BGP (*Border Gateway Protocol*) where routes are announced and routers take that information for keeping their tables up-to-date. BGP allows having multiple route selection with different weights.

5.4 Exercises

TBD

CHAPTER

SIX

DATA-LINK LAYER

This lesson describes in depth the data-link layer. It descripts its functionality, the link-layer addressing and ARP. You can find the slides here.

The data-link layer (or just link layer) is in charge of the *node-to-node* communication. It is important to note that the *domain* of the link layer is limited to the nodes that are *directly* connected. This means there is *no mediators* in the communication provided by this layer.

It is located under the network layer, so it receives/sends *frames* of data that are going to be used by the network layer. It is also located on top of the physical layer so it does not deal with the transformation of the data into the physical signals that will be sent through the communication medium. However, it will have to deal with some problems that the physical medium generates like transmission errors, propagation delays, etc.

6.1 Functionality

The data-link layer is usually divided in 2 sublayers that perform operations at different abstraction layer:

1. The *data link control* layer (DLC): focused on error and flow control on both point-to-point and broadcast links.
2. The *media access control* layer (MAC): focused on the problems associated with sharing a medium like a broadcast links. This sublayer is not required in those situations where only point-to-point links are used.

6.1.1 Framing

The first fundamental service that provided is *framing*. The PDU is called *frame*. The network datagrams are encapsulated in frames that are going to be sent to the destination. In the destination, those frames are going to be decapsulated and passed the transported datagram to the network layer. The *intermediate nodes* perform these 2 operations all the time. This is required because:

- They connect two different physical technologies
- The addresses between link domains are different, so the frames need to be updated.

Each frame has the following parts (shown in the slide):

- *Flag*: used at the beginning and at the end of the frame to mark when it starts and when it ends. The byte `01111110` is used for this.
- *Header*: with the source and destination addresses, sizes, etc.
- *Payload*: the data coming from the network layer.
- *Trailer*: typically used for error checking.

Note: This framing process can be seen as our *postal* code. When we put a message in an envelope, we are delimiting the information to be transported in a *frame* (the envelope).

The frame size can be:

- *Fixed*: in which case there is no need of using any kind of delimiters as the fixed size itself can be used to distinguish different frames. ATM (mostly used for WAN connections) uses this type of fixed frames called *cells*.
- *Variable*: prevalent in LAN environments and need the use of delimiters. We have seen that we are going to use a known flag as a delimiter but: what would it happen if the flag is part of the network data? In order to solve this problem, an *escape* value is used:

  1. If FLAG is in the data, then add a special value ESC in front of it.
  2. If ESC is also in data, then add ESC again as a prefix to avoid it.

  Since this is done at byte level, this technique is known as *byte stuffing*. This process is performed for every single FLAG or ESC by found in the data.

  Note: In the figure, the data contains FLAG and ESC. The generated frame will have `ESC FLAG` and `ESC ESC`. When this frame is received, the receiver will just to remove the escaping characters to get the original data.

  Since process is done at byte level, it does not work well with today's data encoding like Unicode that uses 16 or 32 bits. For this reason, most of the protocols perform the this escaping technique at bit level. This is known as *bit stuffing*:

  Note: Since de flag is `01111110`, the algorithm for escaping a problematic sequence goes as this:

  – If `0 11111` is found in the data then a `0` is appended to avoid matching the FLAG. Then the next bit is added normally.

  – If the receiver detects `0 11111 0`, then it transforms it to `0 11111`.

  – The real flag is never stuffed so there is no possible conflict.

6.1.2 Flow control and congestion control

As we have seen in other layers, a frame producer can overwhelm a receiver if the number of frames pushed from the first one to the second one is too fast. The link layer provides flow control by:

- Using buffers on both sides so those frames that cannot be processed at a certain time can be done later.
- Use control frames for signalling the producer to reduce the transmission rate.

6.1.3 Error control

The physical layer is not reliable. Even technologies like fibre optic can produce transmission errors and corrupt data while sending bits through the transmission medium. In order to detect this kind of problems, a *Cyclic Redundancy Check* (CRC) is added to the frames by the sender and tested by the receiver. This is very similar to the checksum fields we have seen so far. If a corrupted frame is silently discarded, meaning that the network layer will not receive any of those. For those that are *not* corrupted it needs to:

1. Pass it to the network layer,
2. Optionally, signal the sender with an ACK frame that the sent frame was received correctly. This approach is also related with the flow control.

6.1.4 Connectionless and connection-oriented

Depending on the protocol used, the data transmission can happen:

- *Connectionless*: where each frame is independent from each other and it is independent from the others. Most wired LAN protocols use this.
- *Connection-oriented*: where the establishment of a connection is required prior the data transmission. Some WLAN protocols and point-to-point WAN protocols use this approach.

6.2 Ethernet

This was the first LAN protocol in history designed by Xerox in 1976 and made it available in 1980. The IEEE organisation started a project in 1985 called "Project 802" to define the link layer of the Internet stacks (OSI and TCP/IP). They subdivided the data-link in 2 sublayers:

- The *logical link control* (LLC): similar to our already known DLC. LLC is optional if the upper-layer protocols do not need them. For TCP/IP traffic, they are not usually required.
- The *media access control* (MAC): similar to our already known MAC. In Ethernet, this layer is always present.

Ethernet is one of the few protocols that have survived since the very beginning and today is common to find LANs at 10Gbps which is the forth generation. Ethernet is connectionless and unreliable:

- It does not require the creation of a connection.
- Frames can overwhelm the receiver and being discarded silently. Frames can also be corrupted and the will be discarded silently. If frames are discarded the sender will never know.

Note: It is up to the upper layers to provide reliability. IP nor UDP do it, so unless the application layer provides it, we can only rely on TCP to get a frame restransmitted.

6.2.1 Ethernet addresses

The Ethernet protocol defines 6 byte long (48 bits). They are represented as 6 bytes in hexadecimal separated by colons. These addresses are associated to the NICs (*network interface controller*) attached to the node. Each interface has its own MAC address. That way, a node can have multiple NICs attached to it each of them with a different MAC address. The MAC address identifies the NIC in a particular LAN.

The structure of a MAC address can be divided in 2 groups:

- The *organisationally unique identifier* (OUI): the first 3 bytes, that identifies a specific manufacturer. IEEE assigns them. When an OUI is used, then the last 2 bits of the first byte will be always `00`. All addresses using an OUI will be *globally unique*. If an administrator replaces the OUI and uses a *locally unique* identifier, then it is expected that these bits are `10`.
- The *NIC specific*: the last 3 bytes.

When a MAC address is used as destination address, the last bit of the first byte determines the type of the frame:

- *Unicast*: if the last bit is 0.
- *Multicast*: if the last bit is 1. A *broadcast* frame is really a special case of a multicast one as all bytes are set to `FF`.

The MAC addresses are written as 6 bytes separated by colons. However, they are *not* transmitted like that. Instead, each byte is transmitted in the reversed order so the LSB of each byte is sent first. This is why the last bits of the first byte contains the type of the frame: *they are transmitted and received first.*

Note: Important to note: even though a frame is unicast, that does not mean that, in a *shared medium* the frame is only received by the destination. In a shared medium like a bridged cable LAN or a WLAN environment, *all* nodes will receive the unicast frame. However, only the affected node will reply to it.

6.2.2 Ethernet MAC layer

The MAC sublayer of Ethernet provides a way to access to a *shared medium*. Shared media are problematic:

- Multiple nodes want to transmit at the same time. It requires a way to provide access to the transmission media in turn.
- Collisions: when 2 or more nodes transmit at the same time a *collision* is produced. This degrade the media utilisation, waste resources and increases the delay during communcation.

The solution to this problem proposed by Ethernet is to use a technique called *carrier sense multiple access with collision detection* (CSMA/CD). This mechanism to avoid collisions work as follows:

1. Nodes listen to the channel before start transmitting.
2. If it is used, then wait for a certain time.
3. If not used, then transmit and listen for other frames coming in. If you can only see your message, then OK. If not, then abort the transmission and as a collision has been detected. The frame will need to be resent again.

6.2.3 Ethernet frame

The standard Ethernet frame size goes from 64 to 1518 bytes. It can carry from 46 to 1500 bytes of data:

- *Preamble*: 8 bytes that is not counted as part of the frame legnth. The preamble is made of 8 bytes:
    1. First 7 bytes are `10101010` (55 decimal).
    2. Last byte is the *start frame delimiter* (SFD) and it is `10101011` (213).
- *Target address*: 6 bytes.
- *Source address*: 6 bytes.
- *Type*: 2 bytes defined by the IANA:
    1. From 0 to 45 are invalid values.
    2. From 46 to 1500 are reserved to encapsulate LLC frames (802.2). *In this case, the field indicates the length of the data field and not just the type.*
    3. From 1500+ uses values assigned by the IANA to define the type of the protocol encapsulated. For example `0x8000` is IPv4.
- *Data*: from 46 to 1500 bytes. If the data is lower than 46, then it is fill with 0s.
- *Frame check sequence*: the CRC for detecting transmission errors.

It is interesting that, if the `type` field value is between 46 and 1500, this field actually tells the length of the data field because it will carry an LLC frame encapsulated in it. These frames are useful for supporting upper layers protocols that:

- Protocols that do not have a length field for the data.
- Protocols that use flow control features like NetBIOS.
- Protocols that require the *subnetwork access protocol* (SNAP header) since they used private protocol identifiers.

In these situations, the LLC frame is the one carrying the data and because of some extra headers introduced by LLC, the minium and maximum data that they can carry changes from 46-1500 to 38-1492 (8 bytes less).

6.3 ARP

Note: Now it is time to revisit the encapsulation process from top (application) to bottom (data-link). Whatever message we generate at the application layer, we will have to send no less than 64 bytes at the link layer (14B of header and 46B of data encapsulating multiple and 4B as tail). This means that if the application layer generates 4 bytes of data, it will fit in this minimal frame size. Note that if not, the data will be filled with 0s.

Note: Let's suppose our PC sends an HTTP request to a web server:

1. The PC generates the message with the right source and destination IP addresses and, based in its routing table, it is sent to its gateway.
2. The router analyses the destination IP and, based in its routing table, forward the IP packet to its next hop.
3. The second router analyses the destination IP and, based in its routing table, locally delivers the packet to the web server.

The local delivery performed in the first and last step is performed within the same network, at link level. There must be a mechanism that transforms the destination IPs to the actual MAC addresses of the devices.

Address resolution protocol (ARP) is a neighbour discovery protocol that helps with the problem of getting the MAC address given an IP address. When a node needs the MAC address of a given IP for delivering a frame to a local neighbour it sends a broadcast request. The answers are stored in a cache for future use.

Note: The ARP cache can be managed using `arp` and `ip neigh` commands:

$ ip neigh 192.168.223.238 dev wlp0s20f3 lladdr 11:22:33:44:55:66 REACHABLE 172.17.0.2 dev docker0 lladdr 01:02:03:04:05:06 STALE

$ sudo ip neigh add 192.168.1.1 lladdr 66:11:66:11:66:11 dev wlp0s20f3

$ ip neigh 192.168.223.238 dev wlp0s20f3 lladdr 11:22:33:44:55:66 REACHABLE 192.168.1.1 dev wlp0s20f3 lladdr 66:11:66:11:66:11 PERMANENT 172.17.0.2 dev docker0 lladdr 01:02:03:04:05:06 STALE

$ sudo ip neigh del 192.168.1.1 dev wlp0s20f3

1. The *ARP request* is broadcasted to all the nodes. The request is for discovering the MAC address of a certain IP.
2. The *ARP reply* is unicast, from the node that has the matched IP. It includes its MAC address.

The ARP message is encapsulated in an Ethernet frame of type `0x0806` and has the following structure:

- *HW type*: 2 bytes. Tells the type of the link layer protocol. For Ethernet `0x0001` is used.
- *Network type*: 2 bytes. Tells the type of the network layer protocol. For IP `0x0800` is used.
- *HW len*: 1 byte. The length of the link-layer addresses. `0x06` for Ethernet.
- *Protocol len*: 1 byte. The length of the network-layer addresses. `0x04` for IP.
- *Operation type*: 2 bytes. The ARP operation. `0x0001` for request or `0x0002` for replies.
- *Sender HW address*: variable length.
- *Sender Network address*: variable length.
- *Target HW address*: variable length.
- *Target Network address*: variable length.

Note: In the example, node A wants to know the MAC address of B. It sends an ARP requests (the example needs to be read from the right to the left):

- The frame's detination address is the broadcast: `FF:FF:FF:FF:FF:FF`.
- The frame's type is ARP (`0x0806`).
- The data in the frame is an ARP message with the following content:
  - HW type and network type sets to Ethernet and IP. (there is a mistake in the slide: it is `0x0001` not `0x0002`)
  - HW len and protocol len are set to 6 and 4 because of the length of the Ethernet and IP addresses.
  - The operation type is `0x0001` for a request.
  - The sender's addresses are filled accordingly.
  - The target HW address is set to `00:00:00:00:00:00` as it is unkown

B replies with a unicast ARP reply and it contains its IP address (as part of the sender's protocol address).

6.4 Bridges and Switches

As we have seen before, connecting multiple devices requires sharing a transmission medium. That is problematic in many cases because of collisions. The use of CSMA/CD (and other techniques) reduces the chance of collisions. However, collisions may still happen and that reduces the utilisation of the channel.

Ethernet has evolved to provide better performance and in its earlier stages used *bridges* (also known as *hubs*) to connect multiple hosts and reduce the amount of collisions. *Switches* were introduced later as an improvement of bridges to avoid collisions even further.

6.4.1 Bridges

Bridges (hubs) are used to split a collision domain in multiple ones. They forward the frames coming from one collision domain (connected to one of the bridge *port*) to the other ones (connected to the rest of the ports).

Note: Let's suppose the 2 port hub of the slide:

1. A frame coming from the left side is forwarded to the right side to *all* hosts.
2. Only the target host generates a reponse and send the frame back to the left side.
3. The bridge forward the response to *all* the nodes at the left side.
4. Again, only the target host processes the response.

By separating the collision domains, bridges improve the bandwidth of the shared medium. For example, 6 nodes connected to a shared medium of `10Mbps` would have `10/6 = 1.6Mbps`. Adding a bridge of 2 ports, the collision domains can be split in 2, so each domain will have 3 hosts. Now, the bandwidth will be `10 / (3 + 1)` (as we should count the bridge itself too) which is `2.5Mbps`.

Bridges are not smart enough for *filtering*. In fact, they are physical-layer devices because they do not check the link layer addressing at all. It would be a great improvement if bridges could check the link layer so they send frames to the right ports.

6.4.2 Switches

Switches are link-layer devices beucase they check link layer addresses to perform a smarter forwarding of the frames. In order to do so, a *switch table* is required. This table tells, for each destination MAC address, what port needs to be used. However, this table starts empty when the switch starts and needs to be filled. Switches can learn to fill the table like this:

1. When a frame arrives, the switch writes down in the table the *source address* and the *port* from which it arrived. Incoming frames are useful for learning new paths.
2. The switch reads the *target address* and look for it in the table. If it is present, then the frame is sent through the designated port. If not, then send it through all the rest of ports.
3. The rows of the time are removed if they are not used for a certain period of time.

Note: In the slide, the table is built out of the frames that are being sent between hosts:

- A to D: A is learnt. The frame to D goes to ports 2 and 3.
- E to A: E is learnt. The frame is sent to port 1 because we knew A already.
- B to C: B is learnt. The frame is sent to ports 2 and 3.