Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Introduction
Personal Information

- Riccardo Caccavale, PhD
  - Ricercatore (RTD-A) presso il dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione (DIETI).
    - E-mail: riccardo.caccavale@unina.it
    - Web: https://www.docenti.unina.it/riccardo.caccavale
  - Research activities in the field of robotics, specifically in AI-based, cognitive, and autonomous robotics.
  - Member of the PRISMA Lab (from 2016).
    - Website: https://prisma.dieti.unina.it/index.php
  - Ricevimento:

Giorno	Ora	Luogo
Lunedì	10:00 – 11:00	Teams - Via Claudio 21, edificio 3/A, stanza 2.11

Previo appuntamento

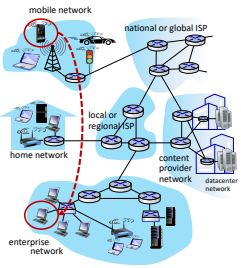
Introduction
About the Course

- Lessons calendar:
  - Thursday (Giovedì) 10:30 – 12:30, Via Claudio aula CL-II-1.
  - Friday (Venerdì) 16:30 – 18:30, Via Claudio aula CL-II-3.
- Teams:
  - Code *b51ai1p*
- Text books:
  - James Kurose, Keith Ross, Computer Networking A Top-Down Approach. [Main reference]
  - Andrew Tanenbaum, David Wetherall, Computer Networks.
- Additional Materials:
  - Slides (in English).

Introduction
Computer Networks and Internet

Computer networking: the process of connecting computer together so that they can share information.

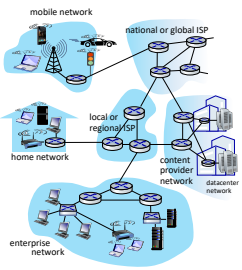
[Cambridge Dictionary]



- **Internet** is the most important and widespread computer network, not to mention probably the **largest engineered system ever created**.
  - Internet includes hundreds of millions of network **devices**, **links**, **computers**, etc. offering hundreds of **services** for the users.
- There are also **smaller networks** (local or detached from internet).

Introduction
Computer Networks and Internet

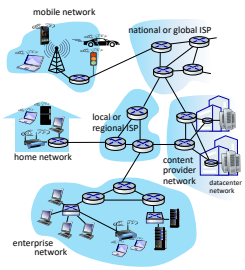
- In the past years, internet was mainly used to connect devices such as desktop PCs, workstations, and servers that store and transmit information such as Web pages and e-mail messages.
- Nowadays, not only computers are connected (laptops, smartphones, tablets, TVs, gaming consoles, home devices, etc.).
- Networking is now **pervasive:** internet connection is everywhere; complex devices may have networks of their own.



Introduction
Devices, Links, Hosts

- Network **devices** and **links** are the **infrastructure** that allow hosts to connect:

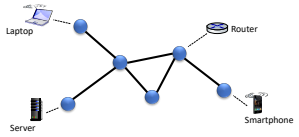
- **Hosts** are devices on which **applications** (or programs) runs:



Introduction
Computer Networks and Graph Theory

- Computer networks share terminologies with graph theory:
  - The devices connected through the network are called **nodes** while the connections between nodes are called communication **links** (or **channels**).
  - A **sequence** of nodes/links is called path.

- The end-points (or end-systems) of the network, which provide or use **services**, are special nodes called **hosts**.



Ideally hosts are the leaves of the network (often it is not true) while intermediate nodes are routing devices (routers, switches).

Introduction
Data Communication

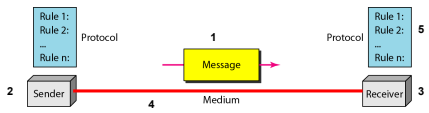
- The **goal of networking** is to share information: to allow large-scale internet functionalities as well as small-scale network functionalities we need to connect two hosts so they can **communicate data over distance**.

- The term **telecommunication** means communication at a distance. The word data refers to information presented in whatever form is agreed upon by the parties creating and using the data.

- **Data communication** is the process of exchanging data between two devices via some form of transmission medium such as a wire cable or wireless, ensuring a certain degree of:
  - **Reliability**: data is received correctly.
  - **Performance**: data is received within a reasonable amount of time.

Introduction
Data Communication

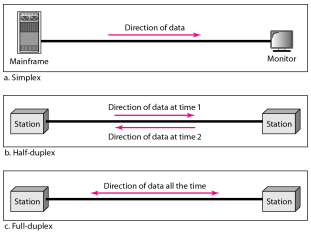
- Data communication has basically **five** components:
  1. **Message**: contains the **data** we want to communicate.
  2. **Sender**: the entity which is sending the message.
  3. **Receiver**: the entity which is supposed to receive the message.
  4. **Medium**: the channel between **sender** and **receiver** where the **message** travels.
  5. **Protocol**: a set of rules, known to **sender** and **receiver** used to manage the **message**.



Introduction
Data Representation and Flow

- The **data** we want to exchange can be represented in different forms.
  - Text, numbers, images, audio, video, etc.

- Depending on the type and the purpose of the communication, the **data flow** can be:
  - **Simplex**: monodirectional.
  - **Half-duplex**: bidirectional (taking turns).
  - **Full-duplex**: bidirectional (simultaneous).

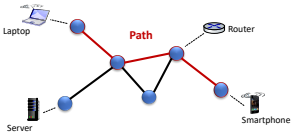


Introduction
Data Representation and Flow

- The **transmission rate** is the **maximum** amount of information that a **channel** can transmit and is measured in bits/secs (or bytes/secs).

- The **bandwidth** is the **maximum** amount of information that a **path** (links and nodes) can transmit, measured in **bits/secs** (or bytes/secs).

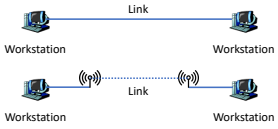
- The **throughput** is the **actual** (instantaneous) amount of information that a path or a link transmits, also measured in **bits/secs** (or bytes/secs).



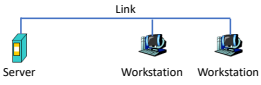
Introduction
Types of Connections

- The **hosts** of a network can be connected in different ways.

- **Point-to-point**: a dedicated link is provided between two devices (wireless or wired).

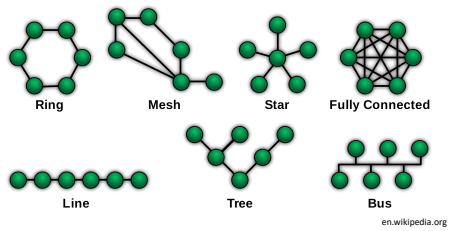


- **Multipoint** (**broadcast**): more than two specific devices share a single link.



Introduction
Network Topologies

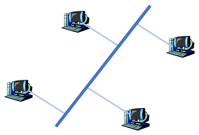
- **Network topology** is the arrangement of the elements (links, nodes, etc.) of a communication network.



en.wikipedia.org

Introduction
Network Topologies: Bus

- In the **Bus** topology hosts are connected to a central backbone (bus) cable.
  - Messages sent by 2 hosts generate **collisions**.

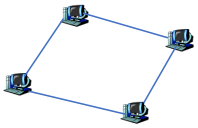


- Pros:
  - Simple and cheap.
  - Good for small networks.
- Cons:
  - Single point of failure (broken bus) but sub networks may still be available.

Bus networks have *1* physical duplex link (or *1* backbone and *n* links).

Introduction
Network Topologies: Ring

- In the **Ring** topology hosts are point-to-point connected to exactly two other ones. The signal is **forwarded** along the ring, from device to device, until it reaches its destination.

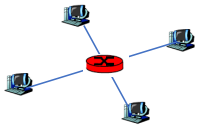


- Pros:
  - Simple and cheap.
  - Performs better than the bus.
- Cons:
  - Adding new nodes is harder.
  - Nodes malfunctioning may impair the network.

Ring networks have *n* duplex links.

Introduction
Network Topologies: Star

- In the **Star** topology hosts are linked to a central controller (hub, switch or router), there is no direct link between hosts.
- The central controller redirects messages.

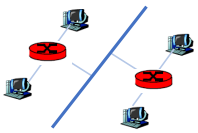


- Pros:
  - Less expensive, simple, robust, more scalable.
- Cons:
  - The controller must be reachable by all hosts.
  - Single point of failure.

Star networks have *1* controller and *n* physical duplex links!

Introduction
Network Topologies: Tree

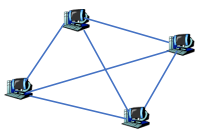
- In the **Tree** topology multiple star topologies are integrated typically through a bus cable.



- Pros:
  - Versatile, scalable, robust.
  - Well supported by HW and SW providers.
- Cons:
  - Hard to configure.
  - Weakness of the bus.

Introduction
Network Topologies: Mesh

- In the **Mesh** topology hosts are point-to-point linked in a non-hierarchical way.
  - **Full mesh**: all nodes connected (full-connected).
  - **Partial mesh**: nodes connected with some others.

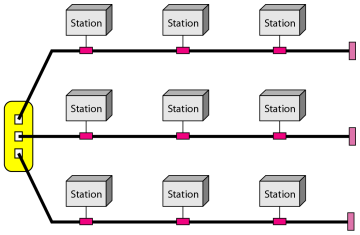


- Pros:
  - Low traffic, robust, secure, dedicated.
- Cons:
  - Hardly scalable.
  - Expansive (need for devices with multiple ports).

Full mesh networks have ${}^{n}C_{2} = n(n-1)/2$ physical duplex links!

Introduction
Network Topologies: Hybrid

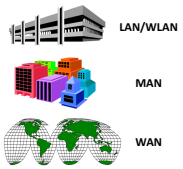
- Topologies can be **mixed** into a hybrid network.

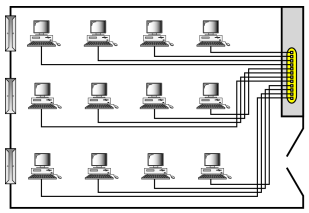


Star-backbone with 3 bus networks

Introduction
Categories of Networks

- Networks are categorized by dimension, number of hosts, bandwidth.
  - **Local Area Network** (LAN) or Wireless Local Area Network (WLAN).
  - **Metropolitan Area Network** (MAN).
  - **Wide Area Network** (WAN)

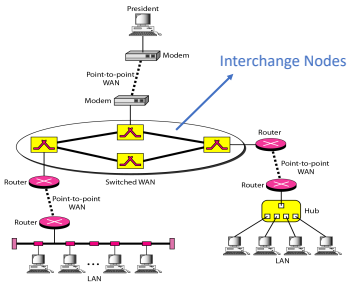




Example of LAN connecting 12 computers to a switch

Introduction
Complexity of WANs

- The complexity of the topology may increase with the increased dimension of the network.
- WANs connecting nations or continents can obviously be very complex and heterogeneous.



Example of heterogeneous WAN

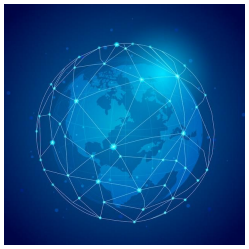
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.



Internet is network of networks. It is technically huge WAN that allows sub-networks and devices to be connected over different nations and continents

The idea of such "universal" big network was initially theorized by researchers (most notably J.C.R. Licklider) around 1950 and started to become reality almost 10 years later.

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).

Initial ARPANET having 3+1 nodes: SRI, UCLA, UCSB, and UTAH (1969)

The initial ARPANET was mainly a closed network designed to include universities and research centers. Similar networks were independently created in US and Europe.

Services
Internet

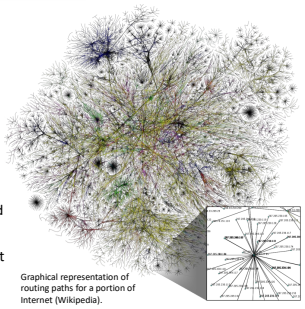
- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).
  - R. Kahn (ARPA) and V. Cerf (Stanford) designed the Transmission Control Protocol (TCP) and Internet Protocol (IP), two protocols of the **Internet protocol suite** (1974).
  - The National Science Foundation (NSF) awarded contracts for the **NSFNET** project, a TCP-IP based network (1986).

NSFNET in 1992 (Wikipedia).

The NSFNET and the TCP/IP protocol went toward the idea of a *network of networks*. Both ARPANET and NSFNET carried **commercial restrictions**.

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).
  - R. Kahn (ARPA) and V. Cerf (Stanford) designed the Transmission Control Protocol (TCP) and Internet Protocol (IP), two protocols of the **Internet protocol suite** (1974).
  - The National Science Foundation (NSF) awarded contracts for the **NSFNET** project, a TCP-IP based network (1986).
  - The **ARPANET decommissioned** (1990).
  - The **NSFNET decommissioned**, removing the last restrictions on the use of the Internet to carry commercial traffic (1995).



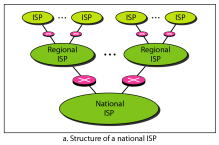
Graphical representation of routing paths for a portion of Internet (Wikipedia).

Services
Introduction

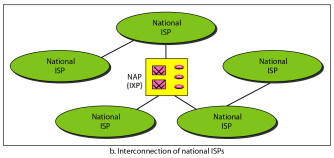
- The main role of computer networks is to allow data exchange between multiple devices from different locations, but raw data are just a mean to reach **services** and **resources** provided by the network.
- A network **resource** is a remote "object" we want to have access to (web pages, storage, computation, video or audio files, etc.).
- A network **service** is an "action" that a remote device performs for us (give me the current time, let me send an e-mail, let me store files, etc.).
- The difference between these two definitions is somehow subtle (e.g., let me *see* a *video*, *give* me access to a *web page*, are actions involving objects).
- Entities who offers services on Internet are called **Service Providers**.

Services
Internet Service Providers

- **Internet access** is the basic service.
- An **Internet Service Provider** (ISP) is an organization that provides services for accessing, using, or participating the Internet.
- ISPs can be organized as commercial, community-owned, non-profit, or privately owned (e.g., by private companies or universities).
- Different ISPs exchange data through **Network (neutral) Access Points** (NAPs) or **Internet Exchange Points** (IXPs).



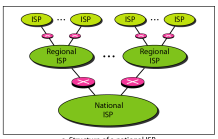
a. Structure of a national ISP



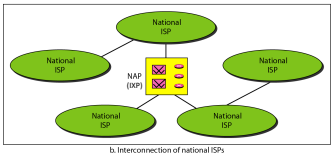
b. Interconnection of national ISPs

Services
Internet Service Providers

- An ISP network is hierarchically defined:
  - A **point of presence** (PoP) is a group of one or more routers used by ISPs to reach the customers.
  - **Access** ISPs for local areas.
  - **Regional** ISPs for larger areas.
  - **National** ISPs for nations.
- An **Internet exchange point** (IXP) works as a meeting point between multiple ISPs, typically managed by third parties.



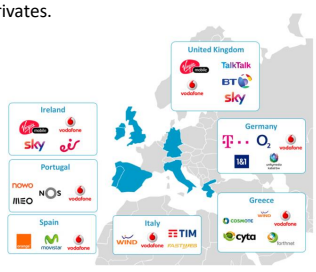
a. Structure of a national ISP



b. Interconnection of national ISPs

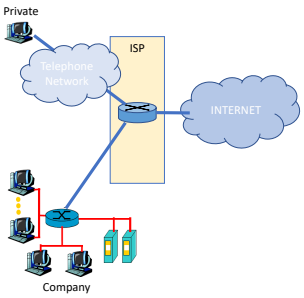
Services
Example of ISPs

- On a national scale there can be several commercial ISPs which provides internet services to companies and privates.
- Some Italian ISPs are:
  - Telecom Italia.
  - Fastweb.
  - Vodafone.
  - Tiscali.
  - ...



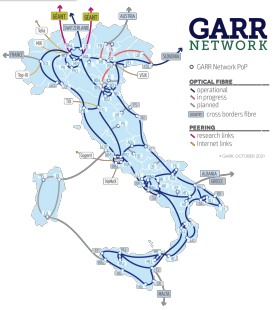
Services
Internet Connection

- Private internet connections are established via **Modem** enabled by the telephone network.
  - Analogic (56kbps).
  - Integrated Services Digital Network ISDN (128kbps).
  - Asymmetric Digital Subscriber Line ADSL (1Mbps to 20Mbps).
  - Twisted-Pair Copper Wire (10Mbps to 100Mbps).
  - Optical Fiber (50Mbps to 40Gbps).
- Companies (especially medium/large ones) can have a direct/dedicated connection to the ISP.



Services
Example of the GARR Network

- The internet access of Italian universities is managed by GARR.
- GARR (Gruppo per l'Armonizzazione delle Reti della Ricerca) is the Italian national computer network for universities and research.
- The GARR network is connected to other national research and education networks in Europe and the world, is an integral part of the global Internet.



Services
Sharing Services and Resources

- The main goal of computer networks (and internet) is **to share services** or **resources** between different devices (and users beyond devices).
- There is a variety of **services** enabled by computer networks:
  - **File sharing**: moving, reading, copying files from remote hosts over the network.
  - **E-mails**: sending/receiving electronic mails.
  - **Instant messaging**: sending/receiving messages on-line.
  - **Video telephony**: organize and participate to calls.
  - **Web pages**: watching/showing hypertextual pages.
  - **Hardware management**: remote control (printing, surveillance, domotic systems, etc.).
  - **Hardware sharing**: using/providing hardware (CPU, memory, etc.) of/to remote hosts.
  - **Application sharing**: using/offering applications of/to remote hosts.
  - …

Services
Sharing Services on Local Networks

- Resources can be **private** and locally available: for instance, in local networks, there can be devices (printers, air conditioners, etc.) that offer services and devices (tablet, laptop, smartphone) that uses them.
- It could be possible to use such services (for instance, printing a document or turning on air conditioning) even **without internet access**.
- However, most of todays services are **public** and provided through Internet by Service Providers.

Services
Service Providers Beyond Connectivity

- There are several famous companies whose main business is to provide services over Internet:
  - Amazon (AWS),
  - Google,
  - Microsoft (Azure),
  - IBM,
  - Oracle,
  - Aruba,
  - Alibaba,
  - …
- There are different ways of providing services.

Services
Dedicated Servers

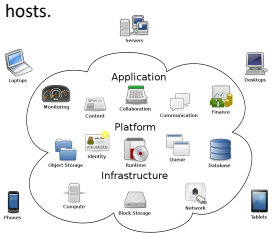
- Resources and services can be public and provided by distant **servers**.
  - A **server** is a special host designed to provide one or multiple **services**.
- For example, a web server (web pages) or an e-mail server may be on the other side of the world, but we are *almost unaware of this distance*.
- Moreover, there could be multiple servers cooperating to provide services and we would be again *unaware* of that.

Services
Grid Computing

- The **grid computing** is a decentralized resource-sharing infrastructure that typically combines hardware (resources) from different hosts in different geographic locations to achieve a common goal:
  - to perform complex calculations.
  - to store big quantities of data.
- A famous example is **SETI** (Search for Extra-Terrestrial Intelligence - 1999).
- The **SETI@home** project allows PCs all over the world shares resources to analyze radio signals from space, searching for extraterrestrial intelligence.

Services
Cloud Computing

- The **cloud computing** is centralized architecture in which resources (hardware and software), typically managed by companies (service providers), are offered on-demand as services to external hosts.
- In a cloud architecture several services can be offered:
  - Applications for productivity, games, communication, social networks, etc.
  - Storage, shared databases, web servers, etc.
  - Virtual machines (with specific configurations), servers, firewalls, etc.



Services
Cloud Computing

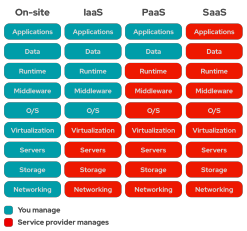
- A widely used definition of cloud computing is provided by the **NIST** (U.S. National Institute of Standards and Technology).
- **Cloud computing** is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
- This cloud model is composed of five essential characteristics:
  1. On-demand self-service.
  2. Broad network access.
  3. Resource pooling.
  4. Rapid elasticity.
  5. Measured service.

Services
Cloud Computing

1. **On-demand self-service**. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
2. **Broad network access**. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
3. **Resource pooling**. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.
4. **Rapid elasticity**. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
5. **Measured service**. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

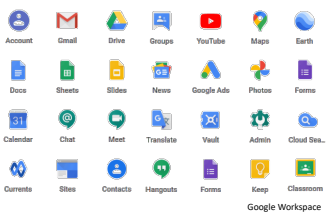
Services
Cloud Computing: Service Models

- The NIST grouped services provided by a cloud computing architecture in 3 categories (**service models**) depending on how much of the computing infrastructure is managed by the service provider.
- **Software as a Service** (SaaS): whole applications provided along with software/hardware components needed to run them.
- **Platform as a Service** (PaaS): working platforms provided with specific configurations (e.g., OS, APIs, etc.).
- **Infrastructure as a Service** (IaaS): mainly the hardware is provided.



Services
Example of Cloud Service Provider

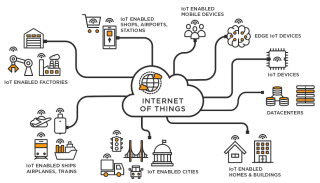
- Google is an example of Cloud Service Provider (CSP).
  - The most popular services are offered through the Google Workspace (mostly SaaS).
  - Additional services are available as part of the Google Cloud infrastructure (100+ available services).



Services
Internet of Things

- The **internet of things** (IoT) is the approach of endowing simple devices (smart devices) with sensors, processing units, connectivity, etc. in order to control/monitor them remotely over internet.

- IoT can be used in combination with cloud computing to:
  - On-line manage devices using graphical user interfaces (GUIs) and remote applications.
  - Store data and information from devices.
  - Elaborate information by means of predictive algorithms or machine learning techniques.



Services
Internet of Things

- Several "things" that are capable of internet connection have already been integrated in our lives:
  - Smartphones
  - Smartwatches
  - Home Assistants
  - Smart TV
  - Smart Cars
  - …





- Connected things particularly emphasize issues about *privacy* and *security*.

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Standardization
Needs for Standards and Protocols

- Computer networks (such as Internet) can be quite **complex and widespread** over different locations, countries, users, etc.

- Clearly, a **set of rules** or a common ground should be defined so that all participants know how **to provide or to use services**.

- From the beginning of Internet, one of the major effort has been made to define **protocols** and **standards** that regulate communication.

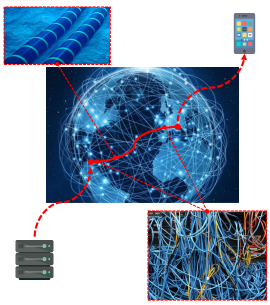

The Tower of Babel by Pieter Bruegel the Elder (1563).

Standardization
Needs for Standards and Protocols

- The communication between devices involves **several problems**:
  - Physical transportation, addressing, error check, data conversion and regulation, security, synchronization, etc.

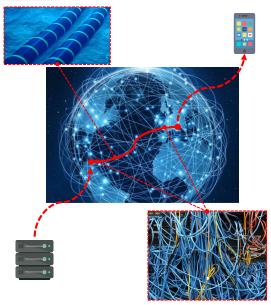
- To allow the communication both the receiver and the transmitter must agree to a **protocol**.

- When multiple/different devices from different places communicate they all must to agree on a common protocol, which becomes a **standard**.



Standardization
Needs for Standards and Protocols

- There are **hundreds of protocols** which regulate all aspects of communication from physical links to applications:
  - How **cables and links** should be created (materials, shielding, frequencies, etc.).
  - How **addresses** should be assigned.
  - How data should be wrapped into **packets or frames** for transmission.
  - How **errors** should be detected or corrected.
  - How **applications** should exchange data.
  - …



Standardization
Who Standardize Standards?

- Todays Internet standards and protocols are defined by a community of experts called **Internet Engineering Task Force** (IETF).

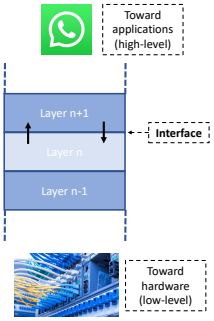
- IETF is typically organized in **open working groups**, each one focused on specific aspects of internet, whose members interacts by mailing list and meetings.

- Each one of these working groups produces numbered documents called **Request For Comments** (RFC) including description and definition of protocols, concepts, methods underlying a standard.
  - For example, the IPv4 protocol was defined in RFC 791 (1981).

Standardization
Network Models: Layered Model

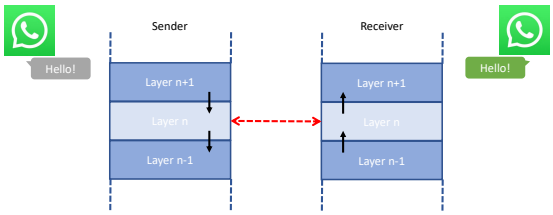
- A reasonable approach to face the different communication problems is to design a **layered model** (*Divide et Impera*):
  - Each layer is conceptually **responsible for one specific task** (solves one problem).
  - Each layer rely on services from the **lower layer** and provides services to the **upper layer**.

- Pros:
  - **Modularity**: layers are simple and independent
- Cons:
  - **Scalability**: climbing too much layers is inefficient



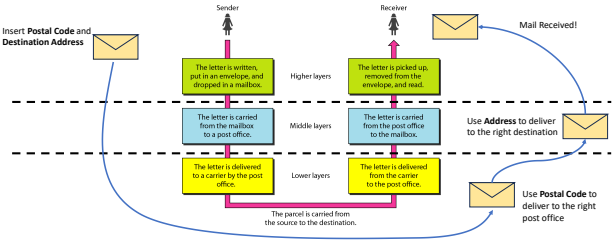
Standardization
Network Models: Layered Model

- When two devices communicate, **each layer on the sender communicates with the same layer on the receiver** by means of a specific protocol.



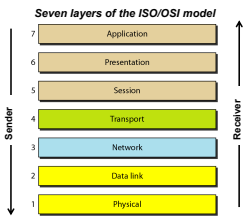
Standardization
Network Models: Layered Model (example)

- When two devices communicate, **each layer on the sender communicates with the same layer on the receiver** by means of a specific protocol.



Standardization
Network Models: The OSI Model

- In the '80 the **ISO** (International Standards Organization) defined a layered model for computer networks: the **OSI** (Open System Interconnection) model.
- The ISO/OSI model includes 7 layers:
  1. Physical transmission of raw bits.
  2. Data format definition (frames).
  3. Routing of messages through the net.
  4. Transmission protocols (TCP, UDP).
  5. Management of ports and sessions (continuity of data stream).
  6. Translation of messages (encoding, compression, decryption, etc.).
  7. Data use and human-computer interaction (file sharing, emails, streaming videos, etc.).

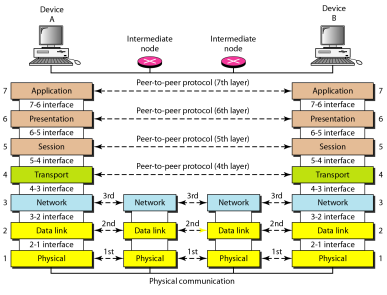


Standardization
Network Models: The OSI Model

- Not always all layers are implemented.

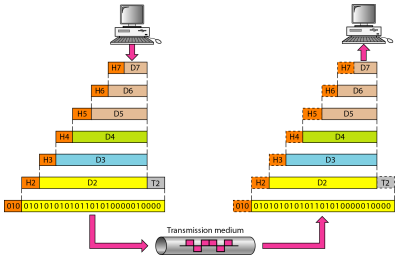
- Typically, in intermediate nodes (switches, routers), **only 2 or 3 layers** are implemented (media layers).

- Only the **end-points implement the whole stack** up to the application layers.



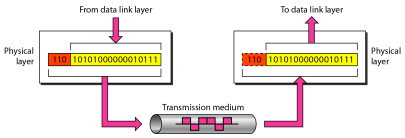
Standardization
Network Models: The OSI Model

- **Encapsulation** (top-down): every layer of the sender adds a layer-specific field to the message (payload) in the form of a header (H) or trailer (T).
- **Decapsulation** (bottom-up): Those fields are removed and interpreted by the same layer in the receiver.



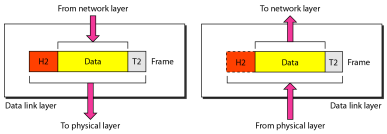
Standardization
OSI Model: 1. Physical Layer

- The **physical layer** is responsible for movements of individual bits from one hop (node) to the next.
  - Physical characteristics of interfaces and media (connectors, cables, electric signals)
  - Line configuration (point-to-point or multipoint)
  - Physical topology (mesh, star, ring or bus)
  - Transmission mode (simplex, half-duplex or duplex)
  - Representation of bits
  - Data rate
  - Synchronization of bits



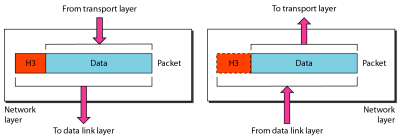
Standardization
OSI Model: 2. Data Link Layer

- The **data link layer** is responsible for moving frames from one hop (node) to the next (segment).
  - Works on frames (portion of data, typically few hundreds of bytes)
  - Flow and error control (frame control sequences)
  - Access control



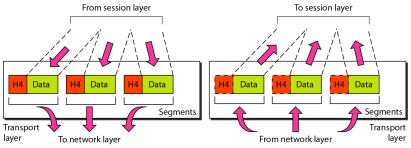
Standardization
OSI Model: 3. Network Layer

- The **network layer** is responsible for the delivery of individual packets from the source host to the destination host (path).
  - Works on packets (typically larger and more complex than frames)
  - Source-to-destination delivery: packets from the source to the destination.
  - Logical addressing
  - Routing (routing tables)



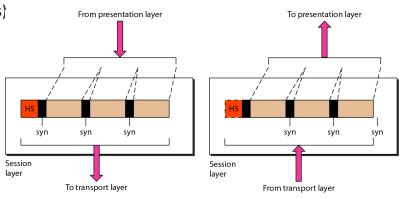
Standardization
OSI Model: 4. Transport Layer

- The **transport layer** is responsible for the delivery of a message from one end to another (message is received by the right program).
  - End-to-end delivery
  - Connection control (Connection-oriented or connection-less)
  - Segmentation/reassembly (to/from packets of layer 3)
  - Port addressing
  - Flow and error control



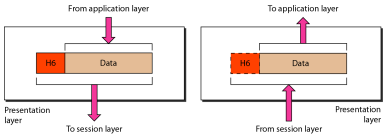
Standardization
OSI Model: 5. Session Layer

- The **session layer** is responsible for dialog control and synchronization of request/response.
  - It establishes, maintains and synchronize the interaction between communicating system (communication session).
  - Dialog control and management
  - Synchronization (checkpoints)



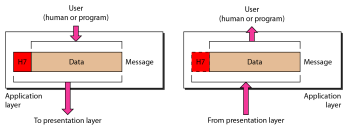
Standardization
OSI Model: 6. Presentation Layer

- The **presentation layer** is responsible for the representation of data, i.e., translation, compression, encryption, etc.
  - Translation (e.g., EBCDIC-coded or ASCII-coded to text)
  - Encryption and Decryption
  - Compression



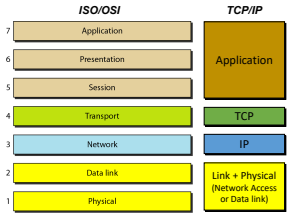
Standardization
OSI Model: 7. Application Layer

- The **application layer** is responsible for providing services to the user.
  - Network virtual terminal (Remote log-in)
  - File transfer and access
  - Mail services
  - Accessing the World Wide Web
  - …



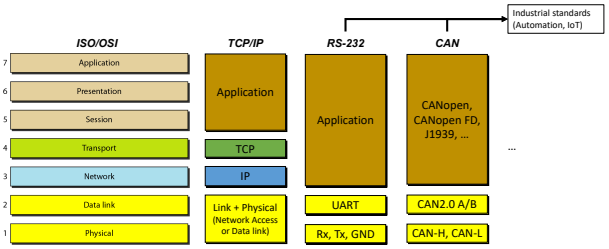
Standardization
Network Models: The TCP/IP Model

- The ISO/OSI model is *de iure* the standard stack for computer networks, but it is quite complex and detailed.
- The TCP/IP (aka Internet Protocol Suite) is the set of communication protocols **actually used** on internet and local networks.
  - TCP: Transmission Control Protocol.
  - IP: Internet Protocol.
- The TCP/IP is *de facto* the standard stack for internet communication.



Standardization
Network Models: Other Protocols

- There are different protocols for device communication, which can be used depending on the situation, that are still relying on the layered model.



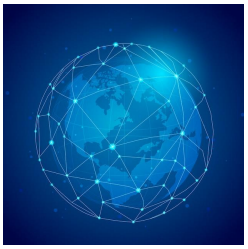
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.



Internet is network of networks. It is technically huge WAN that allows sub-networks and devices to be connected over different nations and continents

The idea of such "universal" big network was initially theorized by researchers (most notably J.C.R. Licklider) around 1950 and started to become reality almost 10 years later.

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).

Initial ARPANET having 3+1 nodes: SRI, UCLA, UCSB, and UTAH (1969)

The initial ARPANET was mainly a closed network designed to include universities and research centers. Similar networks were independently created in US and Europe.

Services
Internet

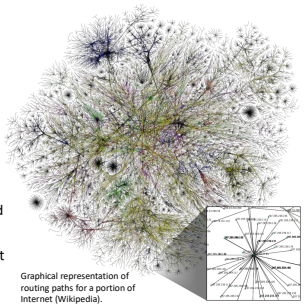
- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).
  - R. Kahn (ARPA) and V. Cerf (Stanford) designed the Transmission Control Protocol (TCP) and Internet Protocol (IP), two protocols of the **Internet protocol suite** (1974).
  - The National Science Foundation (NSF) awarded contracts for the **NSFNET** project, a TCP-IP based network (1986).


NSFNET in 1992 (Wikipedia).

The NSFNET and the TCP/IP protocol went toward the idea of a *network of networks*. Both ARPANET and NSFNET carried **commercial restrictions**.

Services
Internet

- The **Internet** is the ultimate WAN, connecting devices all around the globe.
- Brief history of Internet:
  - The Advanced Research Projects Agency (ARPA) of the U.S. Department of Defense awarded contracts for the development of the **ARPANET** project (1969).
  - R. Kahn (ARPA) and V. Cerf (Stanford) designed the Transmission Control Protocol (TCP) and Internet Protocol (IP), two protocols of the **Internet protocol suite** (1974).
  - The National Science Foundation (NSF) awarded contracts for the **NSFNET** project, a TCP-IP based network (1986).
  - The **ARPANET decommissioned** (1990).
  - The **NSFNET decommissioned**, removing the last restrictions on the use of the Internet to carry commercial traffic (1995).

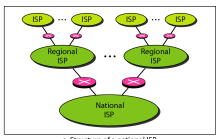

Graphical representation of routing paths for a portion of Internet (Wikipedia).

Services
Introduction

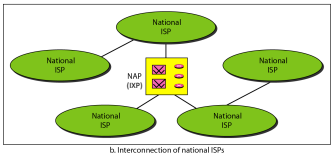
- The main role of computer networks is to allow data exchange between multiple devices from different locations, but raw data are just a mean to reach **services** and **resources** provided by the network.
- A network **resource** is a remote "object" we want to have access to (web pages, storage, computation, video or audio files, etc.).
- A network **service** is an "action" that a remote device performs for us (give me the current time, let me send an e-mail, let me store files, etc.).
- The difference between these two definitions is somehow subtle (e.g., let me *see* a *video*, *give* me access to a *web page*, are actions involving objects).
- Entities who offers services on Internet are called **Service Providers**.

Services
Internet Service Providers

- **Internet access** is the basic service.
- An **Internet Service Provider** (ISP) is an organization that provides services for accessing, using, or participating the Internet.
- ISPs can be organized as commercial, community-owned, non-profit, or privately owned (e.g., by private companies or universities).
- Different ISPs exchange data through **Network (neutral) Access Points** (NAPs) or **Internet Exchange Points** (IXPs).



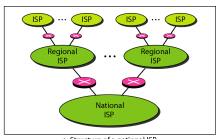
a. Structure of a national ISP



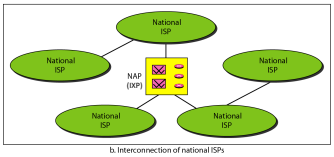
b. Interconnection of national ISPs

Services
Internet Service Providers

- An ISP network is hierarchically defined:
  - A **point of presence** (PoP) is a group of one or more routers used by ISPs to reach the customers.
  - **Access** ISPs for local areas.
  - **Regional** ISPs for larger areas.
  - **National** ISPs for nations.
- An **Internet exchange point** (IXP) works as a meeting point between multiple ISPs, typically managed by third parties.



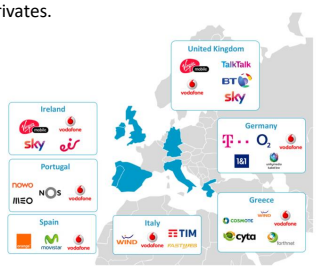
a. Structure of a national ISP



b. Interconnection of national ISPs

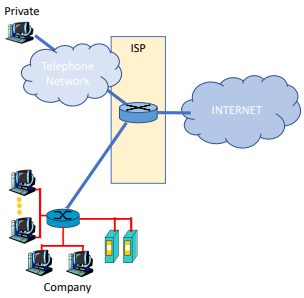
Services
Example of ISPs

- On a national scale there can be several commercial ISPs which provides internet services to companies and privates.
- Some Italian ISPs are:
  - Telecom Italia.
  - Fastweb.
  - Vodafone.
  - Tiscali.
  - …



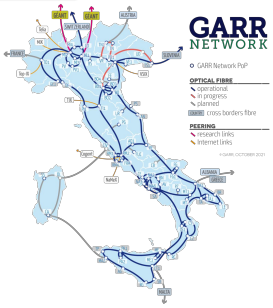
Services
Internet Connection

- Private internet connections are established via **Modem** enabled by the telephone network.
  - Analogic (56kbps).
  - Integrated Services Digital Network ISDN (128kbps).
  - Asymmetric Digital Subscriber Line ADSL (1Mbps to 20Mbps).
  - Twisted-Pair Copper Wire (10Mbps to 100Mbps).
  - Optical Fiber (50Mbps to 40Gbps).
- Companies (especially medium/large ones) can have a direct/dedicated connection to the ISP.



Services
Example of the GARR Network

- The internet access of Italian universities is managed by GARR.
- GARR (Gruppo per l'Armonizzazione delle Reti della Ricerca) is the Italian national computer network for universities and research.
- The GARR network is connected to other national research and education networks in Europe and the world, is an integral part of the global Internet.



Services
Sharing Services and Resources

- The main goal of computer networks (and internet) is **to share services** or **resources** between different devices (and users beyond devices).
- There is a variety of **services** enabled by computer networks:
  - **File sharing**: moving, reading, copying files from remote hosts over the network.
  - **E-mails**: sending/receiving electronic mails.
  - **Instant messaging**: sending/receiving messages on-line.
  - **Video telephony**: organize and participate to calls.
  - **Web pages**: watching/showing hypertextual pages.
  - **Hardware management**: remote control (printing, surveillance, domotic systems, etc.).
  - **Hardware sharing**: using/providing hardware (CPU, memory, etc.) of/to remote hosts.
  - **Application sharing**: using/offering applications of/to remote hosts.
  - …

Services
Sharing Services on Local Networks

- Resources can be **private** and locally available: for instance, in local networks, there can be devices (printers, air conditioners, etc.) that offer services and devices (tablet, laptop, smartphone) that uses them.
- It could be possible to use such services (for instance, printing a document or turning on air conditioning) even **without internet access**.
- However, most of todays services are **public** and provided through Internet by Service Providers.

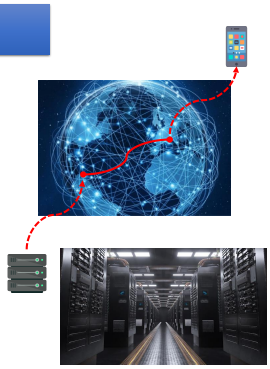
Services
Service Providers Beyond Connectivity

- There are several famous companies whose main business is to provide services over Internet:
  - Amazon (AWS),
  - Google,
  - Microsoft (Azure),
  - IBM,
  - Oracle,
  - Aruba,
  - Alibaba,
  - …

- There are different ways of providing services.

Services
Dedicated Servers

- Resources and services can be public and provided by distant **servers**.
  - A **server** is a special host designed to provide one or multiple **services**.
- For example, a web server (web pages) or an e-mail server may be on the other side of the world, but we are *almost unaware of this distance*.
- Moreover, there could be multiple servers cooperating to provide services and we would be again *unaware* of that.

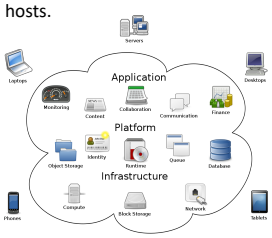


Services
Grid Computing

- The **grid computing** is a decentralized resource-sharing infrastructure that typically combines hardware (resources) from different hosts in different geographic locations to achieve a common goal:
  - to perform complex calculations.
  - to store big quantities of data.
- A famous example is **SETI** (Search for Extra-Terrestrial Intelligence - 1999).
- The **SETI@home** project allows PCs all over the world shares resources to analyze radio signals from space, searching for extraterrestrial intelligence.

Services
Cloud Computing

- The **cloud computing** is centralized architecture in which resources (hardware and software), typically managed by companies (service providers), are offered on-demand as services to external hosts.
- In a cloud architecture several services can be offered:
  - Applications for productivity, games, communication, social networks, etc.
  - Storage, shared databases, web servers, etc.
  - Virtual machines (with specific configurations), servers, firewalls, etc.



Services
Cloud Computing

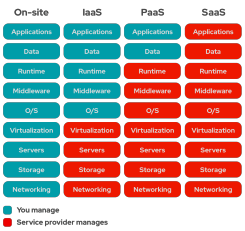
- A widely used definition of cloud computing is provided by the **NIST** (U.S. National Institute of Standards and Technology).
- **Cloud computing** is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
- This cloud model is composed of five essential characteristics:
  1. On-demand self-service.
  2. Broad network access.
  3. Resource pooling.
  4. Rapid elasticity.
  5. Measured service.

Services
Cloud Computing

1. **On-demand self-service**. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
2. **Broad network access**. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
3. **Resource pooling**. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.
4. **Rapid elasticity**. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
5. **Measured service**. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

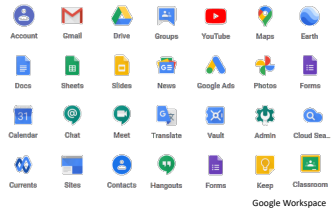
Services
Cloud Computing: Service Models

- The NIST grouped services provided by a cloud computing architecture in 3 categories (**service models**) depending on how much of the computing infrastructure is managed by the service provider.
- **Software as a Service** (SaaS): whole applications provided along with software/hardware components needed to run them.
- **Platform as a Service** (PaaS): working platforms provided with specific configurations (e.g., OS, APIs, etc.).
- **Infrastructure as a Service** (IaaS): mainly the hardware is provided.



Services
Example of Cloud Service Provider

- Google is an example of Cloud Service Provider (CSP).
  - The most popular services are offered through the Google Workspace (mostly SaaS).
  - Additional services are available as part of the Google Cloud infrastructure (100+ available services).



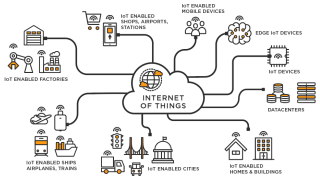
Google Workspace

Groups of Google Cloud services

Services
Internet of Things

- The **internet of things** (IoT) is the approach of endowing simple devices (smart devices) with sensors, processing units, connectivity, etc. in order to control/monitor them remotely over internet.

- IoT can be used in combination with cloud computing to:
  - On-line manage devices using graphical user interfaces (GUIs) and remote applications.
  - Store data and information from devices.
  - Elaborate information by means of predictive algorithms or machine learning techniques.



Services
Internet of Things

- Several "things" that are capable of internet connection have already been integrated in our lives:
  - Smartphones
  - Smartwatches
  - Home Assistants
  - Smart TV
  - Smart Cars
  - …





- Connected things particularly emphasize issues about *privacy* and *security*.

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Standardization
Needs for Standards and Protocols

- Computer networks (such as Internet) can be quite **complex and widespread** over different locations, countries, users, etc.

- Clearly, a **set of rules** or a common ground should be defined so that all participants know how **to provide or to use services**.

- From the beginning of Internet, one of the major effort has been made to define **protocols** and **standards** that regulate communication.

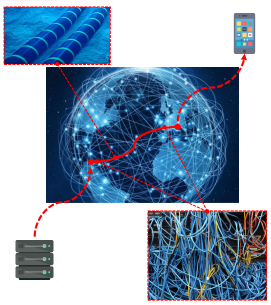

The Tower of Babel by Pieter Bruegel the Elder (1563).

Standardization
Needs for Standards and Protocols

- The communication between devices involves **several problems**:
  - Physical transportation, addressing, error check, data conversion and regulation, security, synchronization, etc.

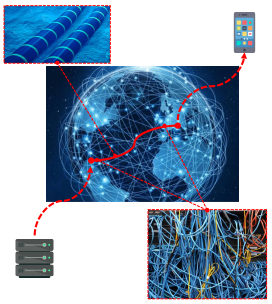
- To allow the communication both the receiver and the transmitter must agree to a **protocol**.

- When multiple/different devices from different places communicate they all must to agree on a common protocol, which becomes a **standard**.



Standardization
Needs for Standards and Protocols

- There are **hundreds of protocols** which regulate all aspects of communication from physical links to applications:
  - How **cables and links** should be created (materials, shielding, frequencies, etc.).
  - How **addresses** should be assigned.
  - How data should be wrapped into **packets or frames** for transmission.
  - How **errors** should be detected or corrected.
  - How **applications** should exchange data.
  - …



Standardization
Who Standardize Standards?

- Todays Internet standards and protocols are defined by a community of experts called **Internet Engineering Task Force** (IETF).

I E T F

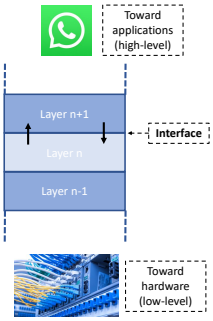
- IETF is typically organized in **open working groups**, each one focused on specific aspects of internet, whose members interacts by mailing list and meetings.

- Each one of these working groups produces numbered documents called **Request For Comments** (RFC) including description and definition of protocols, concepts, methods underlying a standard.
  - For example, the IPv4 protocol was defined in RFC 791 (1981).

Standardization
Network Models: Layered Model

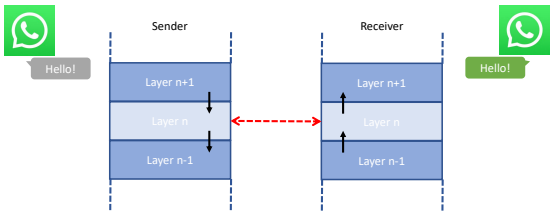
- A reasonable approach to face the different communication problems is to design a **layered model** (*Divide et Impera*):
  - Each layer is conceptually **responsible for one specific task** (solves one problem).
  - Each layer rely on services from the **lower layer** and provides services to the **upper layer**.

- Pros:
  - **Modularity**: layers are simple and independent
- Cons:
  - **Scalability**: climbing too much layers is inefficient



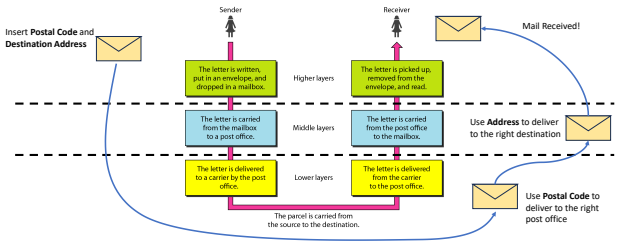
Standardization
Network Models: Layered Model

- When two devices communicate, **each layer on the sender communicates with the same layer on the receiver** by means of a specific protocol.



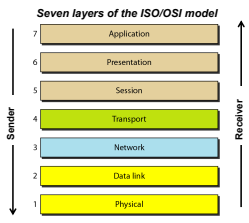
Standardization
Network Models: Layered Model (example)

- When two devices communicate, **each layer on the sender communicates with the same layer on the receiver** by means of a specific protocol.



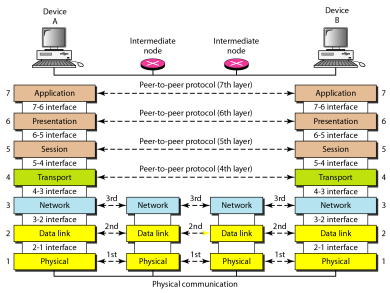
Standardization
Network Models: The OSI Model

- In the '80 the **ISO** (International Standards Organization) defined a layered model for computer networks: the **OSI** (Open System Interconnection) model.
- The ISO/OSI model includes 7 layers:
  1. Physical transmission of raw bits.
  2. Data format definition (frames).
  3. Routing of messages through the net.
  4. Transmission protocols (TCP, UDP).
  5. Management of ports and sessions (continuity of data stream).
  6. Translation of messages (encoding, compression, decryption, etc.).
  7. Data use and human-computer interaction (file sharing, emails, streaming videos, etc.).



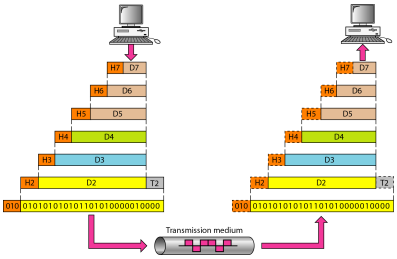
Standardization
Network Models: The OSI Model

- Not always all layers are implemented.
- Typically, in intermediate nodes (switches, routers), **only 2 or 3 layers** are implemented (media layers).
- Only the **end-points implement the whole stack** up to the application layers.



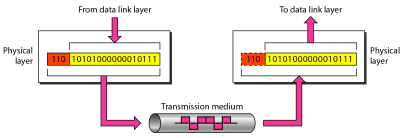
Standardization
Network Models: The OSI Model

- **Encapsulation** (top-down): every layer of the sender adds a layer-specific field to the message (payload) in the form of a header (H) or trailer (T).
- **Decapsulation** (bottom-up): Those fields are removed and interpreted by the same layer in the receiver.



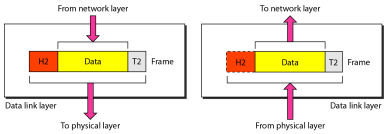
Standardization
OSI Model: 1. Physical Layer

- The **physical layer** is responsible for movements of individual bits from one hop (node) to the next.
  - Physical characteristics of interfaces and media (connectors, cables, electric signals)
  - Line configuration (point-to-point or multipoint)
  - Physical topology (mesh, star, ring or bus)
  - Transmission mode (simplex, half-duplex or duplex)
  - Representation of bits
  - Data rate
  - Synchronization of bits



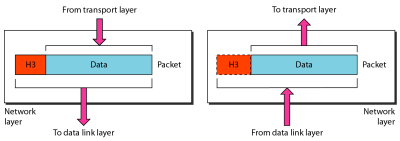
Standardization
OSI Model: 2. Data Link Layer

- The **data link layer** is responsible for moving frames from one hop (node) to the next (segment).
  - Works on frames (portion of data, typically few hundreds of bytes)
  - Flow and error control (frame control sequences)
  - Access control



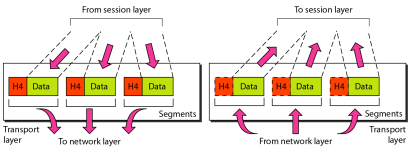
Standardization
OSI Model: 3. Network Layer

- The **network layer** is responsible for the delivery of individual packets from the source host to the destination host (path).
  - Works on packets (typically larger and more complex than frames)
  - Source-to-destination delivery: packets from the source to the destination.
  - Logical addressing
  - Routing (routing tables)



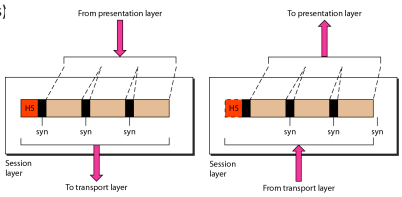
Standardization
OSI Model: 4. Transport Layer

- The **transport layer** is responsible for the delivery of a message from one end to another (message is received by the right program).
  - End-to-end delivery
  - Connection control (Connection-oriented or connection-less)
  - Segmentation/reassembly (to/from packets of layer 3)
  - Port addressing
  - Flow and error control



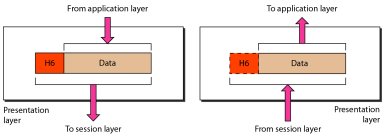
Standardization
OSI Model: 5. Session Layer

- The **session layer** is responsible for dialog control and synchronization of request/response.
  - It establishes, maintains and synchronize the interaction between communicating system (communication session).
  - Dialog control and management
  - Synchronization (checkpoints)



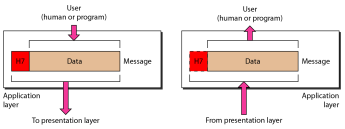
Standardization
OSI Model: 6. Presentation Layer

- The **presentation layer** is responsible for the representation of data, i.e., translation, compression, encryption, etc.
  - Translation (e.g., EBCDIC-coded or ASCII-coded to text)
  - Encryption and Decryption
  - Compression



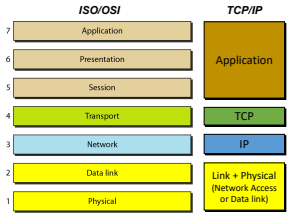
Standardization
OSI Model: 7. Application Layer

- The **application layer** is responsible for providing services to the user.
  - Network virtual terminal (Remote log-in)
  - File transfer and access
  - Mail services
  - Accessing the World Wide Web
  - …



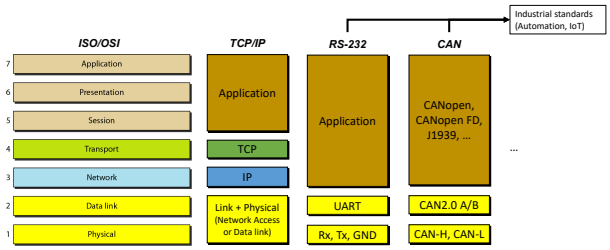
Standardization
Network Models: The TCP/IP Model

- The ISO/OSI model is *de iure* the standard stack for computer networks, but it is quite complex and detailed.
- The TCP/IP (aka Internet Protocol Suite) is the set of communication protocols **actually used** on internet and local networks.
  - TCP: Transmission Control Protocol.
  - IP: Internet Protocol.
- The TCP/IP is *de facto* the standard stack for internet communication.



Standardization
Network Models: Other Protocols

- There are different protocols for device communication, which can be used depending on the situation, that are still relying on the layered model.



Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

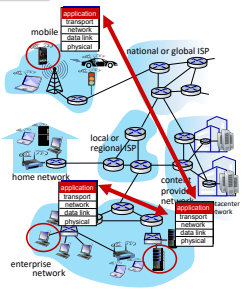
Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

DIETI UNINA

Application Layer
Network Applications

- A **network application** is composed by multiple programs that run on different end-systems and communicate with each other over the network.
- *Example*: a Web application includes two distinct programs:
  - the **browser program** running in the user's host (desktop, laptop, tablet, smartphone, and so on).
  - the **Web server program** running in the Web server host.

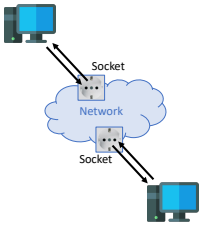


Application Layer
Some Network Applications

- The **applications** are the programs that runs on the **devices** and allow users to access to **services**.
  - social networking
  - Web
  - text messaging
  - e-mail
  - multi-user network games
  - streaming stored video (YouTube, Netflix)
  - P2P file sharing
  - voice over IP (e.g., Skype)
  - real-time video conferencing
  - Internet search
  - remote login
  - …

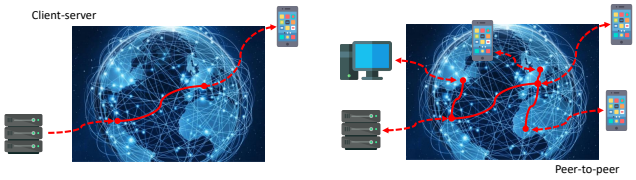
Application Layer
Creating Network Applications

- Creating network applications means to write programs that:
  - **Run on different end systems**, perhaps using different languages or OS.
  - **Communicate over network** (e.g., via sockets) by means of a specific protocol.
- No need to write software for network-core devices:
  - Network devices **do not run user applications**.
  - From applications' perspective, network is ideally **back-box**.
  - There are **specific libraries** (e.g., sockets) implementing network functionalities.



Application Layer
Network Applications

- A **(network) application architecture** is designed by the *application developer* and dictates how the application is structured over the various end systems:
  - **Client-server**: the nodes are heterogeneous, there is an always-on host (server), which provides services to be requests from many other hosts (clients).
  - **Peer-to-peer (P2P)**: the nodes are (ideally) homogeneous, the application exploits direct communication between pairs of intermittently connected hosts (peers).

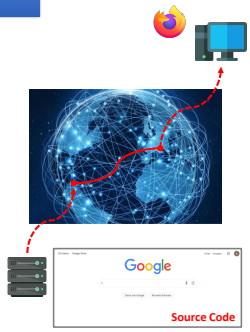


Application Layer
Client-server Applications

- In a client-server architecture, **clients do not directly communicate** with each other.
- The server has a **fixed, well-known address** (IP address) so a client can always contact the server by sending a packet (request) to it.
- In a client-server architecture **multiple servers** are often involved to keep up with all the requests from clients. The **client is typically unaware** of that and perceives them as a single server.
- Multiple servers can be:
  - Grouped into **data centers** containing a huge number of servers in a specific location.
    - Servers must be powered, maintained, and well connected.
  - Scattered as **distributed servers** all around the world.
    - Servers must be interconnected and coordinated.
  - Organized in **distributed data centers** (e.g., Google).

Application Layer
Client-server Applications

- A **web application** is a typical client-server example:
  - There is an **always-on Web server** receiving requests from the browsers on the client hosts.
  - When a Web server receives a **request** for an object from a client host, it **responds** by sending the requested object to the client host.
  - The web **server is always reachable** by the hosts.
  - Google has 30 to 50 **data centers distributed around the world**, which collectively handle search, but also YouTube, Gmail, and other services (in Italy, there is one in Milan).

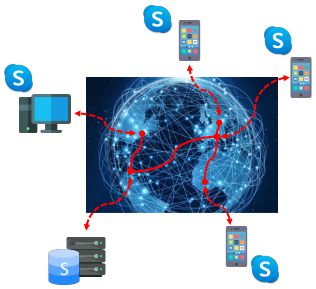


Application Layer
Peer-to-peer Applications

- It is often used for **traffic-intensive applications**.
- There is no single server with fixed address, but **all clients have their own addresses**.
- Pure P2P applications are uncommon: most applications have **hybrid architectures**, combining both client-server and P2P elements.
  - For example, for many instant messaging applications, **servers are used to track the addresses of users**, but user-to-user messages are sent directly between user hosts (without passing through intermediate servers).
- P2P architectures are **scalable** and **distributed**.
  - For example, in a P2P file-sharing application, although each peer generates workload by requesting files, each peer also **adds service capacity** to the system by distributing files to other peers.
- P2P architectures are also **cheap** (no need for infrastructures or significant bandwidth) but there are **security, performance, and reliability issues**.

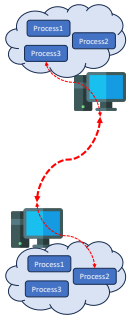
Application Layer
Peer-to-peer Applications

- Typical P2P applications are internet **telephony and video conference** (e.g., Skype) or **file-sharing** (e.g., BitTorent, Napster):
  - Servers are used to **track the IP addresses of users**, but user-to-user messages/requests are sent directly between user hosts (without passing through intermediate servers).
  - In file-sharing applications the server can also **trace all available files** in order to speed up the search.



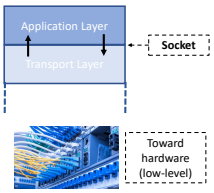
Application Layer
Communication between Processes

- In a network application there are **processes** running on different machines (potentially running on **different operating systems**) that communicate through the network.
  - In *web applications* the process of a client's browser exchanges messages with the process of the web server.
  - In *P2P file-sharing* a file is transferred from a process in one peer to a process in another peer.
- Between a pair of communicating processes there is typically a **client process** and a **server process**:
  - In *web applications* a browser's process is a client, while the server's process is a server.
  - In *P2P file-sharing* the peer that is downloading can be seen as a client, while the peer that is uploading as a server.
- In a P2P application processes may "change role".



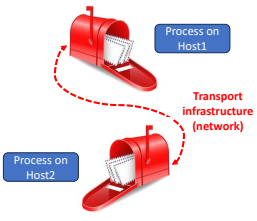
Application Layer
Communication between Processes

- Most network applications consist of **pairs of communicating processes** sending/receiving messages to/from each other through the underlying network.
- A **socket** is a **software interface** that allows processes to send/receive messages over the network.
- Considering the TCP/IP stack, a **socket is the interface** between the application layer and the transport layer (TCP).



Application Layer
Communication between Processes

- A typical analogy is to consider sockets as **mailboxes**:
  - When a process wants to send a message to another process on another host, it puts the message **into a postal mailbox**.
  - This sending process **assumes** that there is a **transportation infrastructure** on the other side of its door that will transport the message to the mailbox of the destination process.
  - Once the message arrives at the destination host, the message passes through the **receiving process's mailbox** (socket).

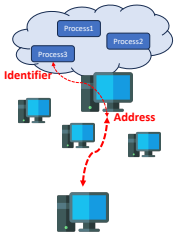


Application Layer
Communication between Processes

- **Sockets** are used as **Application Programming Interface** (API) between the application and the network.
- The application developer has control of everything on the application-layer side of the socket but **has little control of the transport-layer** side of the socket.
- The only control that the application developer has on the transport-layer side is:
  1. The **choice** of transport protocol (TCP/UDP).
  2. Perhaps the ability to set a few **transport-layer parameters** such as maximum buffer and maximum segment sizes.
- Once the application developer chooses a transport protocol (if a choice is available), the application is built assuming as given the transport-layer services provided by that protocol.

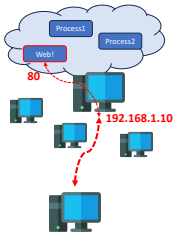
Application Layer
Communication between Processes

- Following the postal mail analogy, **processes need an address** to send messages.
- Since **multiple network applications** can be running on a **single host**, to identify the receiving process, two pieces of information need to be specified:
  1. An **address** of the host (to find the right host on the network).
  2. An **identifier** of the receiving process (to find the right process in the host).
- The **address** works at the network level (routing) while **identifier** works at transport level.



Application Layer
Communication between Processes

- In networks, a host is identified by an **IP address** (32-bit quantity) while the process is identified by a **port number**.
- Popular applications have been conventionally assigned to **specific port numbers**.
  - For example, a Web server is identified by port number 80, a mail server process (using the SMTP protocol) is identified by port number 25.
- A list of well-known port numbers (and protocols) can be found here: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.



Application Layer
Services from transport-layer

- The **transport-layer** (below the application layer) offers protocols to guarantee:
  - **Reliability** of data transfer: data sent by one end of the application is delivered correctly and completely to the other end.
  - **Throughput**: the rate at which the sending process can deliver bits to the receiving process (bit/sec).
  - **Timing**: every bit that the sender pumps into the socket arrives at the receiver's socket within a time range.
  - **Security**: encryption and decryption of the messages.
- TCP: includes a **handshake** between processes service and a reliable data transfer service (aka **connection-oriented**).
- UDP: is **lightweight** with minimal services, there is no handshake, no guarantee that messages are received (aka **connectionless**).

Application Layer
Protocols

- An **application-layer protocol** defines how network application's processes, running on different end systems, pass messages to each other.
  - how are messages structured? What are the meanings of the various fields in the messages? When do the processes send the messages?
- Specifically, application-layer protocols define:
  1. The **types** of messages exchanged (for example request messages and response messages).
  2. The **syntax** of the various message types, such as the fields in the message and how the fields are delineated.
  3. The **semantics** of the fields, i.e., the meaning of the information in the fields.
  4. The **rules** for determining when and how a process sends messages and responds to messages.

Application Layer
Some Application Protocols

- There are several application-layer protocols that are commonly used on networks (and internet), here some example.

Application	Description
DHCP	Dynamic Host Configuration Protocol, assigns IP addresses
DNS	Domain Name System, translate website names to IP addresses
HTTP/HTTPS	HyperText Transfer Protocol (Secure), transfer web pages
SMTP/SMTPS	Simple Mail Transfer Protocol (Secure), sends email messages
SNMP	Simple Network Management Protocol, manages network devices
Telnet/SSH	Teletype Network (Secure SHell), allows command-line interfacing with remote hosts
FTP/FTPS	File Transfer Protocol (Secure), used to transfer files

Application Layer
Some Application Protocols

- Some application may exchange **sensitive information** (e.g., user's credentials, personal data, etc.).
- On public networks, messages should be **secured**.

Application	Description
DHCP	Dynamic Host Configuration Protocol, assigns IP addresses
DNS	Domain Name System, translate website names to IP addresses
HTTP/HTTPS	HyperText Transfer Protocol (Secure), transfer web pages
SMTP/SMTPS	Simple Mail Transfer Protocol (Secure), sends email messages
SNMP	Simple Network Management Protocol, manages network devices
Telnet/SSH	Teletype Network (Secure SHell), allows command-line interfacing with remote hosts
FTP/FTPS	File Transfer Protocol (Secure), used to transfer files

Application Layer
FTP and FTPS

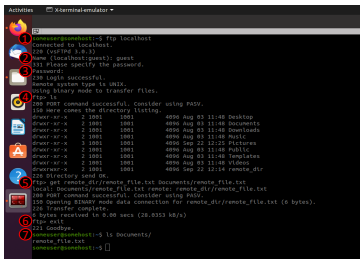
- FTP (File Transfer Protocol) is one of the **oldest** protocols defined in Internet (first version in 1971) and is used to transfer files between hosts over network.
- In the basic version of FTP data transfer is in **clear-text** (username, password, and files) so it is best used in local or private applications.
- The **secure** version FTPS (FTP Secure) protects username and password and encrypt contents.
- Both FTP and FTPS have two components:
  - The **protocol** that specifies commands (show, get, delete files, etc.).
  - A **software application** implementing the protocol (client-side and a server-side software).

Application Layer
FTP Example

- In Linux we can use ftp and vsftpd (very secure FTP daemon) as client and server applications respectively.
- On the server machine:
  - Install vsftpd:
    - $ sudo apt-get install vsftpd
  - Check ssh server running
    - $ service vsftpd status (check if daemon is running)
- On the client machine:
  - Install ftp:
    - $ sudo apt-get install ftp (it is already available in Ubuntu)
  - Connect to server:
    - $ ftp ADDRESS (usr and pass will be asked)
  - Close connection:
    - $ exit

Note: The server side of this application is implemented as a **daemon**: a program that runs in background and waits for clients to connects. This is a common approach.

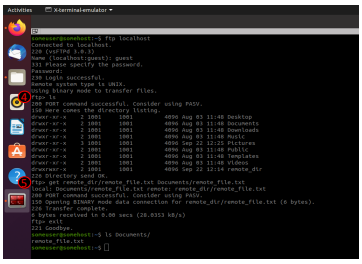
Application Layer
FTP Example (get a file)



Here we connect to ourselves (localhost), but an IP address can be specified:

1. Run ftp with the address of the remote host.
2. Insert username with which you want to log on remote host.
3. Insert password
4. Run ls to see files and directories
5. Copy the file remote_dir/remote_file.txt into (local) Documents
6. Exit from ftp
7. Check the file

Application Layer
FTP Common Commands

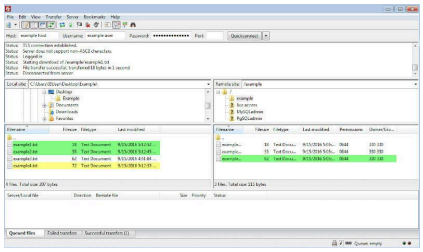


Common FTP commands:

- help - List all available FTP commands.
- cd - Change directory on remote machine.
- lcd - Change directory on local machine.
- ls - View the names of the files and directories in the current remote directory.
- mkdir - Create a new directory within the remote directory.
- pwd - Print the current working directory on the remote machine.
- delete - Delete a file in the current remote directory.
- rmdir- Remove a directory in the current remote directory.
- get - Copies a file from the remote server to the local machine.
- put - Copies a file from the local machine to the remote machine.

Application Layer
FTP Example

- Ubuntu has also GUI-based clients like **nautilus** (standard directory explorer) or **filezilla**.



Screenshot of filezilla's GUI: filezilla is a cross-platform FTP client (Linux, Windows, Mac OS).

Application Layer
Telnet and SSH

- Telnet (short for teletype network) is a client/server application protocol that provides access to **virtual terminals** of remote systems on local area networks or the Internet.
- The Secure Shell Protocol (SSH) is a replacement for the unsecured Telnet (SSH is a cryptographic protocol) for operating network services over an unsecured network. It is still based on a client-server architecture.
- Both Telnet and SSH have two components:
  - The **protocol** that specifies how messages are structured and how hosts communicate within a session.
  - A **software application** implementing the protocol (client-side and a server-side software).

Application Layer
SSH Example

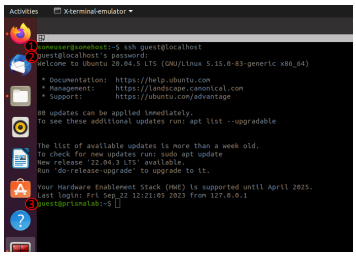
- In Linux we can use openssh as a client/server application that allows SSH between two hosts.

- On the server machine:
  - Install openssh:
    - $ sudo apt-get install openssh-server
  - Check ssh server running
    - $ ssh localhost (try connection with yourself)
      or
      $ sudo service ssh status (check if daemon is running)

- On the client machine:
  - Install openssh:
    - $ sudo apt-get install openssh-client
  - Connect to server:
    - $ ssh USERNAME@ADDRESS (pass will be asked)
  - Close connection:
    - $ exit

Note: Once you are connected to the remote users, you may use all commands the shell can offer.

Application Layer
SSH Example



Here we connect to ourselves (localhost), but an IP address can be specified:

1. Run ssh with username@address of the remote host.
2. Insert password
3. The terminal will show username@hostname of the remote shell. Now you may enter commands or close the connection (exit command).

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Application Layer
Web and HTTP

- Until the early 1990s the Internet was used **primarily by researchers**, **academics**, and university **students** to log in to remote hosts, to transfer files from local hosts to remote hosts and vice versa, to receive and send news, and to receive and send electronic mail.

- Although these applications were (and continue to be) extremely useful, the Internet was essentially unknown outside of the academic and research communities.

- In the early 1990s the **World Wide Web** [Berners-Lee 1994] was the **first Internet application** that caught the general public.

Application Layer
Web and HTTP

- The **World Wide Web** (WWW or simply Web) is a **collection of information** such as documents, images, video, audio, etc. that can be accessed over the Internet according to a specific protocol called **HyperText Transfer Protocol** (HTTP).

- HTTP-based communication is typically client-server:
  - A **client program** that translates users' requests into HTTP messages. This is implemented into the **web browsers** (Chrome, Firefox, Edge, Safari, etc.).
  - A **server program** that executes the **HTTP request** and returns a **HTTP response**.

- HTTP mainly defines the **structure of messages** and how the client and server should exchange such messages.

- The Web and its protocols **serve as a platform** for web-based applications such as YouTube, Web-based e-mail (e.g., Gmail), and most mobile Internet applications, including Social Networks.

Application Layer
URL

- The information on the web are called **resources** (or objects) and are identified by a **Uniform Resource Locator** (URL) which is a string composed as follows:

  [protocol]://[usrinfo@][host][:port][/path][?query][#fragment]

- Where:
  - [protocol] is the **protocol** to use while accessing the resource (HTTP, HTTPS, FTP, etc.).
  - [usrinfo] (optional) are **user's information** such as username and password followed by a @ (e.g., username:password@). This is **mostly deprecated** due to security issues.
  - [host] is the **name or the IP** address of the server.
  - [port] (optional) is the **port** to use (often inferred from the protocol).
  - [path] is the **path of the resource** within the server (e.g., /path/to/resource).
  - [query] is preceded by the ? and specifies **possible requests**.
  - [fragment] preceded by the # **identifies an element** in the resource (e.g., a form).

Application Layer
URL

- Most Web pages consist of a **base HTML** (HyperText Markup Language) file and several references to additional **objects** (images, videos, Java applets, etc.).
  - Example: a Web page containing a HTML text and five JPEG images **has six objects**: the base HTML file plus the five images.
- An example of URL is:

  http://www.someSchool.edu/someDepartment/picture1.gif
- Where:
  - *http* is the **protocol**.
  - *www.someSchool.edu* is the **hostname**.
  - */someDepartment/picture1.gif* is the **path** to the object.

Application Layer
URL

- Real example of URL query:

  https://en.wikipedia.org/w/index.php?title=SSC_Napoli
- Where:
  - *https* is the **protocol**.
  - *en.wikipedia.org* is the **hostname**.
  - */w/index.php* is the **path** to the page.
  - *?title=SSC_Napoli* is the **query**.

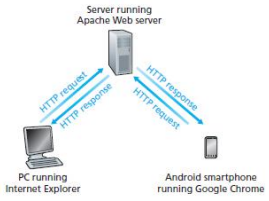
en.wikipedia.org/wiki/SSC_Napoli

- In this case, it is the same as asking for the page:

  https://en.wikipedia.org/wiki/SSC_Napoli

Application Layer
HTTP

- HTTP defines how Web clients request objects (e.g., Web pages) from Web servers and how servers transfer them to clients.
- When a user requests an object (for example, a Web page by clicking on a hyperlink), the **browser** sends HTTP request messages for the **objects in the page** to the server.
- The server receives the requests and responds with HTTP response messages that contain the **objects**.



Application Layer
HTTP

- HTTP uses **TCP as its underlying transport protocol** (rather than UDP).
- The HTTP client **first initiates a TCP connection** with the server. Once the connection is established, the browser and the server processes access TCP through their **socket interfaces**.
  - On the client side the socket interface is the door between the client process and the TCP connection.
  - On the server side it is the door between the server process and the TCP connection.
- The client sends HTTP request messages into its socket interface and receives HTTP response messages from its socket interface. Similarly, the HTTP server receives request messages from its socket interface and sends response messages into its socket interface.

Application Layer
HTTP

- One of the reasons why HTTP uses TCP as a transport protocol is that **several useful services are provided by TCP**, most of all **reliable data transfer**.
- Recall reliability: TCP guarantees HTTP request/response messages to eventually arrive intact at the destination (if possible).
- Here we see one of the great advantages of a layered architecture: HTTP-based applications need not worry about reachability, data loss, etc.
- HTTP applications can be **simpler** (both logically and computationally), by **delegating** most of the work to the lower-level stack.

Application Layer
HTTP: Statelessness

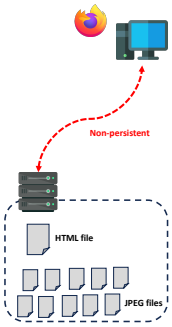
- Simplicity is very important for HTTP-based servers dealing with **large number of requests per second** (servers run *around 1000 HTTP requests per second*, while whole google search infrastructure receives *around 100000 queries per second*).
- HTTP applications are typically **stateless**: the server does not maintain any information about the interaction (sequence of requests/responses) with a specific client.
  - Example: if a client asks for the **same object twice** in a row, the server does not consider this request redundant, but **it just resends the object**, as it has completely forgotten the past.
- Not all applications are stateless, several applications are instead **stateful**.

Application Layer
HTTP: Persistent and Non-persistent Connections

- In many applications (such as HTTP-based), the client and server may communicate for an **extended period of time**, by sending **multiple pairs of request-response**.
  - This flow of messages may be made back-to-back, periodically (at regular intervals), or intermittently.
- When relying on a TCP connection, we can have:
  - **Persistent connections**: all requests and their corresponding responses are sent over the same TCP connection.
  - **Non-persistent connections**: for each request-response pair a new TCP connection is established.
- By default, todays HTTP applications use **persistent connections**, while HTTP clients and servers can be configured to use non-persistent connections if needed. Early versions of HTTP (HTTP 1.0) used non-persistent connections by default.

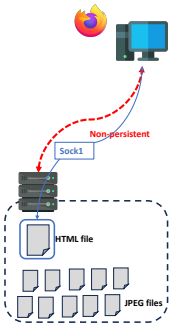
Application Layer
HTTP: Non-persistent Connection (Example)

- In a non-persistent connection TCP connection is closed every time **a request is served** (e.g., an object is sent).
- Example: assume to ask for a Web page consisting of a base HTML file and 10 JPEG images (all 11 of these objects reside on the same server).
- The URL for the base HTML file is:
  - http://www.someSchool.edu/someDepartment/home.index



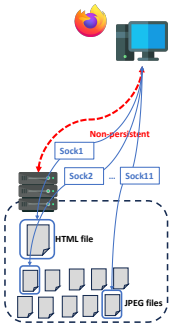
Application Layer
HTTP: Non-persistent Connection (Example)

- The connection takes place as follows:
  1. The HTTP **client process initiates a TCP connection** to the server www.someSchool.edu on port number 80 (default HTTP port), we will have two sockets: one at the client and one at the server.
  2. The HTTP **client sends an HTTP request message** to the server via its socket including the path name /someDepartment/home.index.
  3. The HTTP **server process receives the request message** via its socket, **retrieves** the object /someDepartment/home.index (from RAM or disk), **encapsulates** the object in an HTTP response message, and **sends back** the response message to the client via its socket.
  4. The HTTP **server process tells TCP to close** the TCP connection, the TCP will (later) terminate the connection once it is sure that the client has received the response message intact (reliability).
  5. The HTTP **client receives the response message**. The TCP **connection terminates**. The message indicates that the encapsulated object is an HTML file. The client extracts the file from the response message, examines the HTML file, and **finds references to the 10 JPEG objects**.
  6. The first four **steps are then repeated** for each of the referenced JPEG objects.



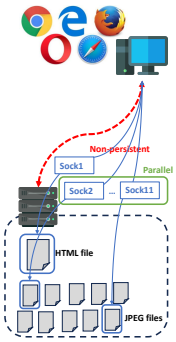
Application Layer
HTTP: Non-persistent Connection (Example)

- Since the connection does not persist over different objects, we need **11 TCP connection** (and 11 pairs of sockets) to transfer the whole page, hence the elements of a page may arrive in **different moments**.
- The way web pages are displayed **depends on the browser**: two different browsers may interpret (hence, display to the user) a Web page in somewhat different ways.
- HTTP **defines only the communication protocol** between the client HTTP program and the server HTTP program, not how contents are displayed.



Application Layer
HTTP: Non-persistent Connection (Example)

- The different **connections** may be established in **sequence** or in **parallel**.
  - In this example, once we know that 10 JPEG images are needed (i.e., the main page is received), we may download all of them in parallel.
- In modern browsers users may also **configure** the degree of parallelism involved:
  - Most browsers open 5 to 10 parallel TCP connections, and each of these connections handles one request-response transaction.
  - Clearly, the use of parallel connections **shortens the response time but increases the computational cost**.
- Despite parallelism, establishing multiple (TCP) connections may still induce a significant **time overhead.**

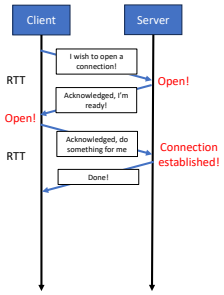


Application Layer
HTTP: Non-persistent Connections (RTTs)

- It is **difficult to precisely estimate times** on Internet. We can estimate the overhead it takes to finalize a HTML request in terms of **round-trip times** (RTTs).
- The **RTT** is the time it takes for a small packet to travel from client to server and back to the client.
- When a user clicks on a hyperlink the browser initiates a TCP connection between with the Web server.
- TCP connections are guaranteed (connection-oriented) to do that, a **three-way handshake** is performed…

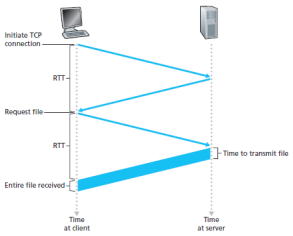
Application Layer
HTTP: Non-persistent Connections (RTTs)

- TCP **three-way handshake** procedure is performed through the following 3 steps:
  - The client sends a small TCP segment to the server, to signal that a connection is requested.
  - The server acknowledges and responds with a small TCP segment.
  - The client acknowledges back to the server.
- From RTT's perspective, it takes 2 RTTs to establish a TCP connection (assuming no packet-loss).
- Notice that a similar handshake also takes place when **connection is closed**.



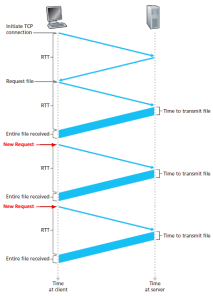
Application Layer
HTTP: Non-persistent Connections (RTTs)

- The first two parts of the three-way handshake take **one RTT**.
- In HTTP, the client sends the HTTP **request message** combined with the third part of the three-way handshake (the acknowledgment).
- Once the request message arrives, the server sends the HTML file. This HTTP request/response eats up **another RTT**.
- Roughly, the total response time **is two RTTs plus the transmission time** at the server of the HTML file.



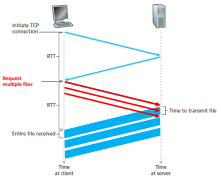
Application Layer
HTTP: Persistent Connections

- In persistent connections, the server **leaves open** the TCP connection after sending a response.
- Requests and responses between the same client and server can be sent over the same connection: **an entire Web page** (the HTML file and the 10 images from the previous example) can be sent over **a single persistent TCP connection**.
- **Multiple Web pages** residing on the same server can also be sent from the server to the same client over a single persistent TCP connection.
- Following this approach, we can save roughly **2-4 RRTs per object**.

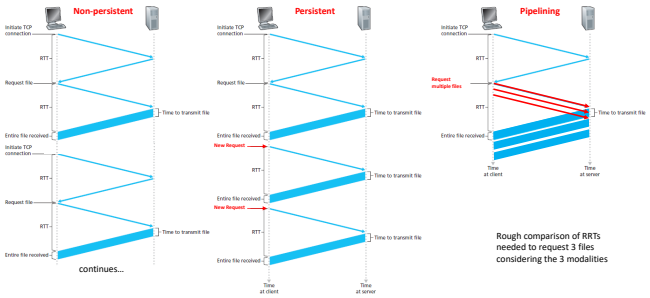


Application Layer
HTTP: Persistent Connections with Pipelining

- **Pipelining**: requests for objects can be made back-to-back, without waiting for replies to pending requests.
- In the last years (from HTTP/2) multiple requests and replies can be interleaved in the same connection.
- There is also a mechanism to prioritize HTTP requests and replies within this connection.
- Persistent connections with **pipelining is currently the default** mode in HTTP.



Application Layer
HTTP: Persistent vs. Non-persistent Connections



Rough comparison of RRTs needed to request 3 files considering the 3 modalities

Application Layer
HTTP: Persistent vs. Non-persistent Connections

- Persistent:
  - Faster, especially using pipelining.
  - Less resources needed (CPU and memory).
  - More complex to implement, especially using pipelining.
  - Connection may be left open even if unused. Typically, the HTTP server closes a connection when it isn't used for a certain time (timeout).
- Non-persistent:
  - Easy to implement and nicely fitting the statelessness of HTTP.
  - Needs more resources, for each of these connections, TCP buffers must be allocated and TCP variables must be kept in both the client and server.
  - Slower, 1-2 additional RRTs needed per request.
  - Connections cannot be left open.

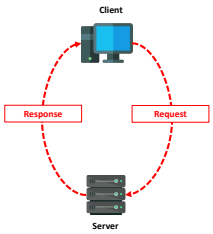
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

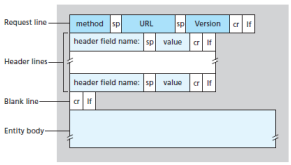
Application Layer
HTTP: Message Format

- In HTTP protocol we have 2 different formats for the request and the response messages.
- **Request**: specifies the command (method) that the HTTP server has to perform (e.g., give me a web page, fill a specific form, etc.).
- **Response**: reports the outcome of the command (success, fail, etc.) and possible data (e.g., the requested file).
- Both messages are written in ordinary ASCII text, so they are easily readable by humans.



Application Layer
HTTP: Request Message Format

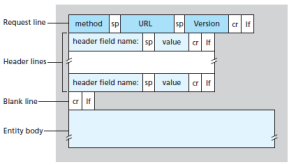
- The request message format include the following fields:
  - The **method**: specifies the requested command to be executed by the server.
  - The **URL**: is used to identify the object on which we want to operate.
  - The **version**: specifies the HTTP version (e.g., HTTP/1.1).
  - The **header lines**: contain the parameters of the request, the number and the type of these lines are not fixed. Each line include the **name** and the **value** of the parameter.
    - For instance, we can here specify if we want **persistent** or **non-persistent** connection.
    - Custom headers can also be used.
  - The **body**: is method-specific and contains data that are potentially associated with the command (e.g., text used to fill a form).



The fields are separated by special characters:
- The sp is space character.
- The cr is carriage return (\r).
- The lf is line feed (\n).

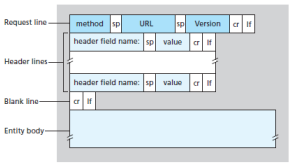
Application Layer
HTTP: Request Message Format (Methods)

- The **GET** method is used to retrieve objects (resources) form the server.
  - This is one of the most used commands as every time we browse a new webpage the associated file must be retrieved.
  - In the GET method the **body is empty**.
- The **POST** method is used to set info inside objects (resources) of the server.
  - A request generated with a form does not necessarily use the POST method, HTML forms often use the GET method and include the inputted data (in the form fields) in the requested URL.
  - The body contains the info to be posted.



Application Layer
HTTP: Request Message Format (Methods)

- The **HEAD** method is similar to the GET method, when a server receives a request with the HEAD method, it responds with an HTTP message, but the object is not returned.
  - Application developers often use the HEAD method for debugging.
- The **PUT** method is often used in conjunction with Web publishing tools.
  - It allows a user to upload an object to a specific path (directory) on a specific Web server.
  - The PUT method is also used by applications that need to upload objects to Web servers.
- The **DELETE** method allows a user, or an application, to delete an object on a Web server.



Application Layer
HTTP: Request Message Format (Example)

- In this example we have a **browser** (Mozilla/5.0), implementing "HTTP/1.1", that is **requesting** the object "/somedir/page.html" to the "www.someschool.edu" web server.

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

- We have **five lines**:
  - The **GET request line** (resource and version).
  - The **four header lines** (host, connection, user-agent, accept-language)…

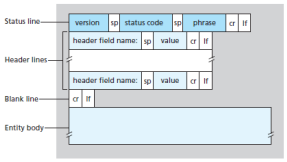
Application Layer
HTTP: Request Message Format (Example)

- The line **"Host: www.someschool.edu"** specifies the **host on which the object resides**.
  - **Destination is not always the next host**: a message could be **forwarded by another host** (e.g., proxy server), so the address in the TCP is different from the real target host.
- The line **"Connection: close"** specifies that the browser is asking for a non-persistent connection (i.e., close after finish).
- The line **"User-agent: Mozilla/5.0"** specifies the **browser's type**.
  - This is sometimes useful because the **server can send different versions** of the same object (each version is addressed by the same URL) **depending on the type**.
- The line **"Accept-language: fr"** indicates that the **user prefers to receive a French version** of the object (if exists).

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

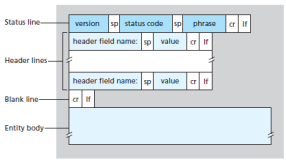
Application Layer
HTTP: Response Message Format

- The general format of the response message is similar to the request one.
- In this case, instead of a request line, we have a **status line** that reports the outcome of the command that includes:
  - The **version**: reports the HTTP version of the server's response.
  - The **status code**: a code (number) that specifies the outcome of the command.
  - The **phrase**: contains the result of the request.



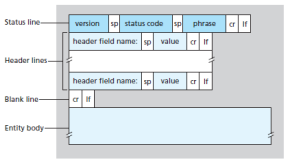
Application Layer
HTTP: Response Message Format (Status Codes)

- **Status codes** are divided into classes:
  - 100-199 **informational**: info regarding the request.
  - 200-299 **successful**: the request has been executed successfully.
  - 300-399 **redirection**: there are additional actions needed from the client.
  - 400-499 **client-error**: the request cannot be executed because of a client issue.
  - 500-599 **server-error**: the request cannot be executed because of a server issue.



Application Layer
HTTP: Response Message Format (Status Codes)

- Some typical examples are:
  - **200 OK**: request succeeded, and the information is returned in the response.
  - **301 Moved Permanently**: requested object has been permanently moved; the new URL is specified in the "Location" header of the response message.
  - **400 Bad Request**: generic error code indicating that the request could not be understood by the server.
  - **404 Not Found**: The requested document does not exist on this server.
  - **505 HTTP Version Not Supported**: The requested HTTP protocol version is not supported by the server.



Application Layer
HTTP: Response Message Format (Example)

- In this example, the status line indicates that **the server is using HTTP/1.1 and that everything is OK** (the server has found and is sending the requested object).
- There are **five header lines** (connection, date, server, last-modified, content-length, content-type).
- The **body contains the requested object** (represented by data data data data data ...).

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

Application Layer
HTTP: Response Message Format (Example)

- The line **"Connection: close"** informs the client that the **connection will be closed** after this message (non-persistent).
- The line **"Date: …"** indicates the time and **date when the HTTP response was created** and sent by the server.
- The line **"Server: …"** indicates that the message was **generated by an Apache Web server** (similar to user-agent).
- The line **"Last-Modified: …"** indicates the time and date when the **object was created or last modified**.
  - Useful in case of caching, as cached files can be outdated.
- The line **"Content-Length: …"** indicates the **number of bytes in the object**.
- The line **"Content-Type: …"** indicates that the **object is HTML text** (here, the object type is officially indicated by this header line and not by the file extension).

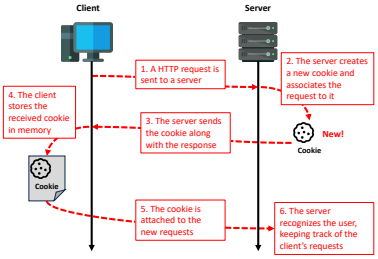
```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```

Application Layer
HTTP: Cookies

- An HTTP **server is typically stateless**, this simplifies server design reduces the use of resources and allow servers to handle thousands of simultaneous TCP connections.
- Pure statelessness is also a strong limitation as **several web functions are client-specific** (for example, Amazon's cart depends on the client, Netflix suggests contents based on client's preferences, etc.).
- For these purposes, HTTP uses cookies. A **cookie** is a digital token (alphanumeric ID) used by servers to identify a specific client.
  - Cookies allow web sites to keep track of users, most of the major commercial Web sites use cookies.
  - Cookies may also have attributes (e.g., expiration date).

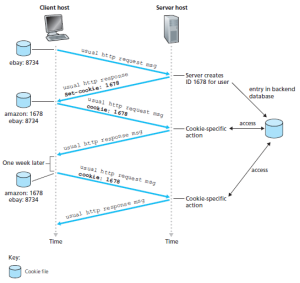
Application Layer
HTTP: Cookies

- A **cookie is created by the server** and delivered to the client.
- Cookie technology involves four main components:
  1. A **"Set-cookie" header** line in the HTTP response message.
  2. A **"Cookie" header** line in the HTTP request message.
  3. A **file** on the client system (managed by the user's browser).
  4. A **back-end database** on the server.



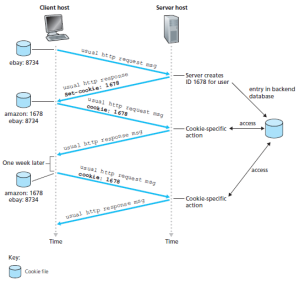
Application Layer
HTTP: Cookies (Example)

- Let's assume a client host, which uses eBay regularly, to **contact Amazon.com for the first time**.
- The client already has a cookie for eBay, but **it has no cookie for Amazon**.
- When the request comes into the Amazon Web server, **the server creates a cookie** (ID number) and an associated entry in its back-end database.
- The Amazon Web server then responds to client's browser and **includes in the HTTP response a "Set-cookie" header** line, which contains the ID (`Set-cookie: 1678`).



Application Layer
HTTP: Cookies (Example)

- When the response is received, **the browser appends a line to a special cookie file** including:
  - the hostname of the server.
  - the ID number of the cookie.
- From now on, the requests from the client to Amazon **will be associated to the new cookie** by including the header line `Cookie: 1678` to the HTTP messages.
- Amazon server is then able to **track the client's activity through the database**. It knows exactly which pages user 1678 visited, in which order, and at what times.
- If the client returns to Amazon's site **one week later**, the browser will continue to put the header line Cookie: 1678 in the request messages.

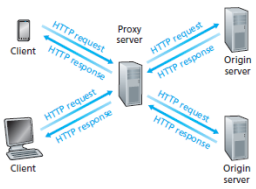


Application Layer
HTTP: Cookies

- Amazon (and other web sites) uses cookies to provide different services:
  - **Shopping cart service**, the server can maintain a list of intended purchases during browsing.
  - **Products recommendations**, based on the visited web pages.
  - **User's registration**, by associating user's info to the cookie in the database (credit-card, name, e-mail, address) so you don't have to re-insert them every time.
- Although cookies often simplify the Internet shopping experience for the user, they are **controversial** because they can also be considered as an invasion of privacy.
- Using a combination of **cookies** and user-supplied **account information**, a Web site can learn a lot about a user and potentially sell this information to a third party.

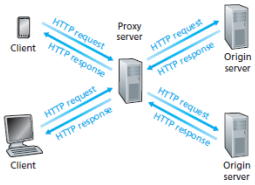
Application Layer
HTTP: Web Caching

- A **Web cache** (also called a **proxy** server) is a network entity that satisfies HTTP requests on the behalf of an origin Web server.
- The Web cache has its own disk storage and **keeps copies of recently requested objects** in this storage.
- A browser can be configured so that all the HTTP **requests are first directed to the Web cache** to check if a copy of the requested object is available.



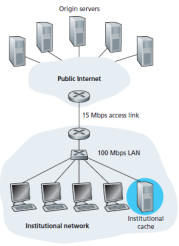
Application Layer
HTTP: Web Caching

- Let's assume a browser to request the object http://www.someschool.edu/campus.gif passing through a web cache:
  1. The **browser establishes a TCP connection** to the Web cache and sends an HTTP request for the object to the Web cache.
  2. The Web cache **checks if it has a copy of the object** stored locally. If so, the Web cache returns the object within an HTTP response message to the client browser.
  3. If the **Web cache does not have the object**, the Web cache opens a TCP connection to the origin server (www.someschool.edu) and sends an HTTP request for the object.
  4. When the **Web cache receives the object, it stores a copy in its local storage** and sends a copy, within an HTTP response message, to the client browser.



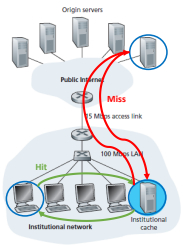
Application Layer
HTTP: Web Caching

- Note that a cache is both a server (when providing objects) and a client (when requesting objects) at the same time.
- Web caching has seen deployment in Internet for two reasons:
  1. A Web cache can substantially **reduce the response time for a client request**, particularly if a high-speed connection stands between the client and the cache.
  2. Web caches can substantially **reduce traffic** of a company or institution toward Internet, in so reducing costs due to bandwidth.



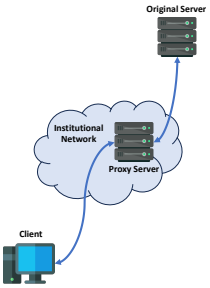
Application Layer
HTTP: Web Caching

- Web caches are **often installed into a company/institution local network** to speed-up and reduce traffic.
- A **hit** happens when a cache successfully provide an object without contacting the original server.
- The hit rate, i.e., the fraction of requests that are satisfied by a cache, typically **ranges from 0.2 to 0.7**.
  - Hit rate increases when more clients use the cache.
- This means that up to 70% of requests can be served locally.



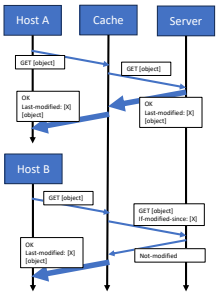
Application Layer
HTTP: Web Caching

- The UNINA network also provides an institutional proxy (proxy.unina.it).
- Besides performance, **proxy servers can also be used to access to institutional services**.
- As requests are forwarded by the institutional proxy, from the original server's standpoint all **requests are coming from the institution**.
- There are also several commercial/free **proxy servers available on Internet**.



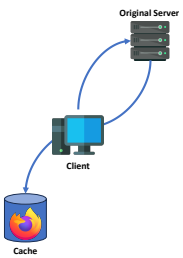
Application Layer
HTTP: Web Caching

- Web caching introduces a new problem: the copy of an object residing in the **cache may be outdated**.
- To avoid this issue, HTTP has a mechanism that allows a cache to **verify the stored object**.
- The **conditional GET** is an HTTP request message including a GET method and "If-Modified-Since:" header.
- The cache **checks if the object is up to date** and, if so, the stored version is sent back to the host (no further communication needed).



Application Layer
HTTP: Web Caching

- Caching **is also performed locally** by browsers.
- The principle is the same as servers, the browser stores objects locally so they no need to be retrieved from the server.
- This is a common techniques in modern browsers as **it drastically improves performance**.
- The local version of **the object may be not updated**, generating errors (quite frequent).



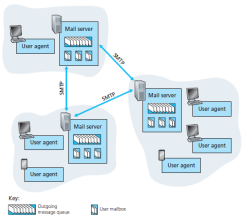
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

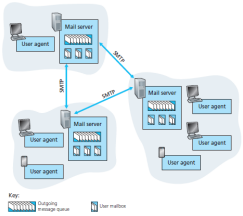
Application Layer
Mails and SMTP

- Electronic mailing (e-mail) is one of the oldest and most important internet applications.
- The **Simple Mail Transfer Protocol** (SMTP): is the main application-layer protocol for Internet electronic mail.
- There are two elements involved:
  - **User agent**: application allowing mail management (e.g., Outlook, Apple Mail, Gmail, etc.).
  - **Mail Server**: a server that stores mails and maintain user-specific mailboxes.



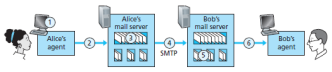
Application Layer
Mails and SMTP

- SMTP mainly **works on mail servers** to allow them to exchange mails.
- Once a new mail arrives on the server, it is stored inside the **user-specific mailboxes**, waiting to be download locally by the user agent.
- Users **do not exchange mails directly** (as in instant messaging). It is preferred to use more reliable and specialized mail servers.
- In SMTP there are a **client-side** (sender) and a **server-side** (receiver) that run on mail servers, both using the reliable **TCP** to transfer mail.



Application Layer
SMTP

- Let's assume to have host A (Alice) that is sending an e-mail to host B (Bob), we have 4 elements involved:
  - Alice's agent.
  - Alice's mail server.
  - Bob's agent.
  - Bob's mail server.



- The process works as follows:
  1. Alice **invokes her user agent** for e-mail, provides Bob's e-mail address (e.g., bob@someschool.edu), composes a message, and instructs the user agent to send the message.
  2. Alice's **user agent sends the message to her mail server**, where it is placed in a message queue.
  3. The client side of SMTP, running on Alice's mail server, sees the message in the message queue. It **opens a TCP connection to an SMTP server**, running on Bob's mail server.
  4. After some initial SMTP handshaking, the SMTP client sends Alice's **message into the TCP** connection.
  5. At Bob's mail server, the **server side of SMTP receives the message**. Bob's mail server then places the message in **Bob's mailbox**.
  6. Bob **invokes his user agent** to read the message at his convenience.

Application Layer
SMTP

- First, the client SMTP (running on the sending mail server host) has **TCP establish a connection to port 25 at the server SMTP** (running on the receiving mail server host).
- If the server is down, the client tries again later. Once this connection is established, **the server and client perform some application-layer handshaking**: SMTP clients and servers introduce themselves before transferring information.
  - During this SMTP handshaking phase, the SMTP client indicates the e-mail address of the sender (the person who generated the message) and the e-mail address of the receiver.
- SMTP can count on the reliable data transfer service **of TCP to get the message to the server without errors**.

Application Layer
Accessing mails

- Deployment of **SMTP servers is more reliable** with respect to direct user-to-user mailing:
  - Servers are always on.
  - Servers are certified.
  - The server continuously tries to re-send mails if delivery fails (which is a quite computationally expansive process).
- On the other hand, to access the mails from the server **an additional client-server application is needed** (and a different protocol), most popular are:
  - Post Office Protocol—Version 3 (POP3).
  - Internet Mail Access Protocol (IMAP).
  - HTTP.

Application Layer
Accessing mails: POP3

- The **Post Office Protocol - Version 3** (POP3) is an extremely simple mail access protocol.
- The user agent (the client) opens a **TCP connection** to the mail server (the server) on port 110.
- With the TCP connection established, POP3 progresses through three phases:
  1. **Authorization**: the user agent **sends a username and a password** to authenticate the user.
  2. **Transaction**: the user agent can **retrieve messages**, mark messages for deletion, remove deletion marks, and obtain mail statistics.
  3. **Update**: after the client has issued the quit command, ending the POP3 session, the mail **server deletes the messages** that were marked for deletion.

Application Layer
Accessing mails: IMAP

- With POP3 access, messages can just **be downloaded or deleted**. Action like searching or organizing mails into folders are not considered (these must be done on the local machine).
- The **Internet Mail Access Protocol** (IMAP) allow **servers to provide additional features:**
  - Managing and **creating folders.**
  - Perform **search** into remote folders for messages matching specific criteria.
  - Allow user agent to obtain just parts of messages (e.g., header only, attachments only, etc.).
  - Allow **multiple clients** to be connected to the same server.

Application Layer
Accessing mails: HTTP

- More and more users today are sending and accessing their **e-mails through their Web browsers**.
- HTTP Web-based access was introduced by Hotmail in the mid 1990 an is now the mainstream approach (Google, Yahoo!, Virgilio, etc.) and it is used by almost every major university and corporation (UNINA included).
- With this service, **the user agent is an ordinary Web browser**, and the user communicates with its remote mailbox via HTTP rather than through the POP3 or IMAP protocols.
- This works for user-agents, while mail **servers still rely on the standard SMTP** to exchange messages.

Application Layer
DNS

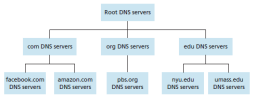
- There are two ways to identify a host: by **hostname and by IP address**. People prefer the more mnemonic hostname identifier, while network devices prefer fixed-length, hierarchically structured IP addresses.
  - Example: www.unina.it -> 143.225.15.50
- The **Domain Name System** (DNS) is an application-layer protocol that manages translation from hostnames to IP addresses.
- It is a **client-server protocol** in which a DNS-client asks to a DNS-server for a specific hostname-to-address translation.
- DNS **servers are often UNIX machines** running the Berkeley Internet Name Domain (BIND) software, **typically using UDP** connection and port number 53.

Application Layer
DNS

- For example, if a browser (HTTP client) requests the www.someschool.edu/index.html URL, the related IP address is retrieved as follows:
  1. The **browser extracts the hostname**, www.someschool.edu, from the URL and passes the hostname to the client side of the DNS application (in C/C++ applications we use the gethostbyname() function to perform that).
  2. The DNS **client sends a query** containing the hostname to a DNS server.
  3. The DNS **client receives a reply** containing the IP address for the hostname.
  4. Once the browser receives the IP address from DNS, it can initiate a **TCP connection to the HTTP server** process located at port 80 at that IP address.
- Notice that DNS process is not trivial! **it may add an additional delay** (sometimes substantial) to the Internet applications.

Application Layer
DNS: Distributed Servers

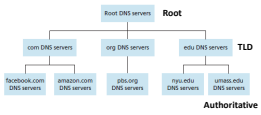
- Internet **DNS is distributed and hierarchically organized**. There are several servers all around the world that provide DNS service.



- This approach is preferred to a centralized DNS for several reasons:
  - **Avoid a single point of failure**: if the DNS server crashes, so does the entire Internet!
  - **Regulate traffic volume**: hundreds of millions of hosts uses DNS.
  - **Better reachability**: a single DNS server cannot be “close to” all clients.
  - **Easy maintenance and update**: we can update a local DNS without effecting the whole Internet.

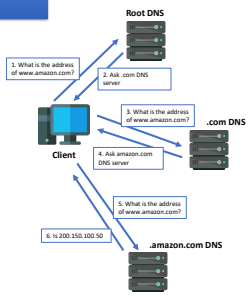
Application Layer
DNS: Hierarchy

- There are basically 3 classes of DNS servers:
  - **Root DNS servers**: there are over 400 root name servers scattered all over the world, managed by 13 different organizations. Root name servers provide the IP addresses of the TLD servers.
  - **Top-level domain** (TLD) servers: there are one or more for each top-level domain (e.g., .com, .org, .net, .it, .uk, .fr, etc.). TLD servers provide the IP addresses for authoritative DNS servers.
  - **Authoritative DNS** servers: provide real hostname-IP mapping. These can be directly owned by organizations (amazon, facebook, etc.) or offered by third-party.



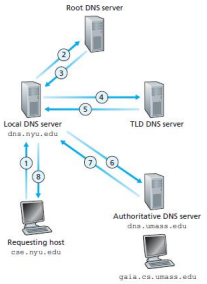
Application Layer
DNS: Example

- When a DNS client wants to determine the IP address for the hostname (e.g., www.amazon.com) it happens that:
  1. The client first **contacts one of the root servers**, which returns IP addresses for TLD servers for the top-level domain ".com".
  2. The client then **contacts one of these TLD servers**, which returns the IP address of an authoritative server for "amazon.com".
  3. Finally, **the client contacts one of the authoritative servers** for amazon.com, which returns the IP address for the hostname "www.amazon.com".



Application Layer
DNS: Resolver

- There is another important type of DNS server called the **local DNS server** (or **DNS resolver**).
- A local DNS server does not strictly belong to the hierarchy of servers and is **typically managed by ISPs**.
  - Google also provides similar servers all around the world (Google Public DNS) on address 8.8.8.8 or 8.8.4.4.
- When a host connects to an ISP, the ISP provides the host with the IP addresses of **one or more of its local DNS servers** (that are typically "close to" the host).
- When a host makes a DNS query, the query is sent to the local DNS server, which **acts as a proxy**, forwarding the query to the DNS server hierarchy.

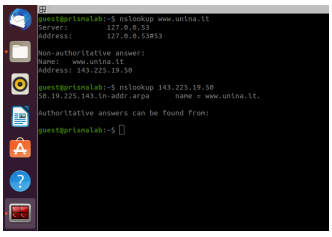


Application Layer
DNS: Aliasing

- DNS provides an **aliasing** service for servers in which complicated/long names (aka canonical) are associated to a more simple/short ones (aka alias).
- **Host aliasing**: hosts with complicated hostnames (e.g., web sites such as relay1.west-coast.enterprise.com) can be **associated to more mnemonic aliases** (e.g., enterprise.com or www.enterprise.com).
- Note that DNS can still be invoked by an application **to obtain the canonical (original) hostname** for a supplied alias as well as the associated IP address.

Application Layer
Lookup Example

- In Linux (and other OSs) we can use nslookup (name server lookup) command to ask for a hostname-address translation.



We can get translations in both ways:

- To get IP address from hostname (first).
- To get hostname from IP address (second).

Application Layer
DNS: Load Distribution

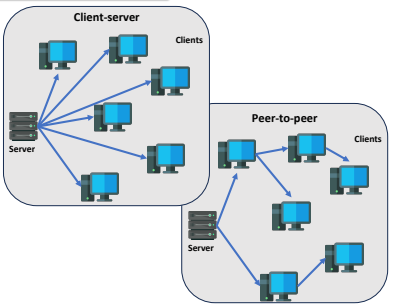
- A DNS can be used to perform **load distribution** among replicated servers (aka, **round-robin DNS**).
- Typically, busy sites (e.g., google, amazon, CNN, etc.) are **replicated over multiple servers**, with each server running on a different end system and each having a different IP address (multiple IP addresses associated with one canonical hostname).
- The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server may **rotate the order in which addresses are replied**.

Application Layer
Peer-to-Peer

- Differently from client-server, P2P architecture **makes minimal (if none) use of servers**, here we have **pairs of intermittently connected hosts** (peers) that communicate directly with each other.
- The peers are **not owned by a service provider** but are instead desktops and laptops controlled by users.
- One natural application of P2P is **file sharing**:
  - In a **client-server** architecture, the server must **share files to all clients** (which is a serious burden on the server and requires large bandwidth).
  - In a **P2P** architecture, the peers that receive the file may also **share it to other peers**. Somehow, peers are clients and servers at the same time.

Application Layer
Peer-to-Peer

- In file sharing, the P2P approach typically **scales better** than client-server one.
- On the other hand, P2P approaches can be quite **complex to implement**.
- There may also be **security issues** in having all this clients in direct communication.



Application Layer
BitTorrent

- A popular P2P protocol for file sharing/distribution is **BitTorrent**.
  - BitTorrent estimated users are 150-170 million (in 2023).
- A **torrent** is the collection of all peers participating in the distribution of a particular file. Peers in a torrent download **equal-size chunks** of the file from one another (typical chunk size is 256 kbytes).
- When a peer joins a torrent (having no chunks):
  - It starts by **accumulating chunks**.
  - While the peer downloads chunks **it also uploads chunks** to other peers.
  - Once the peer **acquires the entire file**, it may (selfishly) **leave the torrent**, or (altruistically) **stay in the torrent** and continue to upload chunks to other peers.
- Peers may **leave the torrent at any time** with only a subset of chunks, and later rejoin the torrent.

Application Layer
BitTorrent: tracker

- Each torrent has an infrastructure node called **tracker** that takes track of the peers participating to the torrent (trackers are basically **servers**).
- When a **new peer** joins a torrent:
  - It **registers** itself with the tracker and **periodically informs** the tracker about its status.
  - The tracker **provides the IP addresses** of a randomly selected subset of peers from the torrent.
- Having the list of peers, the new host attempts to establish concurrent TCP **connections with all the peers on this list** (neighbors).
  - During the execution, some of the connected peers **may leave** while other peers (outside the initial list) may attempt to **establish new connections**.
- At any given time, each peer will have **a subset of chunks from the file**, with different peers having different subsets. Periodically, a host will ask each connected peers (over the TCP connections) for **the list of the chunks they have**.

Application Layer
BitTorrent

- The **downloading client** decides which chunks to request (and to whom) following a rarest-first principle:
  - **Rarest-first**: the chunks with **fewest repeated copies** among the neighbors are prioritized. In this way, the **rarest chunks get more quickly redistributed**, so to (roughly) equalize the numbers of copies in the torrent.
- The **uploading client** decides which requests are served following two intertwined principles:
  - **Trading**: hosts give priority to **the best 4 neighbors** that are currently supplying data at the **highest rate**. This check is periodically performed (10 seconds) and the list of 4-best neighbors is updated.
  - **Random selection**: every 30 seconds, a host also picks **one additional neighbor** at random and sends it chunks. If this **random exchange is good** (the 2 hosts are good partners) **they will enter the respective best lists**. This process also allows peers with compatible upload rates to find each other.

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

DIETI UNINA

Application Layer
Creating Network Applications

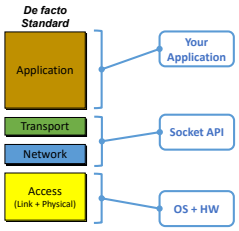
- So far, we have looked at several network applications and protocols, we will now see **how network applications can be created**.
- Most network applications include a **client-side** program and a **server-side** program that communicate through the network.
- When these two programs are executed, a client process and a server process are created, and these processes communicate with each other by reading from, and writing to, **sockets**.
- When creating a network application, the developer's main task is therefore to write the code for both the client and server programs.

Application Layer
Creating Network Applications

- There are two types of network applications:
  - Applications relying on a **standard protocol**. There are several "open" protocols whose rules are known. In this case, the client and server programs must conform to the rules to be compatible with the protocol.
  - Applications relying on a **proprietary protocol**. In this case the client and server programs employ a custom protocol (not been openly published in an RFC or elsewhere). In this case, a single developer (or development team) creates both the client and server programs.
- There are two transport protocols that can be used:
  - **TCP** (Transmission Control Protocol), which is **connection oriented** and provides a reliable byte-stream channel through which data flows between two end systems.
  - **UDP** (User Datagram Protocol), which is **connectionless** and sends independent packets of data from one end system to the other, without any guarantees about delivery.

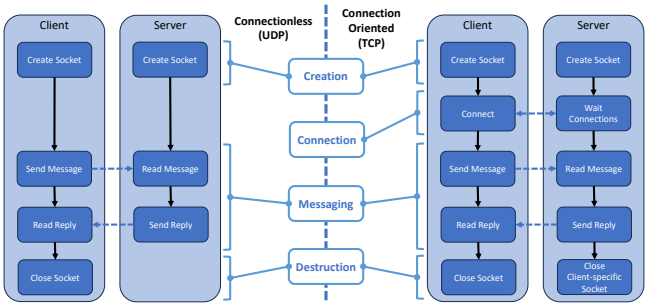
Application Layer
Creating Network Applications

- **Sockets** are a central element for the creation of network applications.
- They typically manages (black-box) the transport (UDP/TCP) and the network (IP) layers functionalities.
- Note that there are several applications (middleware) that may **wrap sockets** simplifying their creation and providing more complex functionalities.
- However, **basic API are still widely used**.



Where layers are implemented

Application Layer
Connection-oriented vs. Connectionless



Application Layer
UDP and TCP

- UDP and TCP sockets implement the previous procedure to allow connectionless and connection-oriented communication, respectively.
- The basic APIs providing UDP and TCP sockets are typically available in almost **all programming languages and operating systems**:
  - C, C++, C#, Java, Perl, Python, Matlab, etc.
- The underlying implementation and the usage can be more or less different depending on the configuration of the machines.
- In Unix domain we use **Berkeley sockets** (aka BSD or POSIX sockets) for both TCP and UDP connections. These are native C APIs.

Application Layer
Sockets: Definitions

- Internet-domain sockets in C rely on some structure to define the addresses and ports of the sending/receiving hosts:

```
#include <netinet/in.h>
#include <arpa/inet.h>          // for inet_addr()
#include <sys/types.h>          // some custom types are defined here


struct sockaddr_in {
   short          sin_family;   // family of the address, typically set to AF_INET (IPv4)
   unsigned short sin_port;     // port number, e.g. htons(3490)
   struct in_addr sin_addr;     // see struct in_addr below
   char           sin_zero[8];  // typically zeros, now this structure has same size as sockaddr and can be casted to it
};

struct in_addr {
   unsigned long s_addr;  // IP address, can be loaded with inet_addr() or set to INADDR_ANY for localhost
};
```

Application Layer
Sockets: Creation

- Sockets are created in C by using the socket() function:

```
#include <sys/socket.h>

int sockfd = socket(int domain, int type, int protocol);
```

Where:

- domain: specifies the family as in previous structure (AF_INET).
- type: specifies the type of socket to be created.
  - SOCK_STREAM -> TCP socket
  - SOCK_DGRAM -> UDP socket
- protocol: Specifies a particular protocol to be used within the socket.
  - This is often 0, so no specific protocol is used.
- sockfd: is the file descriptor that identifies the newly created socket.
  - Notice that sockets also follow the Unix philosophy that "everything is a file" where I/O sources are treated as files (and associated to a file descriptor).

Application Layer
Sockets: Binding

- We can associate a socket to local address and port by using the bind() function:

```
#include <sys/socket.h>

int val = bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Where:

- socket: is the file descriptor of the socket we want to bind.
- address: pointer to a sockaddr structure specifying on which port and address to bind.
  - address.sin_addr often set to INADDR_ANY so connections from all addresses are listened (multiple IPs).
- address_len: specifies the length of the sockaddr structure pointed to by the address argument (we can use sizeof() to get it).
- val: return value which is 0 on success, and -1 otherwise.

- Notice that this function **is used on servers** as we must bind the socket to a specific port but is **optional on clients** (the OS assigns a free port automatically).

Application Layer
Sockets: Send and Receive (UDP)

- Processes of sending or receiving messages in UDP are performed by the sendto() and recvfrom() functions:

```
#include <sys/socket.h>

int ob = sendto(int osock, const void *obuf, size_t olen, int oflags, const struct sockaddr *oaddr, socklen_t oaddr_len);
```

```
#include <sys/socket.h>

int ib = recvfrom(int isock, void *ibuf, size_t ilen, int iflags, struct sockaddr *iaddr, socklen_t *iaddr_len);
```

Where:

- osock/isock: are the file descriptors on which to send/receive a message.
- obuf/ibuf: are pointers to the buffers containing a message to send/receive.
- olen/ilen: are lengths of the messages in bytes.
- oflags/iflags: specifies flags (typical values are 0 or MSG_WAITALL to wait until all olen/ilen bytes are sent/received).
- oaddr/iaddr: pointers to sockaddr structures containing the receiving/sending address.
- oaddr_len/iaddr_len: length in bytes of the sockaddr structure.
- ob/ib: number of bytes that are actually sent/received.

Application Layer
Sockets: Closing

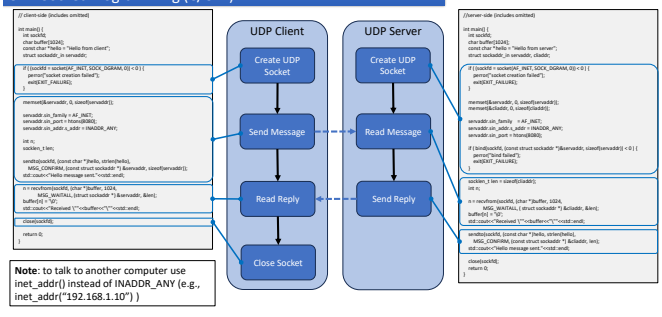
- Sockets can be closed by using the close() function:

```
#include <unistd.h>

int val = close(int socket);
```

Where:

- socket: is the file descriptor of the socket to close
- val: return value which is 0 on success, and -1 otherwise.

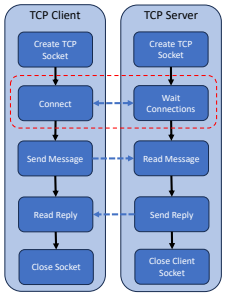
Application Layer
UDP Socket Programming (C/C++)



Note: to talk to another computer use inet_addr() instead of INADDR_ANY (e.g., inet_addr("192.168.1.10"))

Application Layer
From UDP to TCP

- Unlike UDP, in TCP client and server need to **handshake** before messages can be transmitted.
- We use **two sockets on the server-side**:
  - A **welcoming socket**, which is always on, and allows clients to perform handshakes with the server.
  - A **client-specific socket**, which is created after the handshake, and is used by client and server to communicate.
- The three-way handshake, which takes place within the transport layer, is completely **invisible to the client and the server programs**.
- Once the connection is accepted by the server (handshake success) **the communication moves to the newly created socket**.



Application Layer
Sockets: Connection (TCP only)

- Client-side TCP connection is performed by using the connect() function:

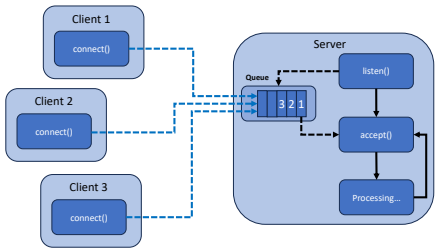
```
#include <sys/socket.h>

int val = connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

Where:

- socket: is the file descriptor of the socket we use to connect.
- address: pointer to a sockaddr structure containing the server address.
- address_len: specifies the length of the sockaddr structure pointed to by the address argument (we can use sizeof() to get it).
- val: return value which is 0 on success, and -1 otherwise.

Application Layer
Sockets: Wait-for-connection (TCP only)

- Server-side TCP wait-for-connection is performed through two functions: listen() and accept(), which works in combination with the clients connect().
- The listen() opens a queue where incoming connections are stored.
- The accept() opens the client-specific socket.



Application Layer
Sockets: Wait-for-connection (TCP only)

- Server-side TCP wait-for-connection is performed by the listen() and accept() functions:

```
#include <sys/socket.h>

int val = listen(int socket, int backlog);
int new_sockfd = accept(int socket, struct sockaddr *address, socklen_t *address_len);
```

Where:

- socket: is the file descriptor of the socket on which to wait for connections (must be bound to address/port).
- backlog: maximum length of the queue of pending connections.
- address: can be a null pointer, or a pointer to a sockaddr structure where the address of the connecting socket shall be returned.
- address_len: length of the address.
- new_sockfd: new socket descriptor on which the client and the server will communicate.
- val: return value which is 0 on success, and -1 otherwise.

Application Layer
Sockets: Send and Receive (TCP)

- Processes of sending or receiving messages in TCP are performed by the send() and read() functions (simpler than UDP ones):

```
#include <sys/socket.h>

int ob = send(int osock, const void *obuf, size_t olen, int flags);
```

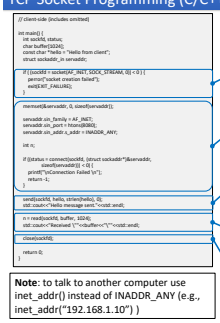
```
#include <unistd.h>

int ib = read(int isock, void *ibuf, size_t ilen);
```

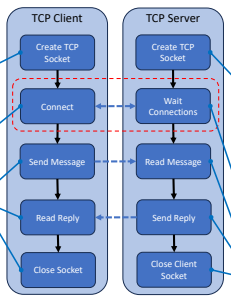
Where:

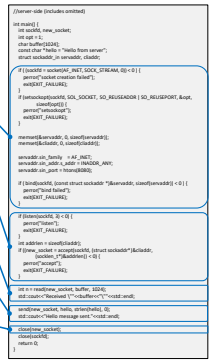
- osock/isock: are the file descriptors on which to send/receive a message.
- obuf/ibuf: are pointers to the buffers containing a message to send/receive.
- olen/ilen: are lengths of the messages in bytes.
- flags: specifies the type of message transmission (depends on protocol, typically 0).
- ob/ib: number of bytes that are actually sent/received.

- Notice that we have already specified client and server addresses during the connection phase, we no need to do it here as in sendto/recvfrom.

Application Layer
TCP Socket Programming (C/C++)







Note: to talk to another computer use inet_addr() instead of INADDR_ANY (e.g., inet_addr("192.168.1.10"))

Computer Network I
Reti di Calcolatori I

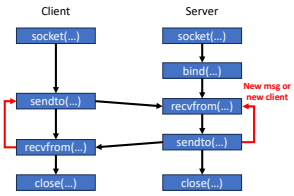
Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

DIETI UNINA

Application Layer
UDP Socket Commands

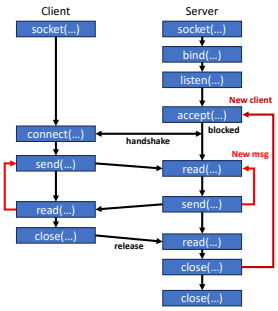


- For UDP connection we need both client and server to open a socket.
- The server has to bind the socket to a specific port.
- Then we can exchange messages using sendto and recvfrom functions.
  - Here IP and port of the client are retrieved through the recvfrom function.
  - We may loop over sendto/recvfrom to keep alive the communication.
- When communication is over, both hosts close the socket.
  - Notice that if client only closes the socket, the server may accept other clients.

Application Layer
UDP Socket Programming (Keepalive Example C/C++)

Outsourced…

Application Layer
TCP Socket Commands



- For TCP connection we need both client and server to open a socket.
- The server has to bind the socket to a specific port, listen and eventually accept new connection. The client has to connect to the server.
- Then we can exchange messages using send and read functions.
  - Here IP and port of the client are retrieved from the newly created socket.
  - We may loop over send/read to keep alive the communication.
- When communication is over, both hosts close the socket.
  - Notice that the server should perform an additional read before to close the client-specific socket (connection release) otherwise we must wait about 4 minutes to reuse the port.
- The server may also go back to the accept function and welcome another client.

Application Layer
TCP Socket Programming (Keepalive Example C/C++)

Outsourced…

Application Layer
DNS Translation (C/C++)

- As just shown, we can create sockets between two hosts, but **we need to know (IP) address and port of the server**.
- On Internet, hosts are typically identified by **hostnames rather than by IP** addresses.
- Berkeley sockets (C/C++) only works with IP addresses.
  - This is not surprising as **sockets works on transport layer**, while **hostnames works on application layer**.
- If we are to connect with a host by hostname, **we need to use DNS translation** and get the associated IP address to be passed to the socket.

Application Layer
DNS Translation (C/C++)

- C/C++ offers some functions and structures that perform such translation for us, which are used **to contact DNS servers** and to retrieve addresses.
- In particular, we can use the function *gethostbyname()* to contact DNS servers.
- The above function **does not return just the IP address**. It returns a *hostent* structure that contains some info about the host:
  - Canonical name.
  - Possible aliases.
  - One or more addresses.

Application Layer
DNS Translation (C/C++)

- The hostent structure is defined as follows:

```
#include <netdb.h>

struct  hostent {
    char *  h_name;             // original name of the host (canonical)
    char ** h_aliases;          // list of aliases (terminated by a NULL pointer)
    int     h_addrtype;         // family of the address (typically AF_INET)
    int     h_length;           // length of the address (typically 4 bytes)
    char ** h_addr_list;        // list of addresses (terminated by a NULL pointer)
};

// for compatibility reasons
#define h_addr h_addr_list[0]
```

- A DNS may provide a list of addresses (which may be randomly ordered if the query is sent to round-robin DNS servers).
  - We can get the address of the first element of the array (h_addr_list[0] or h_addr).
- The returned addresses can be casted to *in_addr* structure used by sockets.

Application Layer
DNS Translation (C/C++)

- The gethostbyname() function is defined as follows:

```
#include <netdb.h>

struct hostent *host_info = gethostbyname(const char *name);
```

- Where:
  - name: is a string containing the hostname to be translated.
  - host_info: is a pointer to the hostent structure containing the information about host.
    - This is NULL if error occurred.
- Notice that there is also a similar function *gethostbyaddr()* that returns a hostent structure for a specified IP address.

Application Layer
DNS Translation (C/C++)

```
#include <iostream>
#include <cstring>
#include <string>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <stdio.h>
```

- Include libraries to allow DNS translation.

```
int main(int argc, char **argv){
  struct hostent *server_info;
  char *server_name;

  if(argc < 2){
     std::cout<<"No hostname specified"<<std::endl;
     return 0;
  }
  else {
     server_name = argv[1];
  }

  server_info = gethostbyname(server_name);
  if (server_name == NULL){
     std::cout << "could Not resolve hostname " << server_name << " :(" << std::endl;
     return 0;
  }
```

- Initialization.
- Get hostname as argument.
- Invoke DNS and check if a response has been received.

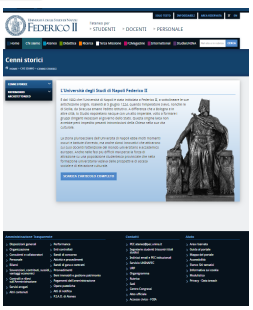
Application Layer
DNS Translation (C/C++)

```
  //plot output
  std::cout<<"Canonical name:"<<std::endl;
  std::cout<<"\t"<<server_info->h_name<<std::endl;

  std::cout<<"Aliases:"<<std::endl;
  int i = 0;
  while(server_info->h_aliases[i] != NULL){
     std::cout<<"\t"<<server_info->h_aliases[i]<<std::endl;
     i++;
  }

  std::cout<<"IPs:"<<std::endl;
  i = 0;
  while(server_info->h_addr_list[i] != NULL){
     std::cout<<"\t"<< inet_ntoa( *( (struct in_addr *) server_info->h_addr_list[i] ) )<<std::endl;
     i++;
  }

  return 0;
}
```

- Plotting output from hostent structure:
  - Canonical name.
  - List of aliases
  - List of IPs.

Application Layer
HTTP Socket Example

- We will now create a C++ code implementing from scratch a HTTP request (HEAD) for a web page.
- The web page we will check is the “cenni storici” from the UNINA website:
  - http://www.unina.it/chi-siamo/cenni-storici



Application Layer
HTTP Socket Example (C/C++)

```
#include <iostream>
#include <cstring>
#include <string>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <stdio.h>
```

- Include libraries to allow TCP connection to the web server.

```
int main(){
  int socket_desc;
  struct sockaddr_in serv_addr;
  struct hostent *server;
  char buffer[4096];

  socket_desc = socket(AF_INET, SOCK_STREAM, 0);
  if (socket_desc < 0){
     std::cout << "failed to create socket" << std::endl;
     return 0;
  }
```

- Initialize variables and create the TCP socket. For a HTTP request 3 elements are specified:
  - Hostname of the web server.
  - Port number.
  - Resource to be retrieved (object’s path).
- Note: here we use the C++ cout for simplicity.

Application Layer
HTTP Socket Example (C/C++)

```
server = gethostbyname("www.unina.it");
if (server == NULL){
  std::cout << "could Not resolve hostname :("
    << std::endl;
  close(socket_desc);
  return 0;
}

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(80);
bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);

if (connect(socket_desc, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
  std::cout << "connection failed :(" << std::endl;
  close(socket_desc);
  return 0;
 }
```

- Connection to the web server. Here we use the following function:

  server = gethostbyname(host.c_str());

  that invokes DNS and to gets the IP of the server from the hostname (inside the *hostent* struct).

- The IP from the returned structure is copied into the *serv_addr* structure for the following connect function.

Application Layer
HTTP Socket Example (C/C++)

```
const char *request = "HEAD /chi-siamo/cenni-storici HTTP/1.1\r\nHost: www.unina.it\r\nConnection: close\r\n\r\n";

if (send(socket_desc, request, strlen(request), 0) < 0){
  std::cout << "failed to send request..." << std::endl;
  close(socket_desc);
  return 0;
}
std::cout<<"message sent:"<<std::endl;
std::cout<<request<<std::endl;

int n = recv(socket_desc, buffer, sizeof(buffer), 0);
```

- Messaging with the web server. The HTTP request is defined into the request string.
- Here we are creating a non-persistent connection since we just want to check one page.

```
std::cout<<"received "<<n<<" bytes:"<<std::endl;
std::cout<<buffer;

close(socket_desc);

return 0;
```

- Close socket and plot the received page (HTML).

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Transport Layer
From Application to Transport

- A transport-layer protocol mostly provide **logical communication between application processes** running on different hosts.
  - From the application's perspective, hosts running **the processes looks directly connected** as if they were on the same machine (even if they are on opposite sides of the planet).
- The **transport layer converts the application-layer messages into** one or more transport-layer packets, called **segments** or **datagrams** (the latter is mainly used for UDP packets):
  - If application messages are broken into smaller chunks (segments/datagrams), **each chunk is provided with transport-layer header**.
  - Segments are **passed one by one to the network layer** to be transmitted.

Transport Layer
Responsibilities of Transport Layer

- Transport-layer protocols (UDP and TCP) have four main responsibilities:
  1. **Process-to-process delivery:** messages are delivered independently of where these processes are.
     - Note that this is **different from host-to-host delivery** (i.e., between two different machines), which is responsibility of the lower-level IP protocol.
  2. **Integrity checking:** by including error detection fields into the segments' headers.
  3. **Reliable data transfer**: ensuring that data is delivered from sending process to receiving process, correctly and in order.
  4. **Congestion/flow control**: it prevents connection from swamping network devices (links or routers) with an excessive amount of traffic. This service is useful to improve performance and **is very beneficial to the whole (internet) network**.
- In particular, UDP (faster but simpler) provides **only the first two services**, while TCP provides all the four ones.

Transport Layer
Process-to-process Delivery

- At the application level, **several processes can be accessing the network at the same time**.
  - For example: listening to Spotify music, downloading files, navigating web sites, all together.
- A **process may also be connected to multiple processes** at the same time:
  - For example: proxy servers, mail servers, etc., all may be connected to multiple remote processes.
- To solve these issues, we use sockets for process-to-process delivery:
  - Instead of delivering the message directly to a specific process, we **use sockets as end-points from which processes get the messages**.
  - This approach somehow **decouples** the number of processes from the number of connections.

Transport Layer
Process-to-process Delivery

- Since there are multiple sockets in the receiving host, we need:
  - A **unique identifier** for each socket (**port**) that must be attached to the received segments.
  - A **multiplexing/demultiplexing** process on the sender/receiver.
- **Demultiplexing**: is the process of redirecting a segment to the right socket depending on the associated identifier.
- **Multiplexing**: is the process of creating segments from different processes, assigning to them the right identifier.
- Notice that the problem of redirecting messages to multiple sources is quite common in computer networks, multiplexing and demultiplexing are also present in other layers (not only in transport).

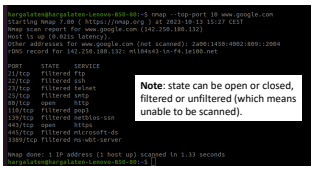
Transport Layer
Ports

- The identifiers of sockets are called source and destination ports (or simply ports):
  - A **port is a 16-bit number** (port number) ranging from 0 to 65535.
  - The port numbers ranging **from 0 to 1023 are called well-known port numbers** and are **restricted**, which means that they are reserved for use by well-known application protocols.
- The **list of well-known port** numbers is updated by **IANA** (Internet Assigned Numbers Authority), available at http://www.iana.org.
- When we develop a new network application, we must assign port numbers to sockets (and therefore to the applications) accordingly.

Port	Usage
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH)
23	Telnet - Remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) E-mail Routing
53	Domain Name System (DNS) service
80	Hypertext Transfer Protocol (HTTP) used in World Wide Web
110	Post Office Protocol (POP3) used by e-mail clients to retrieve e-mail from a server
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Mail Access Protocol (IMAP) Management of Digital Mail
161	Simple Network Management Protocol (SNMP)
443	HTTP Secure (HTTPS) HTTP over TLS/SSL

Transport Layer
Ports: Nmap

- To **check port usage** (and more) on a Linux machines we can use the nmap command.
- Nmap (Network Mapper) is a **network scanning utility** basically created to map (discover) hosts and services on a network.
- It can be used to probe ports of hosts **in order to detect open ones** (on which an application is listening) and possibly the associated services.
- Usage:
  - Port-scan of a given target:
    - $ sudo nmap [target address]
  - Probe port to determine services:
    - $ sudo nmap –sV [target address]
  - Check first N top used ports:
    - $ sudo –top-port [N] [target address]



Transport Layer
Multiplexing and Demultiplexing

- Assume we have a process in Host A (port 19157) that wants to send a message to a process (port 46428) in Host B.
- Multiplexing:
  - The transport layer in **Host A creates a segment/datagram that includes the application data**, the source port number (19157), the destination port number (46428).
  - The transport layer then **passes the resulting segment/datagram to the network layer** that encapsulates it in an IP datagram, which will make a best-effort attempt to deliver the segment to the receiving host.
- Demultiplexing:
  - If the segment/datagram arrives at the Host B, **the transport layer receives and decapsulates** it.
  - The transport layer then **passes the segment/datagram to the appropriate socket** by examining the segment's destination port number.

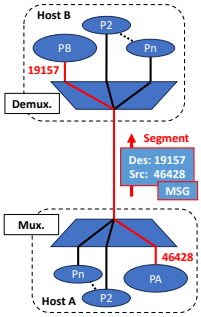
Transport Layer
Multiplexing/Demultiplexing in UDP (C/C++)

- In a C++ UDP client, socket is created as follows:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

- The socket is typically not bound to a specific source port, it is the OS that selects a free one (e.g., 46428) but we can always bind it to a specific port.
- Destination port/address is set by passing a suitable sockaddr_in structure to the sendto function:

```
servaddr.sin_port = htons(19157);
servaddr.sin_addr.s_addr = inet_addr( address_of_B );
...
sendto(sockfd, (const char *)msg, strlen(msg), 0,
        (const struct sockaddr *) &destaddr,
        sizeof(destaddr);
```



Transport Layer
Multiplexing/Demultiplexing in UDP (C/C++)

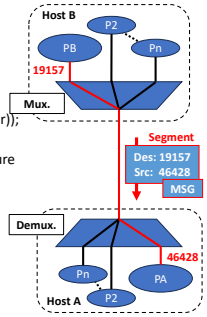
- In a C++ UDP server, socket is created and bound as follows:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
...
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(19157);
...
bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr));
```

- Destination port/address is retrieved through the sockaddr_in structure from the recvfrom function and used in the reply message:

```
recvfrom(sockfd, (char *)msg, 1024, 0,
        ( struct sockaddr *) &cliaddr, &len);
...
sendto(sockfd, (const char *)msg, strlen(msg), 0,
        (const struct sockaddr *) &cliaddr, sizeof(cliaddr);
```

- The reply will be sent back **whatever the client is**.



Transport Layer
Multiplexing/Demultiplexing in UDP

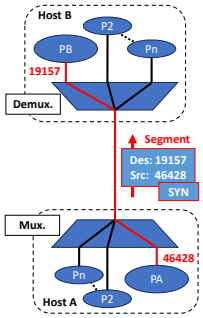
- We can say that a UDP socket can be identified **by two elements**:
  - Destination address (IP).
  - Destination port.
- In fact, we can use the same socket to send a return message **back to multiple source ports/addresses**.
- In the example, the cliaddr structure is filled by the recvfrom function, so we can use it as a destination address for the following sendto function, independently on who the host is.

Transport Layer
Multiplexing/Demultiplexing in TCP (C/C++)

- In TCP communication the server application **has a welcoming socket**, that waits for connection establishment requests from clients on a specific port number (19157 as before).
- The TCP client creates a socket and **sends a connection establishment request segment** to the host specified in the sockaddr_in structure (servaddr in this case):

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
...
servaddr.sin_port = htons(19157);
servaddr.sin_addr.s_addr = inet_addr( address_of_B );
...
connect(sockfd, (struct sockaddr*)&servaddr,
        sizeof(servaddr));
```

- A **connection-establishment request is just a TCP segment** with a destination port number (19157 here) and a special connection-establishment bit set in the TCP header (SYN = 1).

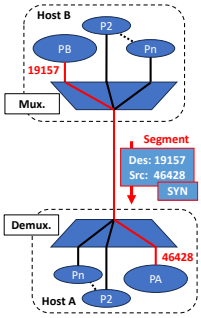


Transport Layer
Multiplexing/Demultiplexing in TCP (C/C++)

- When the server receives the incoming connection-request segment (with destination port 19157), it **locates the server process that is waiting** to accept a connection. Then a new socket is created:

```
new_socket = accept(sockfd,
        (struct sockaddr*)&cliaddr, (socklen_t*)&addrlen)
```

- The server-side transport layer fills the structure (cliaddr) **by using port numbers and addresses from the incoming segment** and creates a new socket (new_socket).
- All future segments having these specific source port, source IP address, destination port, and destination IP address **are redirected (demultiplexed) to new_socket**.

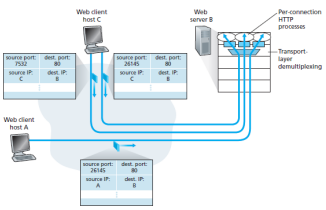


Transport Layer
Multiplexing/Demultiplexing in TCP (C/C++)

- We can say that a TCP socket can be identified **by four elements**:
  - Source address (IP).
  - Source port.
  - Destination address (IP).
  - Destination port.
- Because of the initial handshake the TCP connection **entangles the processes** of both sides of the socket.
- Only one source and one destination will use this socket, if the client or the server sides of the communications are interrupted, the socket is closed on both sides (which does not happen in UDP).

Transport Layer
Multiplexing/Demultiplexing HTTP example

- We can then have multiple processes from one machine communicating with multiple processes (or with the same process) on another.
- In this example **a host C initiates two HTTP sessions to server B**, and **host A initiates one HTTP session to B**. Hosts A and C and server B each have their own unique IP address (A, C, and B). Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections.
- Host **A may assign a source port of 26145** to its HTTP connection (not aware of C).
- This is not a problem, since **the two connections have different source IP** addresses.
- In this example, the server opens a new process (or a new thread) for each incoming connection by assigning to it the specific socket (new_socket).
- This is quite typical but opening too many processes (or threads) may impair the performance of the server.



Transport Layer
UDP

- As already explained, **UDP is useful for simple and rapid connections**. It mainly performs 2 tasks:
  - Process-to-process delivery (ports, multiplexing and demultiplexing).
  - Error Checking.
- UDP is connectionless: **there is no handshaking** between sending and receiving transport-layer entities before sending a segment.
- UDP has **no guarantee about delivery of messages** (no reliable data transfer implemented) and has **no congestion/flow control**.
- Roughly speaking, UDP is a **minimalistic protocol** that just adds "the essential" with respect to the lower-level IP protocol.

Transport Layer
UDP

- Why to use UDP instead of TCP?
  - **Application-level control**: UDP is more direct, so the application has more control on the transmission.
    - This can be useful for instance in real-time applications, where sending rate or delays are crucial while data loss can be tolerated, so less checking is preferred.
  - **Fast connection establishment**: there is no handshake, so less delay to establish a connection.
    - This is good for instance in DNS to not provide additional delays.
  - **No connection state**: there are no congestion-control parameters, no sequence nor acknowledgment numbers.
    - A server can typically support many more active clients when the application runs over UDP rather than TCP.
  - **Minimal packet overhead**: The TCP segment adds 20 bytes of header in every segment, whereas UDP only adds 8 bytes.

Transport Layer
UDP: DNS Example

- An example of application using UDP connection is DNS. The DNS client works as follows:
  - In **application layer**: when the host wants to make a query to a DNS server, it constructs a DNS query message and passes the message to UDP.
  - In **transport layer**: without performing any handshaking with the DNS server, UDP adds header fields to the message and passes the resulting segment to the network layer.
  - In **network layer**: the UDP segment is encapsulated into an IP datagram and sent to the DNS server (through the access layer).
  - The client then waits for a reply to its query. If **reply is not received** (possibly because the underlying network lost the query or the reply), it might try different approaches:
    - Resending the query.
    - Sending the query to another name server.
    - Informing the invoking application that a reply is not received.

Transport Layer
UDP: Usage

- Applications needing **reliable data transfer** such as, e-mail, remote terminal access, the Web, run over TCP.
- In this cases **we can't afford packet loss**, for instance, mails must be fully delivered we can't have missing elements.
- **SNMP** uses UDP to carry out device management. Network management applications **must often run when the network is in a stressed state** (precisely when reliable data transfer is difficult).
- **Multimedia applications** such as Internet phone, video conferencing, streaming, often use both UDP and TCP, because small amount of packet loss can be tolerated.
- In general, real-time applications react very poorly to TCP congestion control.

Service	Application Protocol	Transport Protocol
E-mails	SMTP	TCP
Remote terminal	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Name translation	DNS	UDP
Network device management	SNMP	UDP
Multimedia streaming	Proprietary	UDP or TCP
Internet telephony	Proprietary	UDP or TCP

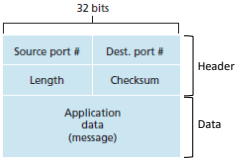
Transport Layer
UDP: Usage

- On the other hand, running **multimedia applications over UDP is controversial** because no congestion control is performed.
- If everyone were to (selfishly) start streaming high-bit-rate video without using any congestion control, **network devices would overflow** causing more UDP packets to be lost.
- High loss rates induced by the uncontrolled UDP senders would also cause the **TCP senders to dramatically decrease their rates**.

Service	Application Protocol	Transport Protocol
E-mails	SMTP	TCP
Remote terminal	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Name translation	DNS	UDP
Network device management	SNMP	UDP
Multimedia streaming	Proprietary	UDP or TCP
Internet telephony	Proprietary	UDP or TCP

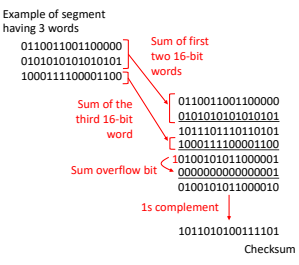
Transport Layer
UDP: Segment Format

- The **UDP segment** (or datagram to be more precise) is composed by 5 elements:
  - **UDP header** (64 bits) which includes information related to the UDP protocol and is composed by **4 elements of 16 bits** each:
    - **Source port**: port number of the sending process.
    - **Destination port**: port number of the receiving process.
    - **Length**: length of the whole datagram (header+data, i.e., N + 64 bits).
    - **Checksum**: used by the receiver to check if the message is intact.
  - **Data** (N bits) which is the actual message from the application layer.



Transport Layer
UDP: Checksum

- Integrity checking is performed by using the **checksum**. The UDP checksum is a 16 bits filed used for error detection as follows:
  - UDP at the sender side performs the **1s complement of the sum of all the 16-bit words** in the datagram, with **any overflow bit being summed** itself.
  - This **result is put in the checksum field** of the UDP datagram.
  - When **the receiver sums all bits inside the message (checksum included)** the sum must be 1111111111111111 (16 ones).
  - If one bit is 0, then we know that **errors have been introduced** into the packet.



Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Transport Layer
Reliable Data Transfer Problem

- Let's assume we are at the railway station waiting for the train number 6 which should arrive on platform 5.
- The train changes platform, it will arrive on platform 9 instead of platform 5, a message is sent from station control to communicate it:

  "Train 6 will arrive on platform 9"

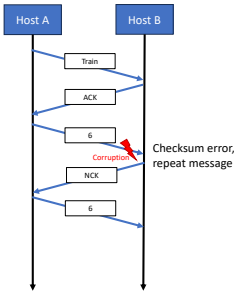
- What happens if the communication is unreliable:
  - Words could be lost: "Train %&! will arrive on platform 9"
  - Words could be altered: "Train 7 will arrive on platform 9"
  - Words could be swapped: "Train 9 will arrive on platform 7"
  - Words could be duplicated: "Train 66 will arrive on platform 9"
- It is possible you can understand that the message is wrong (for example in case 1), but you may also take the wrong train or miss the right one.

Transport Layer
Reliable Data Transfer

- Error control (e.g., through checksum) allows receiver at least to understand **if the message is corrupted**, but this is not enough. Reliable data transfer **is still one of the main problem of networking**.
  - The problem is not only addressed at the transport layer, but also at the link layer and (often) at the application layer.
- In a reliable channel 3 elements must be guaranteed:
  1. None of the transmitted bits **are corrupted** (flipped from 0 to 1, or vice versa).
  2. None of the transmitted bits **are lost or repeated**.
  3. All bits are delivered **in the exact order** in which they were sent.
- In general, we must assume the lower-level **network layer to be unreliable**.
  - This is a realistic assumption, for example, TCP is a reliable data transfer protocol that **is implemented on top of an unreliable (IP) end-to-end network layer**.

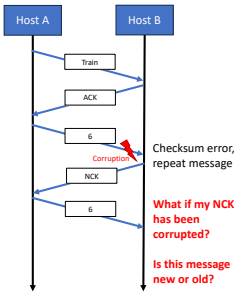
Transport Layer
Reliable Data Transfer: Packet Corruption and Stop-and-wait

- Let's assume we have **a channel where bits may only be corrupted** (not lost).
  - Corruption of bits is a typical problem, which is **mainly due to the physical components** of a network as a packet is transmitted, propagated, or buffered several time during the communication.
- **Stop-and-wait**: a first approach requires each message to be acknowledged before sending a new one:
  - **Positive acknowledgments** (ACK) means that the message has been received intact.
  - **Negative acknowledgments** (NCK) means that an error occurred, so the message must be repeated.
- In a computer network, **protocols based on retransmission are also known as ARQ** (Automatic Repeat reQuest) protocols.



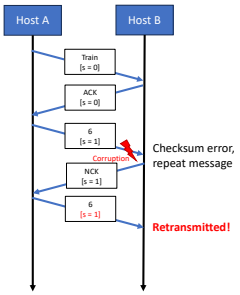
Transport Layer
Reliable Data Transfer: Stop-and-wait

- There are a couple of **issues** with this approach:
  - What happens if the **ACK/NCK message is corrupted** itself?
  - How can Host B be sure that **a message is the repetition instead of a new message**?
- **Sequence number**: add a new field to the headers of packets which specifies the order in which packages should be received.



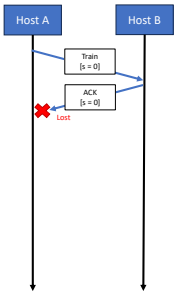
Transport Layer
Reliable Data Transfer: Stop-and-wait

- In a stop-and wait approach we always have **one message** on the line.
- There is no need to specify the whole sequence of packets, we just need to **differentiate between the current packet and the previous one**.
- Therefore, in this case, **we just need to add one bit** (*s = 0/1*) to the headers of packets.



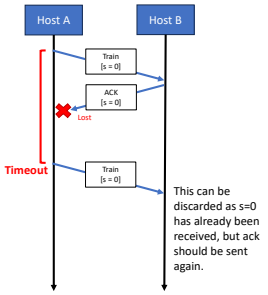
Transport Layer
Reliable Data Transfer: Packet Loss and Stop-and-wait

- Let's assume we now have **a channel where packets may also be lost**.
  - This is also quite reasonable as **network devices may have buffer overflow** in case of intense traffic.
- Here we have a clear problem with stop-and-wait approach: the loop. **If one message is lost, hosts will never send a new one**.
  - Notice that this happens whether we loose message or acknowledgment, as in both cases host A will not be triggered to send a new message.



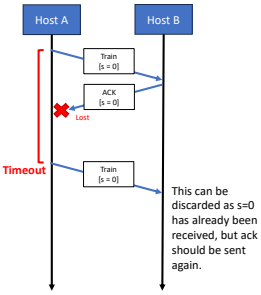
Transport Layer
Reliable Data Transfer: Packet Loss and Stop-and-wait

- **Timeout**: a very simple yet effective solution is to **add a timeout on the sender-side** that, once expired, allows the Host to try again.
  - A similar approach can be used in UDP-based DNS.
- This works **both in the case of packet loss and in acknowledgment loss**.
  - In case of duplicate the receive has just to discard the packet.



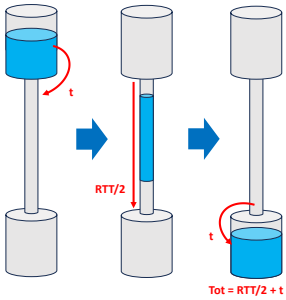
Transport Layer
Reliable Data Transfer: Packet Loss and Stop-and-wait

- The challenge here **is how to estimate a suitable timeout** (it depends on the RTT).
  - Too long timeout causes communication to be slowed.
  - Too short timeout causes the packets to overlap.
- We can say that a reasonable timeout should be somehow longer than the RTT.
- Timeout saves stop-and-wait from loop, but the **performance are quite bad**…



Transport Layer
Reliable Data Transfer: Water Example

- We can see **data transfer as a water flow problem**:
  - There is **the time (t) to move the water from the tank to the tube**. While entering **the water is also moving** toward the tube.
  - After **t all the water from the tank is inside the tube** (the initial tank is empty as well as the destination tank).
  - It **takes a certain time (RTT/2) for the water to flow** into the tube.
  - Finally, **the water arrives at the destination tank** and (by assuming same download/upload time) it takes the same time t to pour out of the tube (RTT/2 + t to receive the last drop).



Transport Layer
Reliable Data Transfer: Performance of Stop-and-Wait

- Let's consider the idealized case of **two hosts located on the opposite coasts** of the United States.

- The speed-of-light RTT between these two end systems **is approximately 30 milliseconds** (0.03 sec.). Let's assume to have:
  - A channel with a transmission rate (R) of 1 Gbps (10^9 bits per second).
  - A packet size (L) of 1000 bytes (8000 bits).
- The time (t) needed to transmit the packet to the channel is:

$$t = \frac{L}{R} = \frac{8000}{10^9} = 0.000008 \text{ (8 microseconds)}$$

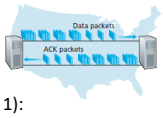
Transport Layer
Reliable Data Transfer: Performance of Stop-and-Wait

- Let's consider the idealized case of **two hosts located on the opposite coasts** of the United States.

- In a **stop-and-wait** protocol:
  - the **sender begins sending** the packet at t_0,
  - the **last bit enters the channel** at t_0 + 0.000008 sec.,
  - the **packet takes 0.015 sec. to reach** the receiver,
  - the **last bit is received** at t = RTT/2 + L/R = 0.015008 sec.,
  - assuming that ACK packets are extremely small (their **transmission time can be neglected**) the ACK emerges back at the sender at t = RTT + L/R = 0.030008 sec.
- In 0.030008 sec. of total transmission time, the sender was waiting almost all the time (99.973% of the time).

Transport Layer
Reliable Data Transfer: Performance of Pipelining

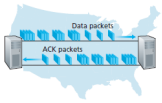
- **Pipelining**: instead of stop-and-wait, the sender is **allowed to send multiple packets** without waiting for acknowledgments.



- In pipelining approaches (e.g., 3 packets instead of 1):
  - the **sender begins sending the 3 packets** at t_0,
  - the **last bit of the last packet enters the channel** at t_0 + 0.000024 sec.,
  - the **packets take 0.015 sec. to reach** the receiver,
  - the **last bit is received** at t = RTT/2 + L/R = 0.015024 sec.,
  - assuming that ACK packets are extremely small (their **transmission time can be neglected**) the ACK emerges back at the sender at t = RTT + L/R = 0.030024 sec.
- In 0.030024 sec. **the sender was waiting 0.035% less** (99.920% instead of 99.973%).

Transport Layer
Reliable Data Transfer: Performance of Pipelining

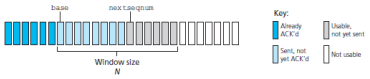
- **Pipelining**: instead of stop-and-wait, the sender is **allowed to send multiple packets** without waiting for acknowledgments.



- In pipelining approaches:
  - The **range of sequence numbers must be increased** since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.
  - We **need buffers to store incoming packets** since we don't know if packets are correct or there are "holes" in the transmission.
- There are 2 basic protocols using pipelining:
  - **Go-Back-N**.
  - **Selective Repeat**.

Transport Layer
Reliable Data Transfer: Go-Back-N

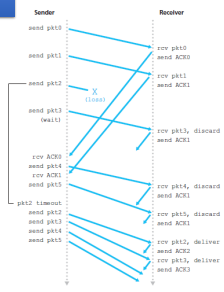
- **Go-Back-N (GBN)** protocol (aka sliding-window protocol): the sender is allowed to transmit multiple packets (when available) **without waiting for an acknowledgment** but is constrained to have **no more than N unacknowledged packets**.



- Base: is the **sequence number of the oldest unacknowledged packet**.
- Nextseqnum: is the **smallest unused sequence number** (next to be sent).
- We have that:
  - Packets in [0, base-1] have been **transmitted and acknowledged**.
  - Packets in [base, nextseqnum-1] have been **sent but not yet acknowledged**.
  - Packets in [nextseqnum, base+N-1] **can be sent**.
  - Packets in [base+N, +inf] **cannot be used** until a new acknowledgment is received.

Transport Layer
Reliable Data Transfer: Go-Back-N

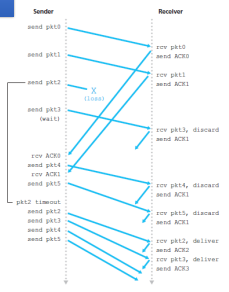
- What about the receiver? In GBN protocol **receiver is quite simple**.
- The receiver has **simply to discard out-of-order packets** (whether they are damaged or not) and to deliver to the upper level (application) in-order packets only.
  - Discarded packets (not acknowledged) will be eventually retransmitted by the sender.
- With this approach good packets are also discarded but the overall process is quite simple:
  - the **sender must maintain the indices of the window**,
  - the **receiver only needs to maintain the sequence number** of the next in-order packet.



Since pkt2 is lost, all pkt3, pkt4, pkt5, must be discarded even if correct.

Transport Layer
Reliable Data Transfer: Go-Back-N

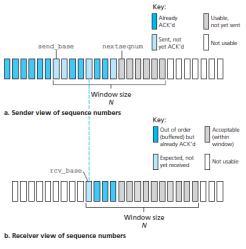
- Of course, the disadvantage of throwing away a correctly received packet is that **we need to resend it again**.
- This is a **chain-reaction**, if we are losing packets due to network difficulties many correctly received but out-of-order packets may be discarded forcing the sender to retransmit them.
- The retransmission itself might be lost or damaged and thus **even more retransmissions would be required**.



Not only pkt2, but also pkt3, pkt4, pkt5, must be transmitted again.

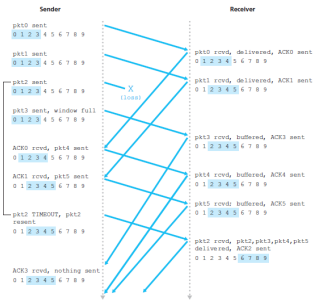
Transport Layer
Reliable Data Transfer: Selective Repeat

- Despite GBN is more effective than stop-and-wait, it may **suffer from performance problems**.
- Considering large window size**, a single packet error can cause GBN to retransmit a large number of packets**, many unnecessarily.
- In **selective-repeat** (SR) protocols: the sender retransmit **only those packets that were likely received in error** (lost or corrupted):
  - As in GBN **individual packets are acknowledged**.
  - A **window size of N is again used** to limit the number of unacknowledged packets.
  - Unlike GBN, **out-of-order packets are buffered** (stored) and acknowledged by the receiver.



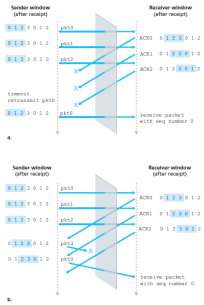
Transport Layer
Reliable Data Transfer: Selective Repeat

- Example of SR operation in the presence of lost packets: the receiver initially buffers pkt3, pkt4, and pkt5, while waiting for pkt2 (lost) to be retransmitted.
- Despite the previous example, here we avoid to resend pkt3, pkt4, pkt5, in so avoiding additional transmission loss or errors.



Transport Layer
Reliable Data Transfer: Selective Repeat

- One issue in SR is that the window size is related to the sequence number, **and sequence number is finite**.
- In this example we have four packets, **a max sequence number of 3** and a window size of 3.
- Let's assume packets 0 to 2 are correctly received and acknowledged, hence receiver's window moves to 6th packet (i.e., to packets 3, 0, 1):
  - **Case a**: the ACKs for the first three packets are lost and the sender retransmits these packets. *Is packet 0 new or old?*
  - **Case b**: the ACKs are received, packets 3 and 0 are sent but packet 3 is lost. *Is packet 0 new or old?*
- To avoid this issue, **the sequence number must be at least 2 times the window size**.



Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

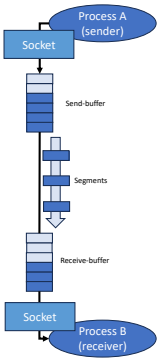
Riccardo Caccavale
(riccardo.caccavale@unina.it)

Transport Layer
TCP

- TCP with respect to UDP is:
  - **Connection-oriented**: there is a "handshake" phase before the transmission that ensures the two processes (sender and receiver) to be "connected".
  - **Reliable**: error detection, retransmissions, acknowledgments, timers, sequence numbers are implemented.
  - **Congestion and flow aware**: there is a regulation of the transmission rate depending on the receiver performance (flow) and the network performance (congestion).
- Connection orientation makes TCP full-duplex and point-to-point:
  - **Full-duplex**: if host A is connected with B then B is connected with A.
  - **Point-to-point**: single sender and single receiver, i.e., the transfer of data from one sender to many receivers is not possible.
    - Note: UDP allows multicasting, there are ways to transfer data to multiple hosts in a single send operation is not possible.

Transport Layer
TCP Buffers

- In TCP connection sending and receiving operations strongly **rely on buffers** to be performed.
- Buffers allow us to **partially decouple** transmission times from:
  - **Application delays**.
  - **OS delays** in multiplexing demultiplexing packets.
  - **Network delays** (oscillations of network performance).
- There is also the limit of the transmission rate, so long messages could **be broken into smaller segments** that may be collected in buffers (on the sender-side).



Transport Layer
Maximum Segment Size

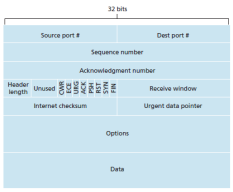
- The amount of **data inside a segment** is limited by the **Maximum Segment Size** (MSS):

$$MSS = MTU - header\_size$$

  - Where:
    - The **Maximum Transmission Unit** (MTU) is the maximum length of a frame acceptable by the link-layer (e.g., in Ethernet it is 1500 bytes).
    - The **combined header size** of TCP and IP headers (typically 20+20 bytes).
- **On sending**: the segments are created from the application data and passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams to be sent.
- **On receiving**: the data is placed into the receive-buffer, the application grabs data from the buffer when ready.

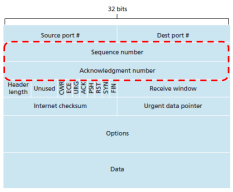
Transport Layer
TCP Segment

- The TCP segment consists of **header fields** and a **data field**, the latter contains some application data **of at most MSS size**.
- The **header** contains several fields:
  - **Sequence number** (32-bit) for reliable data transfer.
  - **Acknowledgment number** (32-bit) for reliable data transfer.
  - **Receive window** (16-bit) used to indicate the number of bytes that a receiver is willing to accept (for flow control).
  - **Header length** (4-bit) specifies the number of 32-bit words contained into the header. Note that **TCP header is variable due to the options** field.
  - **Options field** (K-bit) used for optional processes such as negotiating the MSS, time-stamping, etc.
  - **Flags** (6-bit) including:
    - ACK bit indicates that **this segment is an ACK** packet (so the acknowledgment number field is in use).
    - RST, SYN, and FIN bits used for **connection setup and teardown**.
    - CWR and ECE bits used in explicit congestion notification.
    - PSH bit tells the receiver to pass the data to the application immediately.
    - URG bit indicates that the segment contains "urgent" data.
  - **Checksum** (16-bit): for integrity check.
  - **Urgent data pointer field** (16-bit) indicates the location of the last byte of the urgent part of the data.



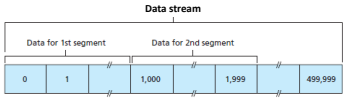
Transport Layer
TCP Sequence and Acknowledgment Numbers

- Sequence numbers and acknowledge numbers are **strictly related**.
- Sequence numbers are used not only for reliable data transfer but **also to manage segmentation**.
  - If a large data stream is transmitted from host A to host B, we need **to break it into smaller pieces** depending on the MSS.
- The acknowledgment number is calculated from sequence numbers. It is conventionally the **next piece of the data stream** we are expecting.



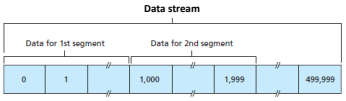
Transport Layer
TCP Sequence and Acknowledgment Numbers

- Notice that TCP is a general-purpose transport protocol, it has no information about the data to be transmitted. From TCP's standpoint data **is just an ordered stream of bytes**.
- The **sequence number** for a segment is then the byte-stream **number of the first byte of the data** in the segment.



- In the example, assuming a **data stream of 500000 bytes** (~500KB), and a **MSS of 1000 bytes**.
- The TCP constructs **500 segments** where the first segment has sequence number 0, the second segment has sequence number 1000, the third segment has sequence number 2000, and so on.

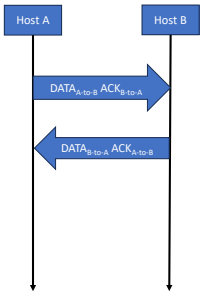
Transport Layer
TCP Sequence and Acknowledgment Numbers



- Let's assume this data stream to be **passed from host A to host B**.
- For simplicity, let's **neglect previous interaction** (data exchange from connection establishment), so **we are starting from sequence number 0**:
  - The receiver will get a first segment of 1000 bytes, i.e., from **sequence number 0 to sequence number 999**.
  - The receiver will then acknowledge the transmission by setting an **acknowledgment number of 1000** (next expected byte in the stream).
  - The receiver will get the next segment of 1000 bytes, i.e., from **sequence number 1000 to sequence number 1999** and so on…

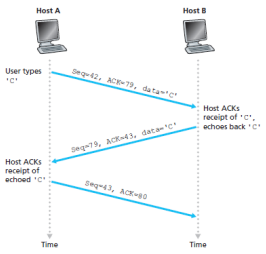
Transport Layer
TCP Sequence and Acknowledgment Numbers

- What happens if the **receiver is sending data in return**?
  - Remind: TCP communication is **full-duplex**, if 2 hosts, A and B, communicate there are also 2 flows of data to be considered for reliable data transfer: A-to-B and B-to-A.
- In this case we can **use sequence numbers and acknowledgment numbers at the same time**:
  - Segments from A have **sequence** numbers related to **A-to-B flow** and **acknowledge** numbers related to the **B-to-A flow**.
  - Segments from B have **sequence** numbers related to **B-to-A flow**, and **acknowledge** numbers related to the **B-to-A flow**.



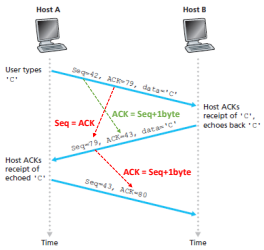
Transport Layer
TCP Sequence and Acknowledgment Numbers

- This is a simplified example of two hosts sending data each other.
  - We are considering 2 messages of **just 1 byte** (1 char) form A to B and vice versa.
  - Specifically, host A transmits a 'c' **that is echoed back** by B.
- Here that segments provide **ACK and DATA at the same time** (so the ACK flag is 1).



Transport Layer
TCP Sequence and Acknowledgment Numbers

1. From A to B:
   - The **ack number as the seq number of the next expected packet** in the B-to-A flow (byte 79).
   - The **seq number** as the position of **the single byte** we are transmitting (byte 42).
2. From B to A:
   - The **ack number as the seq number of the next expected packet** in the A-to-B flow (i.e., seq + 1 byte of the 'c' char).
   - The **seq number** as the position of **the byte we are transmitting** (byte 79).
3. From A to B:
   - Pure **ACK segment** is sent, having seq number 43 (as expected by B) and ack number 80 (next expected).



A-to-B info are in green.
B-to-A info are in red.

Transport Layer
Retransmission Timeout

- Previously we have emphasized that **TCP reliable data transfer mechanism makes extensive use of timeouts**.
- Timeouts can be used **in combination with ACKs** to understand (or at least to guess) if segments have been lost and should be retransmitted.
  - Notice that **wrong ACK can be received for different reasons** (not only when lost).
  - For example, if **segments arrives in the wrong order** at the destination a different ACK is sent.
- ACK management can be tricky. The basic approach followed by pipelining methods (sender-side) is to **ignore wrongly received ACKs**:
  - Retransmission is performed **only when timeout is elapsed**.
  - Timeout estimation is **critical**.

Transport Layer
RTT Estimation

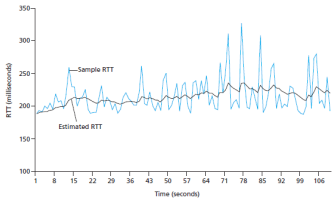
- To suitably se timeouts **we need a round-trip time (RTT) estimation**.
- The timeout, i.e., the time to wait for a segment's ACK, should be **larger than RTT** otherwise unnecessary retransmissions would be sent.
- TCP estimates this value by **sampling the RTT of successfully acknowledged segments** that have not been retransmitted (one-shot).
- Since the time of a sampled RTT ($samRTT$) **may fluctuate** (due to traffic congestion, load of the receiver, etc.) the RTT estimation ($estRTT$) is given by the **exponential weighted moving average (EWMA)**:

$$estRTT_t = (1-\alpha)estRTT_{t-1} + \alpha\, samRTT_t$$

Where α is a parameter which is usually *0.125* (i.e., *1/8*).

Transport Layer
RTT Estimation

- This is an example of how RRT (sampled and estimated) behaves in a **real scenario** (between USA and France).
- It is also useful to measure the **variability of the RTT** ($devRTT$) from the difference between the sample ($samRTT$) and the estimation ($estRTT$):



$$devRTT_t = (1-\beta)devRTT_{t-1} + \beta|samRTT_t - estRTT_t|$$

Where β is a parameter which is usually *0.25* (i.e., *1/4*).

Transport Layer
Retransmission Timeout

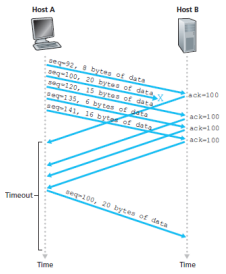
- Having this estimations, **it is possible to define a retransmission timeout** for TCP that is not lower nor too higher than the estimated RTT:

$$timeout_t = estRTT_t + 4devRTT_t$$

- Here, the deviation of the RTT is used to set a **reasonable and adaptive margin** from the estimated RTT.
  - it increases when RTT oscillates, so the timeout window is larger **when we are uncertain** about our current RTT.
- By default, the **initial value** of the RTT (at t=0) is 1 second.

Transport Layer
Fast Retransmit

- One of the problems with timeout-triggered retransmissions is that the **timeout period can be relatively long**.
- To limit this issue TCP applies a mechanism called **fast retransmit** that uses **wrong ACK**:
  - When a TCP receiver receives a segment with a **sequence number that is larger than the expected one** it means that a segment has been reasonably missed.
  - The receiver **resends the old ACK** (duplicate ACK) having ack number of the expected segment.
  - If the **sender receives N duplicates** (typically 3 duplicates), it assumes that the previous segment has been lost so it is (fast) retransmitted **well before the deadline**.



Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Transport Layer
TCP Problems in Connection Management

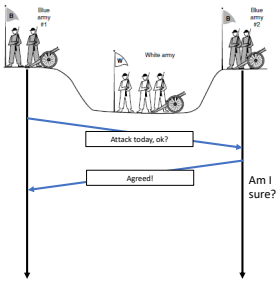
- Since TCP is connection-oriented **two hosts must agree** during both the **opening** and the **closing** procedures.
- Knowing that messages could be **lost or damaged** on the network, **it is quite difficult** for two hosts to find such agreement.
- If messages are lost during transmission **one end of the communication can be open or closed while the other is not**.
- To avoid (or better to mitigate) this issue TCP implements a procedure called **three-way handshake** in which open/close connection requests **have to be acknowledged** by hosts before a connection can be established/released.

Transport Layer
TCP Problems in Connection Management

- The two-army problem:
  - Imagine a **white army encamped in a valley** and, on both hillsides, there **are 2 enemy blue armies**.
  - The **white army is larger than either of the blue armies** alone, but together the blue armies are larger, they will be victorious only if the attack is simultaneous.
  - To synchronize their attacks, **blue armies must send messengers through the valley** where they might be captured (unreliable communication).
  - Does a protocol exist that allows the blue armies to win?

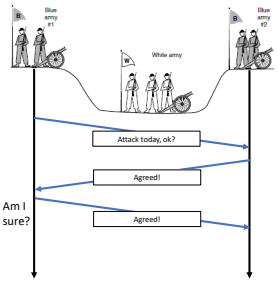
Transport Layer
TCP Problems in Connection Management

- Suppose that the commander of **blue army #1** sends a message reading: "I propose we attack today, is it ok?"
- Now suppose that **the message arrives**, the commander of blue army #2 agrees, and his **reply gets safely back** to blue army #1.
- Will the attack happen? Probably not, because **commander #2 does not know if his reply got through**. If it did not, blue army #1 will not attack.



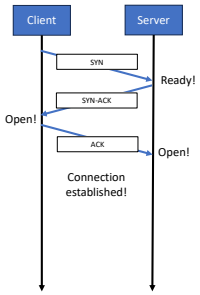
Transport Layer
TCP Problems in Connection Management

- Let's make it a **three-way handshake**: the first commander (of the original proposal) **must acknowledge** the response.
- Assuming **no messages are lost**, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate (he does not know if his acknowledgement got through).
- Now **we could make it a four-way handshake**, but that does not help either. In fact, **it can be proven that no protocol exists that works**.
  - **Three-way handshake** is not perfect, but it is usually adequate!



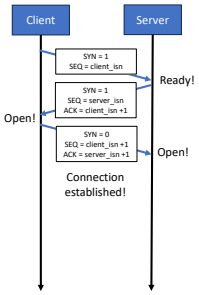
Transport Layer
TCP Connection Establishment

- In TCP **three-way handshake** is performed to establish connection:
  - Client send a **connection request** (SYN segment) to the server.
  - Server responds with a **special acknowledgment** (SYN-ACK segment).
  - Client sends back a **final acknowledgment** (ACK segment).
- TCP connection establishment is a **delicate procedure** that can also add significantly delays.
  - There is typically a **timeout** (30-60 secs) to complete the handshake, after that the procedure is aborted.
  - Some **attacks** (e.g., SYN flood) happen here.



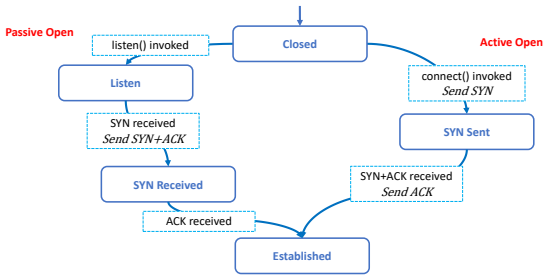
Transport Layer
TCP Connection Establishment

- Details of the **three-way handshake** procedure:
  1. The client sends a **SYN segment** having:
     - No application-layer data (payload).
     - The SYN bit set to 1.
     - A random initial sequence number (client_isn) as sequence number.
  2. When the above SYN segment (hopefully) arrives, the **server allocates TCP buffers and variables** and sends back a **SYNACK segment** having:
     - No application-layer data (payload).
     - The SYN bit set to 1.
     - The acknowledgment number set to client_isn+1.
     - A random initial sequence number (server_isn) as sequence number.
  3. When the SYNACK segment (hopefully) arrives, the **client also allocates buffers and variables** to the connection and sends to the server a final **ACK segment** having:
     - Possibly application-layer data.
     - The SYN bit set to 0 (connection is established).
     - The acknowledgment number set to server_isn+1.



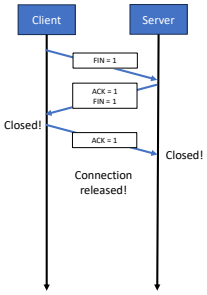
Transport Layer
TCP Connection Establishment

- Simplified state diagram for TCP connection establishment.



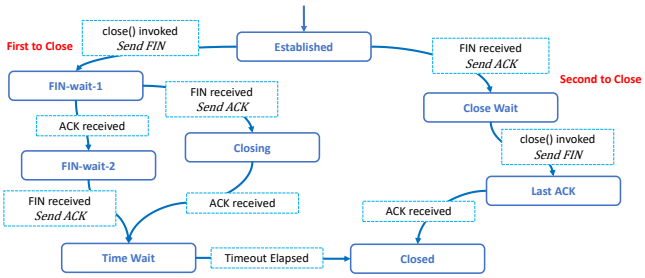
Transport Layer
TCP Connection Release

- TCP connection release (aka **teardown**) can be performed by either hosts.
- Let's assume client is closing the connection, the **three-way handshake** is performed as follow:
  1. The client sends a **special shutdown segment** (FIN segment) to the server having the FIN bit set to 1.
  2. When the server receives this segment, it **sends back an acknowledgment/shutdown segment**, having ACK to 1 and FIN bit set to 1.
     - Note: ACK and FIN can be sent in the same segment or in two separated ones.
  3. Finally, the client **acknowledges** the server's shutdown segment.



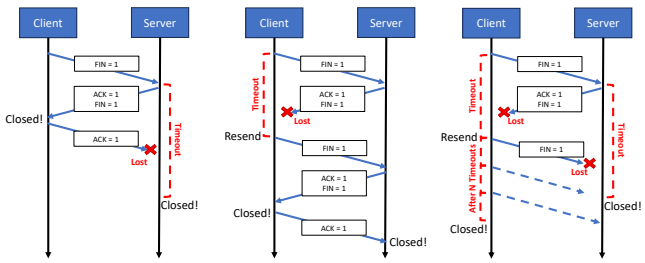
Transport Layer
TCP Connection Release

- Simplified state diagram for TCP connection release.



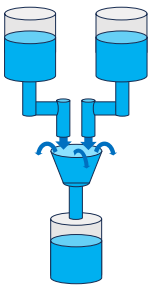
Transport Layer
TCP Connection Release

- How do we save ourselves from packet loss?
- Since three-way handshake is not perfect, TCP rely on timers to eventually close connections or to send again requests. Here some examples during connection release:



Transport Layer
Congestion and Flow Control Problem

- Additional features of TCP with respect to UDP are the **congestion and flow control**.
- We mentioned that packet loss is **often a result of buffers' overflow**:
  - From receiver buffer, if **receiving application is not fast enough** in reading the data.
  - From network devices (e.g., routers), if **nodes are congested**.
- Following our previous water-flowing analogy, **we can see buffers as intermediate buckets** that overflows in case water exceeds.
- If packets are lost, **hosts are forced to rely on retransmission and timeouts** that drastically impair the network performance.



Transport Layer
Flow Control

- When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer. The receiving application **will read data from this buffer**, but **not necessarily at the instant the data arrives** (application may be busy).
- If the application is relatively **slow at reading the data**, the sender can very easily **overflow the receiver buffer** by sending too much data too quickly.
- TCP **flow control is a speed-matching service**: it attempts to match (reduce) the rate at which the sender is sending against the rate at which the receiving application is reading.
  - Remind: TCP segment **allow receiver to tell the sender** how much buffer has left through the **receive window field** (free buffer space).

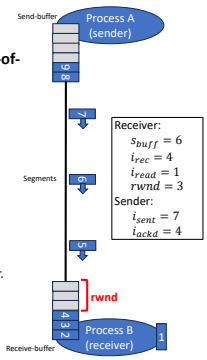
Transport Layer
Flow Control

- Let's assume for simplicity that the **TCP receiver discards out-of-order segments**, so all segments in the buffer are ordered.
- On the **receiver side** (host B) we have:
  - s_{buff} size of the receiver buffer (in bytes)
  - i_{read} the last byte of the stream retrieved by the application.
  - i_{rec} the last byte of the stream that is received.
  - We may define the size of the **receive window** ($rwnd$) as:

$$rwnd = s_{buff} - (i_{rec} - i_{read})$$

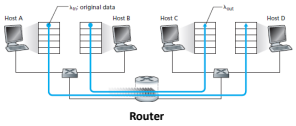
- On the **sender side** (host A) we have:
  - i_{sent} the last byte of the stream to be sent.
  - i_{ackd} the last byte of the stream that is acknowledged by the receiver.
  - Knowing the receive window, the **sender guarantees anytime** that:

$$i_{sent} - i_{ackd} \leq rwnd$$



Transport Layer
Congestion Problem

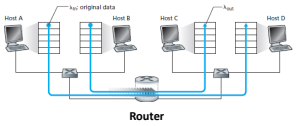
- The **congestion control problem** is similar to flow control, but it is related to the network infrastructure.
- Example: two hosts (A and B) have a connection that **shares a single router** and a single outgoing link of capacity R between source and destination.



- The **router has buffers** that allow it to store incoming packets when the packet-arrival rate exceeds the outgoing link's capacity.
- Let's assume for simplicity that the both applications (in A and B) are sending data into the connection at **the same average rate** of λ_{in} bytes/sec.

Transport Layer
Congestion Problem

- If $\lambda_{in} \leq R/2$, everything sent by the sender is received by the receiver **with a finite delay**.
- If $\lambda_{in} > R/2$, **the link reaches its full capacity** (R), and the exceeding packets are stored into the buffer.



- As long as we exceed the maximum capacity the **packets will be accumulating into the router's buffer** waiting for their turn to be sent into the link.
- Since **buffer is finite**, accumulated **packets will eventually be discarded** and hosts will be forced to retransmit them (even more traffic).
- **Neither an infinite buffer works**, packets will not be lost but **the delay would constantly increase**: if we exceed the capacity forever, the delay will reach infinity (so packets are as good as lost).

Transport Layer
Congestion Control

- There are **mainly two approaches** to congestion control:
  1. **End-to-end congestion control**: this is the standard TCP approach where **congestion is inferred by the end systems** based only on observed network behavior (for example, packet loss and delay).
     - TCP segment loss (due to timeouts or the receipt of 3 duplicate ACKs) is taken as an indication of network congestion, and TCP decreases sending rate accordingly.
  2. **Network-assisted congestion control**: this is a recent (optional) approach where transport-layer works in synergy with network-layer. **Routers provide explicit feedback** to the sender and/or receiver regarding the congestion state of the network.
     - The feedback may be as simple as a **single bit indicating congestion** at a link, but more sophisticated feedback is also possible.

Transport Layer
TCP Congestion Control

- **TCP mostly relies on end-to-end congestion control**.
- The approach is to have **each sender to adapt their sending rate** depending on the **perceived network congestion**.
- Here there are 3 main problems to be considered:
  - **Rate regulation**: how does a TCP sender regulate the rate at which it sends traffic into its connection?
  - **Congestion detection**: how does a TCP sender perceive the congestion on the path between itself and the destination?
  - **Rate adjustment**: what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Transport Layer
TCP Congestion Control – Rate Regulation

- Reminder: in TCP flow control **the sending rate is regulated by considering the buffer space** on the receiver side (receiver window):

$$i_{sent} - i_{ackd} \leq rwnd$$

- In TCP congestion-control the sender **also keeps track of a congestion window** ($cwnd$):

$$i_{sent} - i_{ackd} \leq \min(rwnd, cwnd)$$

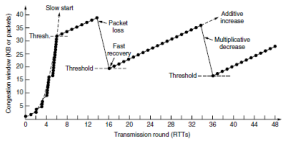
- So, the rate is regulated by increasing/decreasing $cwnd$.

Transport Layer
TCP Congestion Control – Congestion Detection

- A TCP sender perceives that there is congestion on the path between itself and the destination in two ways by checking ***loss events***.
- A ***loss event*** happens **when either a timeout is reached or 3 duplicate ACKs are received**.
  - Both events mean that a previous packet **is not arrived at the destination** probably because of network devices overflow.
- If **loss events do not occur**, i.e., ACKs of segments arrive as expected, TCP will assume that network is not congested, so the **congestion window can be increased**.
- If, on the other hand, **loss events occur**, the **congestion window must be decreased**.

Transport Layer
TCP Congestion Control – Rate Adjustment

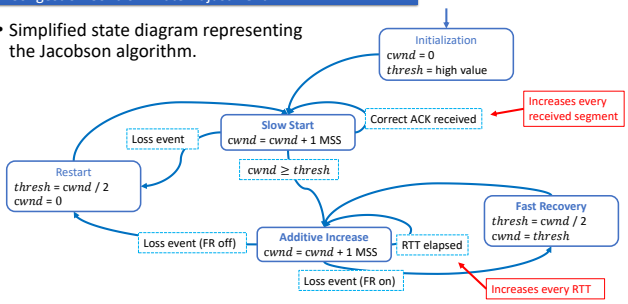
- TCP rate adjustment is then performed through **bandwidth probing**: rate is increased as long as ACKs arrive correctly (probing the network), when congestion is detected (loss events) the rate is decreased. This process is continuously repeated.
- TCP uses **Jacobsen congestion-control algorithm [Jacobson 1988] to regulate the rate**, it includes 3 phases:
  1. **Slow start**: start from 1 MSS/RRT increase rate exponentially.
  2. **Additive Increase** (or congestion avoidance): increase rate linearly.
  3. **Fast Recovery** (optional): halves the rate instead of slow-starting again and proceed with additive increment.



Typical sawtooth behavior of the Jacobsen algorithm

Transport Layer
TCP Congestion Control – Rate Adjustment

- Simplified state diagram representing the Jacobson algorithm.



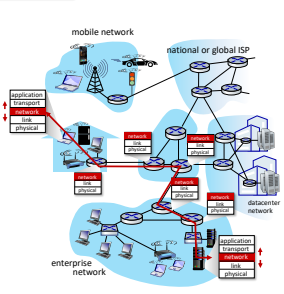
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Network Layer
From Transport to Network

- The job of the network layer:
  - At the **sending host**: to take segments from the transport layer, **to encapsulate each segment into a datagram**, and to send the datagrams into the network.
  - At the **receiving host**: to receive the datagrams from the network, **extracts the transport-layer segments from datagrams**, and delivers the segments up to the transport layer.
- Inside the network there are **nodes that forward datagrams to the adjacent nodes** (routers) up to the destination host.
  - Note: the routers are shown with a **truncated protocol stack**. Potentially routers do not run applications nor transport-layer protocols.



Network Layer
Network Layer in Internet

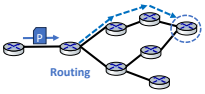
- The network layer is mainly responsible for **host-to-host delivery**.
- What **services could be offered** along with delivery:
  - *Guaranteed delivery*: a packet sent by a source host will eventually arrive at the destination host.
  - *Guaranteed delivery with bounded delay*: packets will be delivered and within a specified host-to-host delay bound (for example, within 100 msec).
  - *In-order packet delivery*: packets will arrive at the destination in the order that they were sent.
  - *Guaranteed minimal bandwidth*: possibility to specify a minimal bit rate (for example, 1 Mbps) such that, if the rate of the sending host is within it, then all packets are eventually delivered to the destination host.
  - *Security*: encryption/decryption of all datagrams at the source/destination.
- Actually, **none of these services** is offered by networks.

Network Layer
Network Layer in Internet

- Conversely, Internet's network layer provides **just one service**, the so called **best-effort service**.
- **Best-effort service** (or best-effort delivery): the network **tries its best** to deliver a packet from its source to its destination.
  - Packets are neither guaranteed to be received in the order in which they were sent, nor to be received at all.
  - There is no guarantee about delays or minimal bandwidth.
- Despite its simplicity, the best-effort service model, in combination with good bandwidth **have proven to be adequate**.
  - For example, applications, such as Netflix and voice-and-video-over-IP, real-time conferencing, Skype, etc. all works with it.

Network Layer
Routers in the Network

- The primary role of the network layer **is to perform host-to-host delivery**, i.e., to move packets from a sending host to a receiving host.
- This process is performed by **routers** (network nodes) that are **special nodes having multiple incoming/outcoming links**. Routers provide two network-layer functions:
  - **Forwarding**: when a packet arrives at a router's input link, the router **must move the packet to the appropriate output** link. It is also possible to:
    a. **Block a packet** from exiting a router (e.g., if the packet originated at a known malicious sending host, or if the packet were destined to a forbidden destination host).
    b. **Duplicate a packet** and sent it over multiple outgoing links.
  - **Routing**: to **decide the route or path to be taken** by packets as they flow from a sender to a receiver. The algorithms that calculate these paths are referred to as **routing algorithms**.



Network Layer
Routers

- Routers may be very **different** depending on their functions:
  - Home or business usage.
  - Wireless or wired connections.
  - **Edge routers**: a router that distributes data packets between one or more networks (e.g., connecting a network with the ISP).
  - **Core routers**: used to distribute packets within the same network rather than across multiple networks.
    - These are also used **on the backbone of the Internet** and its job is to carry out heavy data transfers.


Cisco RV016-G5 business router


Tp-link AX6600 home router


Juniper MX2020, edge router

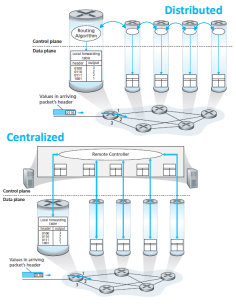

Cisco CRS-1 backbone core router

Network Layer
Routers

- **Forwarding** (data plane) refers to the **router-local** action of transferring a packet from an input link interface to the appropriate output link interface.
  - This is a **fast operation** (typically a few nanoseconds), and thus is typically implemented in hardware.
- **Routing** (control plane) refers to the **network-wide** process that determines the end-to-end paths that packets take from source to destination.
  - This is a **slower process** (typically seconds), and it is often implemented via software.
- At the data plane we have the **forwarding table**: a table that specifies to which output link a packet should be forwarded in order to reach the destination.
  - A router forwards a packet by **examining the value of one or more fields** in the header of arriving packets.
  - The value stored in the forwarding table entry indicates **the outgoing link interface** to which that packet is to be forwarded.

Network Layer
Routers

- Since multiple routers can be encountered before to reach a destination, the **content of a forwarding table** must be determined **by collecting information from different routers**.
- This functionality can be performed in two ways:
  - **Decentralized** (or distributed): we can have each router endowed with a routing component that communicates with the routing component of other routers (this was the mainstream approach).
  - **Centralized**: we can have a physically separated (from the routers) remote controller computes that distributes the forwarding tables to be used by routers.



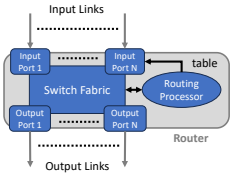
Network Layer
Routers

- In the second case, the remote controller might be implemented in a **remote data center** (with high reliability and redundancy) and might be managed by the ISP or some third party.

- This approach is at the basis of **software-defined networking** (SDN), where the network is "software-defined" because the controller that computes forwarding tables and interacts with routers is implemented as a software.
  - Some of these software are also open.

Network Layer
Routers: Main Components

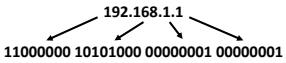
- Main router's **components** are:
  - **Input ports** (different from transport-layer ports): are the **physical input interfaces** that operate with the lower link-layer of the connected link, their job is:
    - To consult the forwarding table (aka **lookup**) and **to prepare the switching factory** for the output port to choose.
    - To **forward control packets** (e.g., packets carrying routing information) to the routing processor.
    - The number of input **ports may range from dozens to hundreds** (e.g., the Juniper MX2020 edge router supports up to 960 10 Gbps Ethernet ports).
  - **Switching fabric**: connects input ports to its output ports.
  - **Output ports**: stores packets received from the switching fabric and transmits these packets on the outgoing link.
    - **Ports are often bidirectional**, i.e., output port is coupled with an input port.
  - **Routing processor**: that executes the routing protocols, maintains state-information about links, and computes the forwarding table.



Network Layer
Routers: Forwarding

- The **role of the forwarding table is to associate IP addresses to output ports**, so that packets can be forwarded to the right output link in order to be transmitted to the next node (and possibly be forwarded again).

- An **IP address is a 4-bytes (32-bits) number** that we usually see in decimal notation, but we can also see it in its binary form:



- Inside a forwarding table, **IPs are associated to output port numbers** for actual redirection.

Network Layer
Routers: Forwarding

- When a packet is received from the link into an input port **a lookup operation** is performed.

- The **port has a copy of the forwarding table** from the routing processor (received on a dedicated bus, e.g., a PCI bus) to avoid bottleneck due to continuous invoking of the centralized routing processor on a per-packet basis.

Example of a 4-ports router's table

Prefix of IP addresses	Link Interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

11001000 00010111 00010000 00000000 TO 11001000 00010111 00010111 11111111

11001000 00010111 00011000 00000000 TO 11001000 00010111 00011000 11111111

11001000 00010111 00011001 00000000 TO 11001000 00010111 00011111 11111111

Network Layer
Routers: Forwarding

Example of a 4-ports router's table

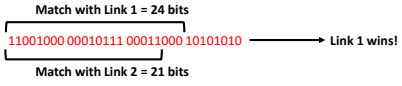
Prefix of IP addresses	Link Interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

11001000 00010111 00010000 00000000 TO 11001000 00010111 00010111 11111111

11001000 00010111 00011000 00000000 TO 11001000 00010111 00011000 11111111

11001000 00010111 00011001 00000000 TO 11001000 00010111 00011111 11111111

- Notice that entries may not be mutually exclusive (for example, IP 11001000 00010111 00011000 10101010 matches links 1 and 2) if so, the router forwards it to the longest matching entry (**longest prefix matching rule**):



Network Layer
Routers: Forwarding

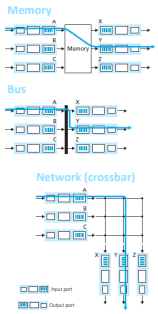
- This **lookup procedure is typically performed on the hardware** to be executed as fast as possible (e.g., in nanoseconds for Gigabit transmissions).
  - Embedded memories and advanced table-search algorithms are also involved.

- Once a packet's output port has been determined (after lookup), **the packet can be sent into the switching fabric**. In some devices, packets may also be temporarily queued (buffered) if other input ports are using the fabric.

- This two-steps operation of **looking up a destination IP address (match)** and **forwarding (action)** is called **match-plus-action** and is performed in many networked devices, such as:
  - **Switches**: similar action as routers.
  - **Firewalls**: where **the action is to filter out specific incoming packets**.
  - **Network address translators** (NATs): where the action is **to rewrite port number** of specific incoming packets before forwarding.

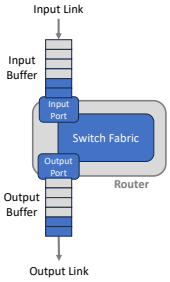
Network Layer
Routers: Switching Fabric

- In a switching fabric, switching can be performed in 3 ways:
  1. **Switching via memory**: ports are **considered as I/O devices** that write packets on memory cells then the **routing process copies the message** on the output port as specified by the forwarding table.
     - This approach is a bit slow (need for memory access) and was more common in early routers (which were standard computers).
  2. **Switching via a bus**: input port transfers a packet directly to the output port **over a shared bus**, without intervention by the routing processor.
     - Only one port per time can be served, but this method is often sufficient for routers that operate in small local area.
  3. **Switching via an interconnection network**: input/output ports **connected by a network** having cross-points which can be open/close and then redirect the packets.
     - Here multiple packets can be forwarded in parallel. This method is used in several modern routers.



Network Layer
Routers: Ports

- Since **switching takes time**, input and output **ports have queues to temporarily store packets** (as cars waiting for semaphores).
- The **extent of queueing is not fixed**, it may depend on traffic load, the speed of the switching fabric, or the line speed.
- In general, **packets may be received/sent faster/slower than switching**, so they may be accumulating into the input/output buffer (that may overflow).



Network Layer
Routers: Ports

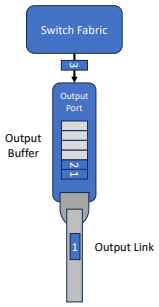
- Example: let's assume to have (1) **a router with N input and N output ports**, (2) that **each port is receiving packets** at the same time, and (3) that all input and output lines **have the same speed** of R_{line} packets per second.
- Let's now consider a **worst-case scenario** in which **all packets have to be forwarded to the same port**.
- If the **switching fabric** have a rate (R_{switch}) we have:
  - If $R_{switch} \cong NR_{line}$, **queuing on input ports is negligible** as all packets are forwarded in time by the forward fabric, but **queuing on the output port is significant** as the incoming packets are N time more than the R_{line} rate of the output link.
  - if $R_{line} < R_{switch} < NR_{line}$, there is **queuing on both ports** as input port must wait for the switch fabric while output port must wait for the link.
  - if $R_{switch} \cong R_{line}$, there is **negligible queueing on output** port but **significant queuing on the input** ports.

Network Layer
Routers: Packet Scheduling

- It is reasonable to have **multiple packets** (potentially from multiple input ports) to be forwarded **to a single output port**.
- The access of queued packets from buffer to the output link needs to be **scheduled**.
- There are basically 3 (famous) approaches:
  - **First-come-first-served** (FCFS, aka **first-in-first-out**, FIFO), which is **simple time-based approach**.
  - **Priority queuing**, which is based on the importance of the packets.
  - **Round-robin queueing**, where are divided into classes (based on priority) and each class is served in turn.

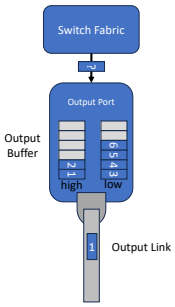
Network Layer
Routers: Packet Scheduling - FIFO

- If the **output link is busy** (transmitting something) the packets arriving at the output port must be **buffered**.
- If **there is no sufficient buffering space** to hold the arriving packet, we must rely on a **packet-discarding policy**.
  - A typical policy is to **drop the recently arrived packets (drop-tail)** but in more sophisticated approaches **also already buffered packets can be removed** to make space for the arriving ones.
- A packet is removed from the queue **only if it has been completely transmitted** over the outgoing link (served).
- In **FIFO scheduling** packets are selected for transmission **in the same order** in which they have arrived at the output port.



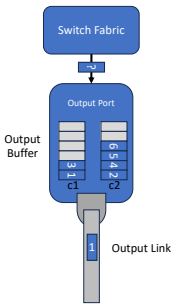
Network Layer
Routers: Packet Scheduling - Priority

- In **priority queuing**, packets arriving at the output link are classified (e.g., through TCP/UDP port numbers) into priority classes upon arrival at the queue.
- A **network operator may configure a queue** so that specific packets (e.g., carrying network management information, real-time voice-over-IP, etc.) may receive priority over user traffic or non-real-time packets.
- **Each priority class typically has its own queue:**
  - Packets from the most prioritized non-empty class **are transmitted first**.
  - The choice among packets in the same priority class is typically done **in a FIFO manner**.



Network Layer
Routers: Packet Scheduling – Round-robin

- In **round robin queuing**, packets are still sorted into classes, but **classes are alternated** rather than selected by priority.
- A common implementation is called ***weighted fair queuing*** (WFQ) where arriving packets are classified and queued in the appropriate waiting area.
- Each class is then associated to a specific weight that dictates the **rate with which the class is selected** over the others.



Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

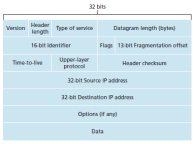
Riccardo Caccavale
(riccardo.caccavale@unina.it)

Network Layer
Internet Protocol (IP)

- The **Internet Protocol (IP)** is the network-layer protocol used to ensure host-to-host delivery. There are 2 versions of IP currently in use:
  - **IP version 4 (IPv4)** which is the most used and the most common.
  - **IP version 6 (IPv6)** which is the newer version, proposed to replace IPv4.
- The **most important functionality** provided by the IP is **to identify hosts** on a network (IP addressing).
- The **IP addressing** is the process of assigning IP addresses to devices. Addressing is **crucial** for network functionalities and is quite **complex** in Internet (there are millions or billions of hosts to be addressed).

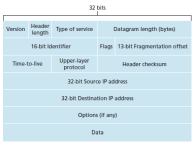
Network Layer
Internet Protocol: Datagram

- Internet's network-layer packet called **datagram** (as UDP). The key fields in the IPv4 datagram are:
  - **Version number** (4 bits) specify **the IP version** (e.g., v4 or v6) of the datagram (formats depends on versions).
  - **Header length** (4 bits) **dimension of the header** (not fixed, options are of variable size), used to know where the payload starts (no options means 20 bytes header).
  - **Type of service** (TOS, 8 bits) identifies specific **proprieties** of the datagram (e.g., real-time/non-real-time). Such types are **defined by network administrators** of routers.
  - **Datagram length** (16 bits) length in **bytes of the datagram** (header + data). Datagrams are rarely larger than 1500 bytes (out of 65535 max).
  - **Identifier** (16 bits) a progressive number that uniquely identifies a datagram (used in fragmentation).
  - **Flags and fragmentation offset** (3+13 bits) proprieties and offset of the fragment (used in fragmentation).
  - **Time-to-live** (TTL, 8 bits) ensure that datagrams do **not circulate forever in the network**. This field is decremented by one each time the datagram is processed by a router. If the TTL field reaches 0, a router must drop that datagram.



Network Layer
Internet Protocol: Datagram

- **Upper-layer Protocol** (8 bits) indicating the **transport-layer protocol** to which the payload (data filed) of this IP datagram should be passed (e.g., 6 for TCP, 17 for UDP).
  - Complete list of protocols available from IANA website: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml
- **Header checksum** (16 bits) for error detection. Here checksum is calculated as 1s complement of the 2 bytes-wise sum of the header. **Routers check this value** and **erroneous datagrams are typically discarded**.
  - Note that the checksum must be **recomputed and stored again at each router** due to the TTL and options fields that can change.
- **Source and destination IP addresses** (32+32 bits).
- **Options** (not fixed) allow an IP header to be extended (**additional functionalities**). Option field provide a certain degree of complexity (unknown size), it is not present in IPv6.
- **Data** (not fixed) contains the **actual message (payload)** typically in the form of a TCP/UDP transport-layer segment.

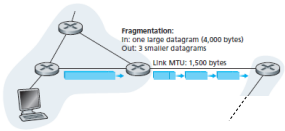


Network Layer
Internet Protocol: Fragmentation

- The first problem with IP datagrams is that they can be **fragmented at the link level**. Some link-layer protocols carries datagrams of **different dimensions**.
  - Example: Ethernet frames can carry up to 1500 bytes of data.
- IP datagrams are eventually encapsulated into link-layer frames to be transported from node to node. The maximum amount of data (i.e., the maximum frame length) that a link-layer frame can carry is the **maximum transmission unit** (MTU).
- A router that interconnects 2 links having different MTUs **may receive an IP datagram from input link that does not fit the output link**.
- The **router has to divide the payload** of the IP datagram into two or more smaller IP datagrams (fragments). **Fragments are then encapsulated into link-layer frames** and forwarded over the outgoing link.

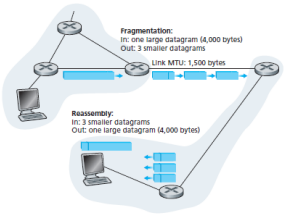
Network Layer
Internet Protocol: Fragmentation

- When a router **fragments a datagram**:
  - It **copies the same identifier, source address, and destination address** into the newer fragments.
  - It sets the **fragment offset** field of all fragments **to progressive numbers**.
  - It sets the **flag of the last fragment to 1** (to signal that the fragments are over).
- Clearly, **fragments need to be reassembled** before they reach the transport layer at the destination since both TCP and UDP are expecting to receive complete segments from the network layer…



Network Layer
Internet Protocol: Fragmentation

- The designers of IPv4 felt that **reassembling datagrams in the routers** would introduce significant complication into the protocol and **put a damper on router performance**.
- In IPv4 datagram **reassembly is then performed into the end systems**:
  - If multiple datagrams having **same addresses and identifier** are received, it means that the original datagram has been fragmented.
  - The host has to **recreate the original datagram** from the fragments.



Network Layer
Internet Protocol: Addressing

- IP addresses are typically written in so-called **dotted-decimal notation**, in which each byte of the address is written in its decimal form and is separated by a period (dot) from other bytes in the address.
- For example:

	Dotted-decimal	Binary
IP address	193.32.216.9	11000001 00100000 11011000 00001001

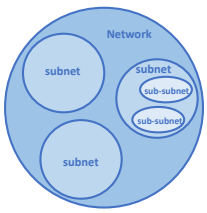
- Each device in the global Internet must have (somehow) an **IP address that is globally unique**.
- Since each **IP address is 32 bits (4 bytes) long** (equivalently, 4 bytes), there are a total of **2^{32} (around 4 billion) possible IP addresses**.

Network Layer
Internet Protocol: Addressing

- Hosts and, in general, **network devices are connected through links** (wired or wireless).
- A host typically has only a single link into the network, while a network device (e.g., a router) **may have multiple links**.
- The boundary between the host and the physical link is called **interface**.
- Because every host and router is capable of sending and receiving IP datagrams, IP requires **each interface to have its own IP address**.
- Thus, an **IP address is technically associated to the interface**, rather than the host or router containing that interface.

Network Layer
Internet Protocol: Addressing

- Assigning IP addresses to different interfaces is **not trivial**. It would be **unwise to randomly assign IP** addresses for several reasons, for example:
  - As IPs give no indication about locations, **we wouldn't know where to find hosts**.
  - We would need for **huge forwarding tables** inside routers.
- The network addressing resembles the one of *standard telephony*: **networks are hierarchically divided into sub-networks** (or **subnets**) having different prefixes.
- An IP address is then divided in **two parts**:
  - The **first portion** (leftmost) identifies the subnet to which the node is connected.
  - The **second portion** (rightmost) identifies the single interface.
- The **number of bits belonging to each portion is not fixed**.



Network Layer
Internet Protocol: Addressing

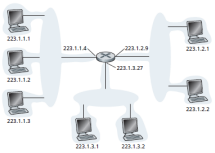
- Specifically, to distinguish the subnet-part from the interface-part, an IP address is associated to a **subnet mask** that specifies which bit of the address belongs to the subnet-part.
- For example:

	Dotted-decimal	Binary
IP address	193.32.216.9	11000001 00100000 11011000 00001001
Subnet Mask	255.255.255.0	11111111 11111111 11111111 00000000

- Another common way to represent the masks is the ***slashed notation***, for instance 193.32.216.0/24 represents the IP of the subnet, where the /24 indicates that the leftmost 24 bits of the address are dedicated to the subnet.

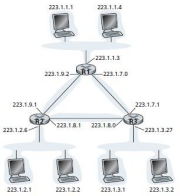
Network Layer
Internet Protocol: Addressing

- In this example, one router (with three interfaces) is used to interconnect seven hosts. Hosts are divided in 3 **subnets** (left, right, down) each one linked to one interface of the router.
- Each subnet is associated to a specific address, for instance the **left subnet** has address 223.1.1.0/24, so all interface in this network have IP addresses in the form 223.1.1.XXX:
  - 223.1.1.1 (host),
  - 223.1.1.2 (host),
  - 223.1.1.3 (host),
  - 223.1.1.4 (router).
- The other two subnets have 223.1.2.0/24 and 223.1.3.0/24 addresses.



Network Layer
Internet Protocol: Addressing

- In this example **there are 3 routers** that are interconnected by a direct (point-to-point) link. Each router has **3 interfaces**, one for each **point-to-point link** and one for the **broadcast link** that directly connects the router to a pair of hosts.
- There is a total of 6 subnets:
  - 223.1.1.0/24 (R1-hosts),
  - 223.1.2.0/24 (R2-hosts),
  - 223.1.3.0/24 (R3-hosts),
  - 223.1.9.0/24 (R1-R2),
  - 223.1.8.0/24 (R2-R3),
  - 223.1.7.0/24 (R3-R1).
- In this case routers are like gates (**gateways**) connecting different networks: if we could detach the interfaces from each router, we would have 6 isolated networks.
- Typically, **medium/large organizations** (such as a companies or academic institutions) have multiple interconnected subnets.



Network Layer
Ifconfig

- On Linux machines we can check our own address by using the *ifconfig* command (the equivalent on windows machines is *ipconfig*).
- Ifconfig (Interface configuration) provides the **list of all network interfaces** of the machine **along with their network configuration** (addresses, masks, etc.).
  - There are also modern commands to do so like ip.
- Usage:
  - To get the configuration:
    - $ ifconfig

```
eth0      Link encap:Ethernet  HWaddr 00:0F:20:CF:8B:42
          inet addr:217.149.127.10  Bcast:217.149.127.63  Mask:255.255.255.192
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2472694671 errors:1 dropped:0 overruns:0 frame:0
          TX packets:44641779 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1761467179 (1679.8 Mb)  TX bytes:2870928587 (2737.9 Mb)
          Interrupt:28
```

Ifconfig output example (Wikipedia)

Network Layer
Internet Protocol: Internet Addressing

- The idea of **breaking large networks into smaller ones is particularly important** on Internet where billions of devices (not to mention interfaces) must be connected.
- On Internet, IP **addresses must be carefully assigned** to avoid some issues:
  - The **forwarding tables** of routers may become very large.
  - We could have different interfaces with the **same address**.
  - We cloud **run out of addresses**.
- The approach is to divide Internet addresses by **providing subnets for the organizations** (such as ISPs, companies, institutions, etc.). There are 2 ways:
  - **Classful addressing** (older, no more used in practice).
  - **Classless addressing** (current).

Network Layer
Internet Protocol: Classful Addressing

- In **classful addressing** Internet addresses were divided into classes depending on a specific division:

	Format	Example	IPs per network
Class A	a.b.c.d/8	10.X.X.X	> 16 million
Class B	a.b.c.d/16	10.10.X.X	65535
Class C	a.b.c.d/24	10.10.10.X	254

- For example, if your organization needed 300 IPs, **you would have assigned a class B address** (e.g., 241.115.0.0) along with all IPs within it.
- There is a clear problem with this approach, since **only 300 IPs are needed, the additional 65335 IPs are wasted!**

Network Layer
Internet Protocol: Classless Addressing

- The modern approach is more flexible and is called **Classless InterDomain Routing** (**CIDR**, pronounced cider). Here, an organization could have assigned network addresses of any form:

a.b.c.d/X

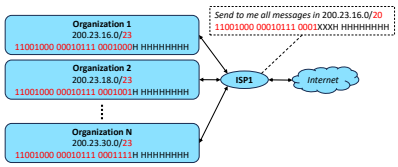
- Now if you need 300 IPs you could have assigned a.b.c.d/23 (e.g., 241.115.2.0/23) which **provides 512 IPs and only 212 wasted ones**.
- In "CIDRized" addresses:
  - The network-part of the address is called **prefix.**
  - The set of IPs reserved to the organization is called **block**.
- Note: **it is possible for blocks to overlap**, in this case the length (X) of the prefix can be used to discriminate different blocks.
  - Example: the IP 10.10.10.15/16 is not in the same block of 10.10.10.14/24.

Network Layer
Internet Protocol: Classless Addressing

- There is also the possibility **to create inner subnets** (i.e., subnets inside an organization) within a CIDR block.
- For example, the **CIDR block 241.115.0.0/16 can be decomposed** into additional subnets:
  - 241.115.1.0/24 (subnet 1),
  - 241.115.2.0/24 (subnet 2),
  - 241.115.3.0/24 (subnet 3),
  - etc.
- From a binary standpoint:

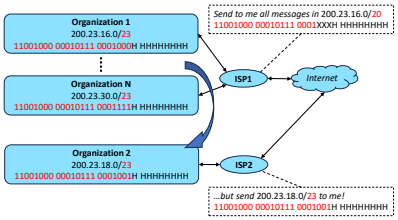
Network Layer
Internet Protocol: Address Aggregation

- The prefix-based addressing is very useful for devices that connect different prefixes.
- It is possible for a router to just remember (i.e., save into forwarding table) the prefixes.
- When a message with a specific prefix is received, it is forwarded to a more specific router, and so on.
- This approach is called **address aggregation** (or route summarization).



Network Layer
Internet Protocol: Address Aggregation

- The prefix-based addressing is very useful for devices that connect different prefixes.
- It is possible for a router to just remember (i.e., save into forwarding table) the prefixes.
- When a message with a specific prefix is received, it is forwarded to a more specific router, and so on.
- This approach is called **address aggregation** (or route summarization).



It is more effective as long as blocks are **clustered**!

Network Layer
Internet Protocol: Obtaining a Block

- In order **to obtain a block of IP addresses** for use within an organization's subnet there are 2 ways:
  1. You can **contact an ISP**, which would provide addresses from a larger block of addresses that had already been allocated.
     - For example, **the ISP may itself have been allocated the address block** 200.23.16.0/20, that can be further separated into sub-blocks of variable size depending on the ISP policy.
  2. You can ask to the **Internet Corporation for Assigned Names and Numbers** (**ICANN**), which is a no-profit global authority that has the responsibility to manage the IP address space (e.g., allocating address blocks to ISPs, etc.). **ICANN also manages the DNS root servers**, by assigning domain names and resolving domain name disputes.
     - Link: https://www.icann.org/

aruba.it TIM FASTWEB vodafone ICANN

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

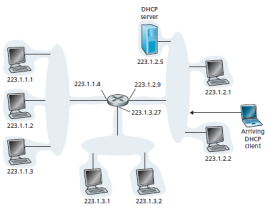
DIETI UNINA

Network Layer
Internet Protocol: IP Assignment

- Within a block of addresses, **IP addresses must be assigned** to individual interfaces.
- A **system administrator** can configure the IP addresses in two ways:
  - **Manually**: by assigning one-to-one IP addresses to hosts.
  - **Automatically**: the network autonomously assigns free IP to incoming hosts.
- The second approach (most common) is done by using the **Dynamic Host Configuration Protocol (DHCP)**.
- In addition to host IP address, **DHCP provides a host with the information needed to join the network**, such as subnet mask, the address of the default gateway (to go outside of the network), and the address of the local DNS server.
- The **DHCP is plug-and-play**, and it is typically used in our homes!

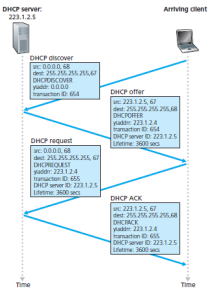
Network Layer
Internet Protocol: DHCP

- The **DHCP is a client-server protocol**: a newly arriving host (client) connects to the DHCP server to receive network information**. DHCP service may be provided by a computer or by the router itself**.
- Let's look to the previous example of **3 subnets connected by a single router**.
- Here we assume **the network on the right (233.1.2.0/24) to be endowed with a DHCP server**.
- When a new host joins the network, **it trades the IP address with the DHCP**.



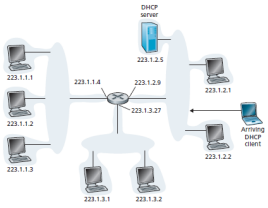
Network Layer
Internet Protocol: DHCP

- Adding a new host is a 4-steps process:
  1. **DHCP server discovery**: the new host sends in **broadcast a DHCP discovery message (UDP on port 67)** to find the DHCP server.
     - Destination IP: 255.255.255.255 (broadcast).
     - Source IP: 0.0.0.0 (this host).
  2. **DHCP server offer**: since multiple DHCP servers may be present, when a discovery message is received, a **server responds with a broadcast message (on port 68) offering a possible network configuration** (yiaddr filed – Your Internet ADDRess).
     - Destination IP: 255.255.255.255 (broadcast).
     - Source IP: 223.1.2.5 (DHCP server).
  3. **DHCP request**: the new client selects the accepted offer by **echoing back the offer message**.
  4. **DHCP ACK**: final **ACK message** confirming the transaction.



Network Layer
Internet Protocol: DHCP

- In this case everything works smoothly because the **DHCP and the arriving host are in the same subnet**.
- If a server exists but in a different subnet, we need a **DHCP relay agent** to forward DHCP messages (a router can do that).
- In this example we can configure our router to be a relay agent so that **also hosts from subnets 223.1.1.0/24 and 223.1.3.0/24 are served** by our DHCP.



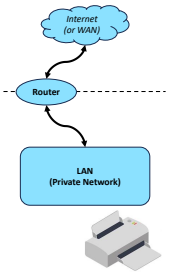
Network Layer
Route

- In Linux we can use the ifconfig in combination with the route command to see the network configuration provided by the DHCP.
- Usage:
  - See our current configuration (IP address, mask):
    - $ ifconfig
  - See our default gateway:
    - $ route -n

```
Lenovo-B50-80:~$ ifconfig wlp9s0
wlp9s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 100.103.0.64  netmask 255.255.0.0  broadcast 100.103.255.255
        inet6 fe80::47c4:ec38:2dc3:5d70  prefixlen 64  scopeid 0x20<link>
        ether 34:e6:ad:f0:d5:11  txqueuelen 1000  (Ethernet)
        RX packets 14398  bytes 20680768 (20.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4180  bytes 451906 (451.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

Lenovo-B50-80:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         100.103.0.1     0.0.0.0         UG    600    0        0 wlp9s0
```

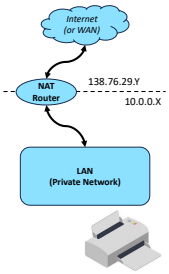
Network Layer
Internet Protocol: Visibility

- In Internet there are billions of connected devices that must be **associated with unique IP addresses to be reached by anyone**.
- Is it really necessary **to have all devices visible to anyone**?
  - For example: is it good to have **my printer reachable (usable) by anyone** on Internet?
- There are **clear issues in having all devices reachable** from outside local networks:
  - IP **addresses are finite**.
  - If devices are locally connected **it is often unreasonable (or undesirable) to expose all of them on Internet**.
  - Local **administrators should be fully aware of the overlying IP block** structure in order to assign unused addresses (which is often not up to them, e.g., in home networks).



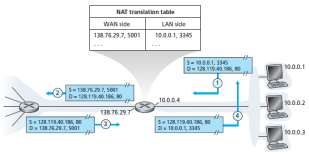
Network Layer
Internet Protocol: NAT

- The **Network Address Translation (NAT)** service allows to **remap the IP addresses** of packets within a network into different ones (**network masquerading**).
- This is performed by using **a translation table** associating **local IPs/ports (of the LAN) into global IPs/ports (of the WAN)**. This service can be offered by **routers**.
- The masqueraded local network is also called ***private network* or *realm with private addresses***. Private IP addresses are valid only inside the private network.



Network Layer
Internet Protocol: NAT

- This is an **example of a NAT-enabled router** connecting a network to internet.
- The four interfaces in the local network have the same subnet address of 10.0.0.0/24. The **router behaves as a message passer** that overrides the IPs with the one specified in the table:
  - **Outgoing packets** have their source IPs/ports overwritten **with a single WAN-side IP** (138.76.29.7) and a **fresh port** (5001).
  - **Incoming packets** have their destination IPs/ports overwritten **with the specific LAN-side IP** of the host (10.0.0.1) and the **initial port** (3345)
- It is important to notice that, since **hosts are unaware of the other hosts' traffic**, the **NAT cannot use the initial port** because multiple hosts may select the same port simultaneously.
- **Since the port is 16bits, NATs can manage over 60000 simultaneous connections**.

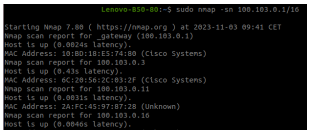


Network Layer
Ping

- To check if a host is reachable on a network, we can use the ping command (ping is the same on Linux/Windows machines).
- **Ping** (Packet Internet Groper) is a utility based on the **ICMP (Internet Control Message Protocol)** transport protocol and sends a "echo request" packet to a host that automatically answers with a "echo reply" message.
  - Ping also provide **the time elapsed between request/reply**, measuring the RTT.
- Usage:
  - To check the reachability:
    - $ ping [target address]

Network Layer
Nmap (pt.2)

- Besides port scanning, the nmap command can also be used to **map/scan the devices on the network**.
- It **relies on ping** (ICMP) to discover hosts on the network.
- Usage:
  - Ping-based hosts discovery:
    - $ sudo nmap -sn [gate address]/[subnet bits]

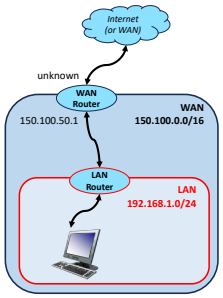


Network Layer
Internet Protocol: Reserved IP addresses

- Since **local administrators should be free to organize a private network** without interferences, by convention there are some **reserved IP blocks**, i.e., that ISPs/ICANN cannot assign to any organization. Some famous examples are:
  - 10.0.0.0 – 10.255.255.255 (reserved for private networks).
  - 192.168.0.0 – 192.168.255.255 (reserved for private networks).
  - 127.0.0.0 – 127.255.255.255 (indicate this machine, i.e., loopback).
    - Typically, 127.0.0.1 is used.
  - 0.0.0.0 (indicates current network).
    - Addresses having 0 in the host-part fall under this definition.
  - 255.255.255.255 (indicates broadcast).
    - Addresses having 255 in the host-part fall under this definition.
  - Etc.
- The necessity to reserve IPs is given by the fact that **private addresses may be the same as public (non-private) ones**, so routing within the network would be **ambiguous**.

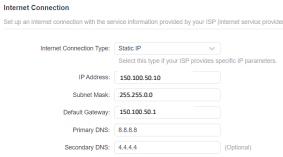
Network Layer
Internet Protocol: LAN Setup Example

- Let's see and example in which we want to create **a new local subnetwork (LAN) into a pre-existing network (WAN)**.
- The **network administrator** of the WAN provides us the following information:
  - The **IP of the WAN**: 150.100.0.0/16.
  - The **IP of the getaway** (leading to external WAN/Internet): 150.100.50.1
  - A free **IP for our network**: 150.100.50.10
- We have now to configure our router and our devices to setup the new LAN.
  - We are considering a **NAT-enabled router**.
  - We decide that **the IP of the new LAN** will be: 192.168.1.0/24 (reserved for local networks).



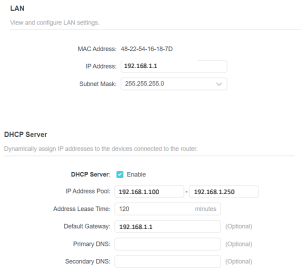
Network Layer
Internet Protocol: LAN Setup Example

- Our **local router has 2 interfaces**, one for the WAN-side and another for the LAN-side.
- Let's start by configuring our local router's **WAN-side settings**.
- On the router we have to specify the **following parameters**:
  - The IP address of the router in the WAN.
  - The subnet of the WAN.
  - The gateway.
  - One or more DNS.
- Notice that we can also specify **a dynamic IP** if the WAN has a DHCP service.



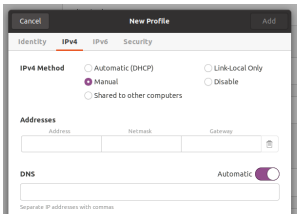
Network Layer
Internet Protocol: LAN Setup Example

- Now let's configure the **LAN-side settings**.
- We have to **give an IP** to the LAN-side interface of the router:
  - Since our network will be addressed as 192.168.1.0/24, we will give 192.168.1.1/24 to the router (it is typical to give .1 to it).
- In this example, our router also provide **DHCP service**, we can then specify:
  - **Address pool** (for DHCP hosts).
  - **Lease time** (timeout of the IP assignment, after that IP can be reused).
  - **Gateway** and **DNS** to be communicated to the hosts.



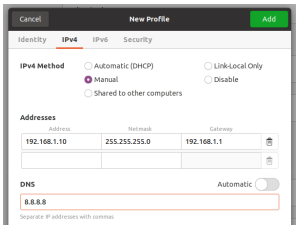
Network Layer
Internet Protocol: LAN Setup Example

- This is a simple example of how to configure this connection on a **new host** (Ubuntu 20.04).
- There are different methods to choose:
  - **Automatic (DHCP)**: uses DHCP to automatically configure the connection (you will get an IP from 192.168.1.100 to 192.168.1.250).
  - **Manual**: setup the connection by manually setting the configuration.



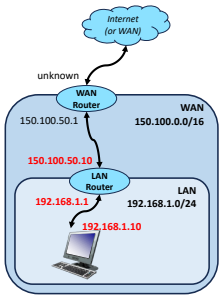
Network Layer
Internet Protocol: LAN Setup Example

- In a **manual setting** we need to know the network configuration (and the available IPs).
- We have to specify the main information for **connection setup**:
  - IP address of the host (static/fixed).
  - Subnet mask.
  - Gateway.
  - DNS (one or more).



Network Layer
Internet Protocol: LAN Setup Example

- In this created LAN we can **connect new hosts to the LAN** by specifying a manual IP address (outside the DHCP pool) or an automatic IP address (inside the DHCP pool).
- The connected **hosts will send their packets** to the local router (unaware of the outside WAN).
- We have the **local router configured to forward packets** to the WAN router (and possibly to the Internet).

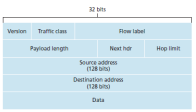


Network Layer
Internet Protocol: IPv6

- In the early 1990s, the Internet Engineering Task Force (IETF), which is an ONG founded in 1986, **began an effort to develop a successor to the IPv4** protocol to face the 32-bit **IPv4 address shortage**.
- IPv6 was designed to **ensure more addresses available**, but it was also an opportunity to **update other aspects of IPv4**, based on the accumulated operational experience.
- It is **not sure when exactly all the available IPv4 would be exhausted** (it was speculated to be 2008 or 2018, but we have still some IPs left).
- But **what about IPv5?** It was proposed in 1979 and was based on the experimental Internet Stream Protocol (ST). It was still relying on 32bit addresses, so it was **dropped mainly because of addresses shortage**.

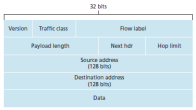
Network Layer
Internet Protocol: IPv6 Datagram

- IPv6 has the following **advantages**:
  - **Expanded addressing capabilities**: from 32 to 128 bits (i.e., 3.4028237e+38), which ensures that the world won't run out of IP addresses, as every grain of sand on the planet can be IP-addressable.
  - **A fixed-length 40-byte header**: some of IPv4 fields have been dropped or made optional.
  - **Flow labeling**: packets from some applications (e.g., audio/video streaming) can be grouped into a unique with flow having specific identification.
    - The precise role and usage of flows is still discussed.



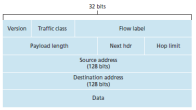
Network Layer
Internet Protocol: IPv6 Datagram

- IPv6 datagram is composed by the following fields:
  - **Version** (4 bits): identifies the **IP version** number. Not surprisingly, IPv6 carries a value of 6 in this field.
  - **Traffic class** (8 bits): like the TOS field in IPv4, can be used to give **priority to datagrams** (e.g., voice-over-IP over SMTP, etc.).
  - **Flow label** (20 bits): identify a **flow** of datagrams.
  - **Payload length** (16 bits): the **number of bytes of the payload** (40-bytes header excluded).
  - **Next header** (8 bits): identifies the **upper-level protocol** to which the contents (data field) of this datagram will be delivered (for example, to TCP or UDP). Similar to protocol field in the IPv4 header.
  - **Hop limit** (8 bits): Specifies the **time-to-live** as in TTL field of IPv4 (decremented every forward).
  - **Source and destination addresses** (128+128 bits): the IPv6 128-bit **addresses**.
  - **Data** (variable): the **payload** portion of the datagram.



Network Layer
Internet Protocol: IPv6 Datagram

- What is **missing** from IPv4:
  - **Fragmentation-related fields**: IPv6 **does not allow fragmentation and reassembly on intermediate routers**, but only on hosts.
    - If an IPv6 datagram received by a router is too large to be forwarded, the **router simply drops the datagram and sends a "Packet Too Big" error** message back to the sender. The sender can then resend the data, using a smaller IP datagram size.
  - **Header checksum**: it takes time on routers, and it is redundant as **link-layer protocols typically perform checksum** on the whole packet.
  - **Options**: are no longer a part of the standard IP header, **some options can be specified into next header** field (along with TCP/UDP identifiers).



Network Layer
Internet Protocol: From IPv4 to IPv6

- There is one big issue with IPv6: **it is not backward-compatible**. IPv4-capable systems are unable to handle IPv6 datagrams.
- How will Internet adopt IPv6?
  - **The flag day approach**: in a given time and date **all Internet machines would be turned off and upgraded from IPv4 to IPv6**. A similar approach have been used when TCP was introduced but it was a mess (even for the small internet network of the time). A flag day involving billions of devices is even more unthinkable today.
  - **The tunneling approach**: the use of specific routers (tunnels) in charge of **mapping IPv4 datagrams into IPv6 datagrams** and vice versa in almost-transparent way. This is probably the most realistic approach, already **used in practice**.
- While the adoption of IPv6 was initially slow, it is accelerating. Google reports that **about 45% of clients (in 2023) accessing Google** services use IPv6 (https://www.google.com/intl/en/ipv6/statistics.html).

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Network Layer
Routing: Introduction

- We recall that, on packet receival, a **router can perform 3 actions**:
  - **Forwarding the packet** as is to a suitable neighbor.
  - **Forwarding a modified/replaced packet** (e.g., fragmentation, masquerading, etc.).
  - **Dropping the packet** (e.g., if expired).
- As we saw**, routers are endowed with forwarding tables** that specify the local forward strategy (in the neighborhood of the router). The way these tables are created can be centralized or decentralized.
- To decide where packets should be forwarded routers make use of **routing algorithms**, which determine good paths (i.e., sequence of routes) from senders to receivers.
- Typically, **a "good" path is one that has the minimal cost** (shortest or fastest path, etc.), in real-world **there are also policy issues** to be considered (e.g., rules specifying if an organization can talk with another etc.).

Network Layer
Routing: Problem Formulation

- We can **formalize** our network as a graph $G = (N, E)$, where:
  - N is a set of **nodes** (the routers of our network).
  - $E \subseteq N \times N$ is a set of **edges** (physical links) represented as a pair of nodes.
- An **edge also has a value representing its cost**, which may reflect the physical length of the link, the link speed, or the monetary cost associated with a link.
- We can formalize a generic **cost** as a function $c: N \times N \rightarrow \mathbb{R}$ such that, given a generic couple of nodes $(x, y) \in N \times N$:
  - $\forall x \in N$, then $c(x, x) = 0$
  - If $(x, y) \notin E$, then $c(x, y) = \infty$
  - If $(x, y) \in E$, then also $(y, x) \in E$ (undirected graph) and $c(x, y) = c(y, x)$

Network Layer
Routing: Problem Formulation

- Following this formulation, a **path** π in G is a **sequence of nodes** $\pi = (x_1, x_2, \dots, x_n)$ such that **all adjacent nodes are linked**, formally:

$$\forall x_i, x_{i+1} \in \pi, (x_i, x_{i+1}) \in E$$

- The **overall cost of the path** is the sum of the links' costs:

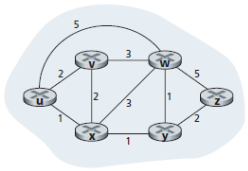
$$c(\pi) = \sum_{i=1}^{n-1} c(x_i, x_{i+1})$$

- Let's call $P(x, y)$ the **set of all possible paths** connecting the nodes x and y, we can define **the best path** as the path $\pi^* \in P(x, y)$ having minimum cost:

$$\forall \pi \in P(x, y), c(\pi^*) \leq c(\pi)$$

Network Layer
Routing: Problem Formulation

- In this example we have a graph composed of 5 nodes:
  - $N = \{u, v, w, x, y, z\}$
  - $E = \{(u, w), (u, v), (w, x), \dots\}$
  - $c(u, w) = 5,\ c(u, v) = 2,\ c(u, x) = 1, \dots$
- For instance, the best path π^* between nodes x and z is:
  - $\pi^* = (x, y, z)$
  - $c(x, y) = 1,\ c(y, z) = 2$
  - $c(\pi^*) = 3$



Network Layer
Routing: Problem Formulation

- Algorithms able to find such optimal path π^* between two nodes are called **shortest path algorithms**.
- In the case of computer networks, we are interested in **finding such optimal path** not for just two nodes but **for all possible couples of nodes**.
- A routing algorithm **converges** when the shortest path for all couples of nodes in the network is found.
- On networks **there are 2 families of algorithms** that are typically used:
  - Distance-Vector.
  - Link-State.

Network Layer
Routing: Distance-Vector Algorithm

- The **Distance-Vector** (DV) algorithm is a **decentralized** approach that exploit **local knowledge** about the network combined with nodes communication to find the shortest paths.
  - It is based on the **Bellman-Ford algorithm** that is used to compute paths.
- Here **each node receives some information** from one or more of its directly attached neighbors, performs a calculation, and then **distributes the results** of its calculation back to its neighbors.
- This algorithm is **asynchronous** in that it does not require all of the nodes to be coordinated with each other.

Network Layer
Routing: Distance-Vector Algorithm

- Let's call $d^*(x, y)$ be the cost (distance) of the best path from node x to node y. We can apply the **Bellman equation**:

$$d^*(x, y) = \min_v \{c(x, v) + d^*(v, y)\}$$

- where v is a neighbor of x. This is quite intuitive, **if v is the first node of the best path**, then the rest of the nodes must be the ones into the best path from v to y.
- If we are able to find such v **we can just add it into the forwarding table** and forward to it all packets directed to y.
- To do that, **we should have an estimation of $d^*(x, y)$** for all neighbors (the distance vector).

Network Layer
Routing: Distance-Vector Algorithm

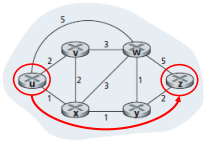
- The pseudocode:

```
DistanceVector(x)
 for all nodes v
  if v is a neighbor of x then
   D_x(v) = c(x,v)
  else
   D_x(v) = ∞
 for all neighbors w and destinations y
  D_w(y) = ?
 send initial D_x(·) to all neighbors

 repeat
  if cost of a neighbor updated then
   for all nodes y
    D_x(y) = min_v{c(x,v) + D_v(y)}
   if D_x(y) changed for any destination y then
    send updated D_x(·) to all neighbors
 until false //forever
```

- We use the **data-structure $D_w(v)$** to store the distance vectors from all nodes w to target nodes v.
- During the **initialization** phase:
  - Only **distances to neighbors are set**.
- During the **online** phase:
  - The **news** from the neighborhood are received.
  - Distance vectors are **updated** (Bellman equation).
  - Local changes are **communicated** to adjacent nodes.

Network Layer
Routing: Distance-Vector Algorithm

- To clarify this process **let's focus on just one specific path**, between u and z:
  1. On start, **u only knows its neighbors**. The cost to z is infinity ($D_u(z) = \infty$).
  2. Node **y wakes up** and discovers a better path to z, which is the direct link y-z with cost 2 ($D_y(z) = c(y,z) = 2$). It communicates the good news to x.
  3. Node **x discovers that the best path to z now passes through y** with cost 3 ($D_x(z) = c(x,y) + D_y(z) = 1 + 2$), and forwards the good news to u.
  4. Node **u receives the news**. Now there is a path to z that costs less than infinity passing through x ($D_u(z) = c(u,x) + D_x(z) = 1 + 3$).



- The **same process** takes place for all nodes/paths.

Network Layer
Routing: Distance-Vector Algorithm

- As for the computational complexity, every time the cost of a node is updated, all nodes have to re-evaluate their edges. This process leads to a typical **worst-case complexity of** $O(|N||E|)$.

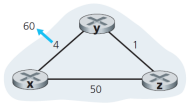
- Pros:
  - The **algorithm is asynchronous**, this facilitates the computation as routers can on-line and in any stage adapt to updated costs.
- Cons:
  - **Slow convergence**: updates spread slowly as all nodes have to detect the change and send an updated to the adjacent ones.
  - **Count-to-infinity**: since only improved paths are communicated, if a link or a node fails, the propagation of this "bad news" will be slow: the adjacent node of a broken link will immediately switch to an alternative route, but they cannot be sure that the new best path does to not pass through the failure.

Network Layer
Routing: Count-to-infinity

- Let's consider this **simplified example** in which the link x-y gets a sudden increase of cost.

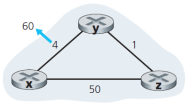
- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

This is the **initial situation** before the increment (c(x,y) is still 4).



Network Layer
Routing: Count-to-infinity

- Let's consider this simplified example in which the link x-y gets a sudden increase of cost.



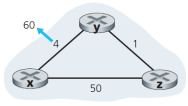
- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

Node y detects the increment and updates distance-vector to x:
$D_y(x) = \min\{ 60, 1 + D_z(x) \} = 1 + D_z(x) = 6$,

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

Network Layer
Routing: Count-to-infinity

- Let's consider this simplified example in which the link x-y gets a sudden increase of cost.



- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

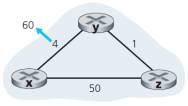
But vector $D_y(x)$ is used by z to compute the path to x, so also $D_z(x)$ must be updated!

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

Network Layer
Routing: Count-to-infinity

- Let's consider this simplified example in which the link x-y gets a sudden increase of cost.



- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

But vector $D_z(x)$ is itself used by y to compute the path to x, so $D_y(x)$ must be updated again!

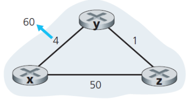
- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

- $D_y(x) = 1 + D_z(x) = 8$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

Network Layer
Routing: Count-to-infinity

- Let's consider this simplified example in which the link x-y gets a sudden increase of cost.



- $D_y(x) = 4$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 5$,
- $D_z(y) = 1$.

This is a loop: the 2 vectors will be continuously updated until a stable configuration is reached.

- $D_y(x) = 1 + D_z(x) = 6$,
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- **$D_z(x)$ = 1 + $D_y(x)$ = 9,**
- $D_z(y) = 1$.

- **$D_y(x)$ = 1 + $D_z(x)$ = 8,**
- $D_y(z) = 1$,
- $D_x(y) = 4$,
- $D_x(z) = 4 + D_y(z) = 5$,
- $D_z(x) = 1 + D_y(x) = 7$,
- $D_z(y) = 1$.

Network Layer
Routing: Link-State Algorithm

- The **Link-State** (LS) algorithm is a **centralized** approach that exploit full knowledge about the network in order to find the best path.
  - The LS algorithms are based on variation of the **Dijkstra algorithm**.
- Notice that such level of knowledge can be achieved in practice by having **each node to broadcast link-state packets**, containing IDs and costs of its attached links, to all other nodes in the network.
- The basic version of the algorithm computes the **shortest paths from a starting node to all possible destination nodes** (so it must be executed **on all nodes**).

Network Layer
Routing: Link-State Algorithm

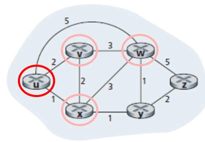
- The pseudocode:

```
LinkState(x):
  N' = {x}
  for all nodes v
    if v is a neighbor of x then
      D(v) = c(x,v)
    else
      D(v) = ∞

  repeat
    find w not in N' such that D(w) is a minimum
    add w to N'
    update D(v) for each neighbor v of w and not in N':
            D(v) = min( D(v), D(w) + c(w,v) )
  until N'= N
```

- We use **the data-structure D(v)** to store the distance from the current node to all possible destination nodes.
- During the **initialization** phase:
  - We assume that **all costs are known**!
  - Only **distances to neighbors are set**.
- During the **construction** phase:
  - Select the **closest** node w.
  - **Explore the neighborhood** of w, checking if the path through w is better than the current one.
  - Exit when **all nodes have been explored**.

Network Layer
Routing: Link-State Algorithm

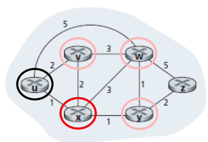
- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

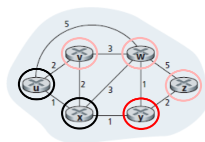
- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

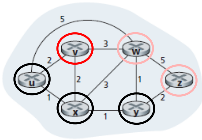
- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

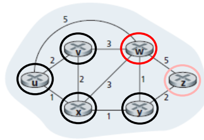
- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

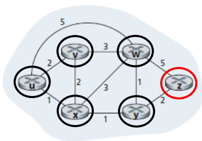
- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

- Let's consider the previous 6-nodes example. Here we show how the LS algorithm works when invoked on the u node:
  - In this case, we will use an additional function p(x) to store the node preceding the selected one.
  - If we want to retrieve the path, we can just backtrack all predecessors until the current node is reached.



step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	2, u	5, u	1,u	∞	∞
1	ux	2, u	4, x		2, x	∞
2	uxy	2, u	3, y			4, y
3	uxyv		3, y			4, y
4	uxyvw					4, y
5	uxyvwz					

Network Layer
Routing: Link-State Algorithm

- As for the computational complexity, **at each iteration we check all nodes** of the network, except the source node, then a new source is selected. This means that **we are removing one node form each iteration**, so the total number of iterations is $|N|(|N|+1)/2$ which **is $O(|N|^2)$ in the worst case**.
- This version of the algorithm is quite basic, complexity can be reduced by introducing more sophisticated data-structures (e.g., heaps), **we can reach better performance**: $O(|N|+|E|\log(|E|))$.

- Pros:
  - **Faster convergence**: all communications are performed simultaneously.
  - **No count-to-infinity**: the state of all links is propagated.
- Cons:
  - **Synchronous**: nodes must receive information from the whole network before to start.

Network Layer
Traceroute

- In Linux we can use **traceroute** to trace the route of our packets toward a specific destination:
  - Install traceroute:
    - $ sudo apt-get install traceroute
  - Trace a route:
    - $ traceroute ADDRESS

- The traceroute command will return **IPs of all devices encountered** from the source host to the destination host.

- If **we try it several times** targeting a distant host (e.g., google.com), **we should see different devices** in the list (due to routes updating).

Network Layer
Routing: DV vs. LS

- The **DV and LS algorithms take complementary approaches** toward computing routing:
  - In the DV algorithm exploits **local information** about neighbors.
  - The LS algorithm requires **global information** about all nodes.

- **Message complexity**: **LS is more complex** as synchronous communication between all nodes is needed. In DV, communication is needed only if a best path changes.
- **Speed of convergence**: **DV requires time to converge**, in the meantime we can have suboptimal paths or loops. DV algorithms also suffer of count-to-infinity problem.
- **Robustness**: **LS is considered more robust** as **forwarding tables are calculated separately**. Each node receives information from all other nodes and creates its own table. In DV all computations are chained, each node depends on the table of others. If one table is incorrect, all tables get the error.
  - In 1997 a **malfunctioning router from a small ISP caused a chain reaction**, which flooded backbone routers, and disconnected part of Internet for several hours.

- In the end, there is no winner, **both solutions are used in Internet**.

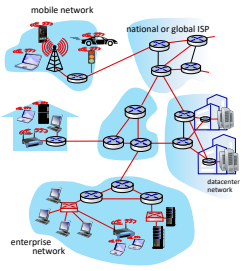
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

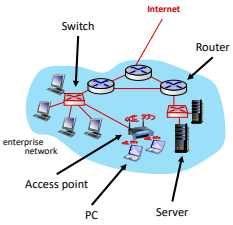
Link Layer
From Network to Link & Physical

- We have seen that the network layer basically provides communication layer between two hosts (wherever they are).
- Link and physical layers provide communication between **two connected hosts**:
  - The **link layer** is the portion of the stack dedicated to the **transmission of packets over links** (transmission channels), from node to node of the network.
  - The **physical layer** is the portion of the stack that regulates the **structure of links** (transmission medium, connectors, cable types, etc.) and how bits are represented/transmitted along the link.



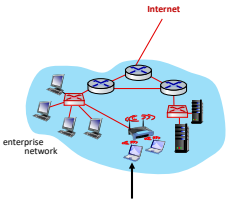
Link Layer
From Network to Link & Physical

- **Link and physical layers are strictly intertwined** (these are often grouped into a single layer called network access).
  - Some common protocols (e.g., Ethernet) cover both layers.
- Differently from the previous layers, **these 2 layers are present in all pieces** of the network infrastructure:
  - Nodes: hosts (PC, servers, etc.), routers, switches, hubs, WiFi access points, etc.
  - Links: wired (copper cables, optical-fiber), wireless (radiofrequencies).



Link Layer
From Network to Link & Physical

- To transmit an IP datagram, we must transfer it from source host to destination host by jumping over each of the individual links/devices along the path.
- To be transferred over a link, **datagrams are encapsulated into link-layer frames** depending on the specific type of link.
- This is **the last encapsulation**, frames are converted into signals and transferred over the link following its specification. Signals can be:
  - Electric pulses.
  - Light.
  - Radio waves.



In order to reach Internet, packets of this host have to cross 4/5 links (wireless and wired), 1 access point, 1 switch and 2/3 routers.

Link Layer
Services

- **Link layer may offer up to 4 services**: framing, link access, reliable delivery, error detection and correction.

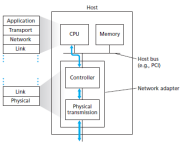
1. **Framing**: to encapsulate **datagrams into link-layer frames** having the upper-layer datagram as payload and **specific header/tailer**. The structure of the frame depends on the link-layer/physical-layer protocols.
2. **Link access**: to define the rules **regulating the access to the link** by means of a **Medium Access Control** (MAC) protocol. Such rules depend on the link architecture:
   - For **point-to-point links** (single sender and single receiver), the **MAC protocol is simple** (or nonexistent). The sender can send a frame whenever the link is idle.
   - For **broadcast links** there is the so-called multiple access problem. Here, **the MAC protocol serves to coordinate the frame transmissions** of the many nodes.

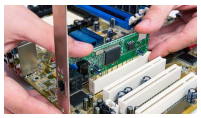
Link Layer
Services

3. **Reliable delivery**: it guarantees that frames transmitted across the link are received.
   - A link-layer reliable delivery service **is often used for links that are prone to high error rates**, such as a wireless link, with the goal of correcting an error locally.
   - It may **produce strong overhead**. Since other application/transport protocols (e.g., TCP) are reliable, this feature **is not always implemented**.
4. **Error detection and correction**: the link-layer **hardware can be affected by the bit flipping problem**. Many link-layer protocols implement more sophisticated (and hardware-embedded) checking/correction strategies in addition to the ones from upper layers (e.g., TCP and IP checksums).
   - Since there is no further encapsulation, these **checks basically cover the whole message**.
   - Hardware-embedded **checks may be a lot faster**.

Link Layer
Implementation

- In network devices (end-systems, routers, etc.) the link-layer functionalities are **implemented both software-side and (mostly) hardware-side**.
- In computers (i.e., end-systems) there are network adapters called **Network Interface Cards** (NICs) having a specialized chip (**controller**) to implement link-layer functionalities.
  - These **cards used to be separated from the motherboard** (plug-in into PCI slots), but recently this approach is changing (LAN-on-motherboard).



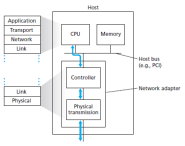


NIC plugged in a motherboard

Lattepanda mini-PC with embedded NIC

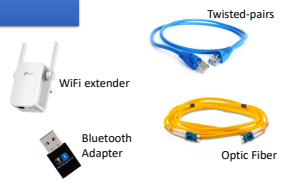
Link Layer
Implementation

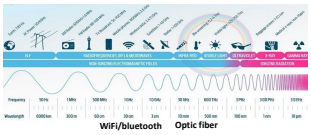
- **Software**: runs on the CPU and provides high-level functionalities such as **addressing, interrupts/hardware management, handling of errors, encapsulation/decapsulation** of datagrams.
  - At link-layer we have an additional address, the **MAC address** that is used to identify the NIC.
  - The **datagrams must be stored into the memory** to be used by the upper-level protocol. They have to be:
    - **Retrieved the datagram** from memory during encapsulation (sender-side).
    - **Inserted the datagram** into the memory during decapsulation (receiver-side).
- **Hardware**: works as a typical I/O device, **converting the data into the signal** to be transmitted depending on the physical protocol (e.g., ethernet, wireless, etc.).



Link Layer
Physical Technologies

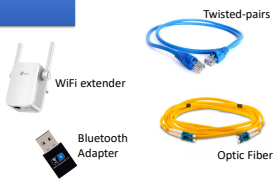
- The **link-layer protocol**, hence the resulting frame structure, **depends on the technology** of the physical link (physical layer).
- Here some **common technologies**:
  - **Ethernet** 802.3: wired connection based on electric pulses over **4 twisted-pairs copper cables** (most common) or photons over **optic fiber** cables.
  - **WiFi** 802.11: wireless connection based on **radiofrequency** (2.4GHz, 5GHz, 6GHz).
  - **Bluetooth**: wireless connection based on **radiofrequency** (2.4GHz).





Link Layer
Physical Technologies

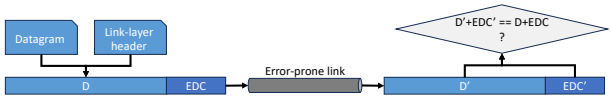
- Physical technologies are continuously evolving, their boundaries in term of distance/bandwidth are often updated.
- Different technologies have different pros and cons.



Type	Signal	Distance	Bandwidth	
Wireless	Radio	30-50 m	1.3 Gbps	→ Security, speed
Twisted pair copper cables	Electricity	30-100 m	1-25 Gbps	→ Interferences
Fiber-optic cable	Light	100-200 km	1-400+ Gbps	→ Fragile, expansive

Link Layer
Error Detection and Correction: Problem Formulation

- In link-layer is possible to perform **bit-level error detection and correction**.
- Let's assume ***D* of size *d* being our data**, to protect it against bit errors we include in the message **some error detection and correction bits: *EDC***.
- The goal is to find **if received *D'* and *EDC'* differs from the original *D* and *EDC*** and, if so, **to possibly retrieve the initial D and EDC**.



- Note that even with the use of error-detection bits **there still may be undetected bit errors**, the receiver may be unaware that the received information contains bit errors.

Link Layer
Parity Check

- **Parity check** is perhaps the simplest form of error detection: **we include only 1 additional parity bit** to the message such that the total number of 1s among the d+1 bits of the message is even (even parity scheme) or odd (odd parity scheme).

Message	Priority bit (even scheme)	Priority bit (odd scheme)
10101100	0	1
11010000	1	0

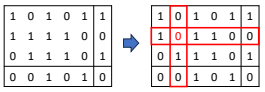
- The **receiver need only to count the number of 1s** in the received d + 1 bits. If an odd number of 1-valued bits are found with an even parity scheme (or vice versa), the receiver knows that at least one bit error has occurred.

Link Layer
Parity Check

- Clearly **this approach only works if an odd number of bit errors have occurred**, but how likely is that?
- If the **probability of bit errors is small** and errors can be assumed to occur **independently from one bit to the next**, the probability of multiple bit errors in a packet would be extremely small, but **this is not the case of computer networks**.
- In practice, it has been observed that **errors are often clustered** together in "**bursts**" (not independent at all). Under burst error conditions, the probability of undetected errors using single-bit parity can approach 50 percent, so it is not really useful.

Link Layer
Parity Check (Two-dimensional)

- A possible way to improve this approach is to **use multiple parity bits**.
- The **two-dimensional parity** is a techniques in which message is divided into *n* rows and *m* columns, each of them having a specific parity bit.
- Here, receiver can detect that an error occurred, and also which bit, therefore **attempting a correction**.



- Two-dimensional parity can also detect (but not correct!) any combination of two errors in a packet.

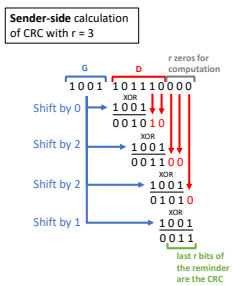
Link Layer
Cyclic Redundancy Check (CRC)

- **Cyclic Redundancy Check** (CRC) is a widely used error detection technique. It works as follows:
  - In order to send *d* bits of data *D* the **sender and the receiver agrees on a pattern of r+1 bits called generator** (G), having the most significant bit (leftmost) at 1.
  - For a given piece of data D, the **sender will choose r additional bits** (CRC bits) to be appended to the message **such that, D+CRC is modulo-2 divisible by G** (no reminder).
  - The **receiver divides the D+CRC message by G**, if the reminder is zero, the message is correct, otherwise an error occurred.
  - This operation is implemented **by shifting G up to the most significant 1 bit of the message and performing a bit-wise XOR**.

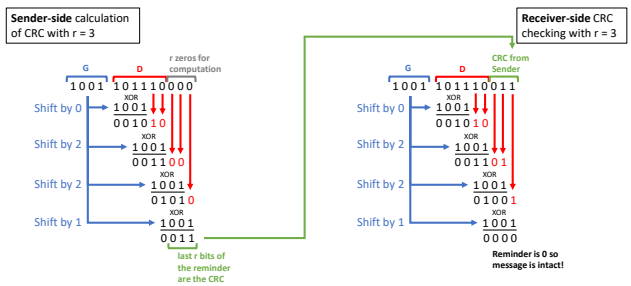
Link Layer
Cyclic Redundancy Check (CRC)

- Let's now consider a **simple example** of a 6-bits message D and a 3-bits CRC where:
  - d = 6
  - D = 1 0 1 1 1 0
  - r = 3
  - G = 1 0 0 1
- On the **sender-side we have to create the CRC** from D and G, this CRC is attached to the message and transmitted to the receiver.
- On the **receiver-side we use D, G, and the received CRC** to understand if the received message is intact.

Link Layer
Cyclic Redundancy Check (CRC)



Link Layer
Cyclic Redundancy Check (CRC)



Link Layer
Cyclic Redundancy Check (CRC)

- International **standards have been defined** for 8-, 12-, 16-, and 32-bit generators (i.e., r=8, r=12, r=16, r=32).
- The **CRC-32 32-bit standard**, which has been adopted in several link-level IEEE protocols, uses the following generator (33 bits):

$$G_{CRC-32} = 100000100110000010001110110110111$$

- Each of the **CRC standards can detect for sure burst errors of less than r + 1 bits**, while for **error greater than r + 1 there is a probability P** of finding the error which is:

$$P = 1 - 0.5^{r}$$

- Therefore, the **probability of finding the error increase with the increment of r**.

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Link Layer
Link Types

- There are basically 2 types of links that can be managed:
  - **Point-to-point link**: consists of a **single sender** at one end and a **single receiver** at the other end. The **point-to-point protocol** (PPP) is one example of protocol managing such links.
    - E.g., direct ethernet link between 2 computers.
  - **Broadcast link**: multiple sending and receiving nodes all connected to **the same shared channel**. The term broadcast is used because when one node transmits a frame it is received by all nodes on the channel.
    - E.g., Ethernet bus, half-duplex Ethernet (rare, as most cables are todays full-duplex) or wireless LANs.

- The access to **broadcast links have to be coordinated** (multiple access problem) as multiple communication on a single link may interfere each other.

Link Layer
Collisions

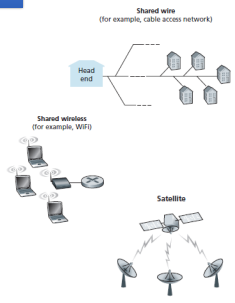
- The main problem of broadcast link is the **collision**: if multiple nodes are simultaneously transmitting frames on the same channel, **all such frames overlap becoming incomprehensible**.

- These **collided frames are** then received by all nodes on the channel and **dismissed as errors** (no harm is done), but:
  - All **transmitted frames are lost**.
  - The **time-interval is wasted**, as the channel has been used to transmit useless data.

Link Layer
Multiple Access Protocol

- In computer networks **multiple access protocols** are used **to regulate transmissions via broadcast channels**, so that:
  - **Collisions are managed**.
  - Each node has a chance to transmit, so **nodes do not monopolize the link**.
  - Established **connections are not interrupted** (e.g., checking if link is busy).

- Such protocols are needed for a variety of network settings, including both wired, wireless or satellite networks, **where hundreds or thousands of nodes can directly communicate over a broadcast channel**.



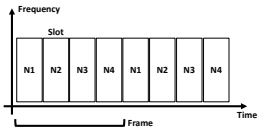
Link Layer
Multiple Access Protocol

- The primary role of a **multiple access protocol** is to somehow **avoid collisions** (there are dozens of protocols over different link-layer technologies). Main approaches are:
  - **Channel partitioning**: the bandwidth is partitioned for different nodes.
  - **Random access**: the nodes "gamble" for the access.
  - **Taking-turns**: the nodes waits for their turn.

- **Desiderata**: a multiple access protocol for a **broadcast channel of rate *R* bps** should also provide the following characteristics:
  - To **maximize the usage of the channel**: if M nodes have data to send, each one should have, in average, a throughput of R/M bps (if M=1 then throughput should be R).
  - To be **decentralized**: a master node may be a single point of failure.
  - To be **simple and lightweight**: tons of frames are sent, there must be no overhead.

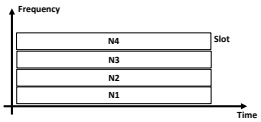
Link Layer
Channel Partitioning Protocols: TDMA

- **Time division multiple access** (TDMA): let's consider a channel with N nodes having transmission rate of *R* bps, **TDMA divides time into time frames** (steps) and further divides **each time frame into N time slots**.
- Each **time slot is assigned to one** of the N nodes. Whenever a node has a packet to send, it waits for the assigned time slot.
- Typically, **slot sizes are chosen so that a whole packet can be transmitted** during a slot time.



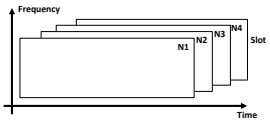
Link Layer
Channel Partitioning Protocols: FDMA

- **Frequency division multiple-access** (FDMA): divides the R bps channel into different frequencies (each with a bandwidth of R/N) and assigns each frequency to one of the N nodes.
- FDMA and TDMA shares pros and cons:
  - They **avoids collisions and divide the bandwidth fairly** among the N nodes.
  - A **node is limited to a bandwidth of R/N**, even when it is the only node with packets to send.



Link Layer
Channel Partitioning Protocols: CDMA

- **Code division multiple access** (CDMA): assigns a different code to each node which is used to encode/decode the data bits it sends.
- If the codes are chosen carefully, **CDMA networks allows different nodes to transmit simultaneously** without causing interference.
- CDMA works **mainly on wireless channels**; it has been used in military systems for some time (due to its anti-jamming properties) and **also in cellular telephony**.

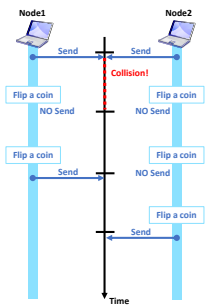


Link Layer
Random Access Protocols

- In random access protocols, a **transmitting node always transmits at the full rate** of the channel (at *R* bps).
- If a **collision occurs** (i.e., at least 2 nodes are transmitting) all transmitting **nodes waits a random delay** before attempting again the retransmission.
- Since this **selection is performed independently**, it is possible for the 2 nodes to **choose a delay which is different enough** to allow one of the two contenders to sneak the message in.
- If, otherwise, a **similar delay is chosen**, a new collision occurs, and **the process is iterated**.

Link Layer
Random Access Protocols: Slotted ALOHA

- The **slotted ALOHA protocol** works as follows:
  - The **time is divided into slots** (as in TDMA) where each slot is large enough to contain one frame.
  - The nodes are synchronized, so **each node transmit frames only at the beginnings of a slots**.
  - **If no collision** is detected within the slot, the **communication continues**.
  - Otherwise, **if a collision is detected** within a slot, **each colliding node have a probability *p ϵ [0,1]* of resending this frame** in each one of the following slots, until the frame is successfully sent.
- Features:
  - **If there is only one node, it will use the full rate** R of the channel (no collisions).
  - Is **decentralized** as nodes are totally independent from the others besides the synchronization.
  - Is **simple to implement** and to execute (random selection is quite fast).
  - Having **multiple consecutive collisions is unlikely**.

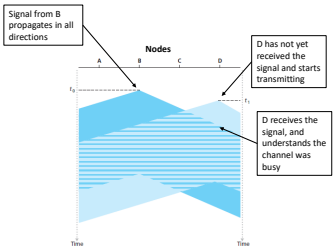


Link Layer
Random Access Protocols: CSMA

- One **weakness of ALOHA is that we may start transmission** (producing a collision) **even if the channel is already busy**. We understand it just through the effect of collisions.
- A solution is **to monitor the channel** and to attempt transmission only if the channel is idle.
- **Carrier sense multiple access** (CSMA) and **CSMA with collision detection** (CSMA/CD) protocols are based on 2 principles:
  - **Carrier sensing**: nodes **listen to the channel before transmitting**. If a frame from another node is currently being transmitted, it waits until no transmission is detected.
  - **Collision detection**: nodes **listens to the channel while it is transmitting**. If a collision is detected, it stops transmitting and waits a random amount of time before restarting.
- If all nodes perform carrier sensing, **why do collisions occur in the first place**?

Link Layer
Random Access Protocols: CSMA

- Because of the **delay in the signal transmission**!
- Even if the propagation of signals in the channel is typically near the speed of light, **it takes time to reach all other nodes**. Therefore, a second node detect the transmission only after it has started.
- Because of this delay between transmission start and detection, **a node may consider as free a channel that is actually in use**, producing a collision.



Link Layer
Random Access Protocols: CSMA

- The **pure CSMA** is very simple:
  1. **Check if the channel is busy**.
  2. **If channel is idle, send a frame**.
- **In CMSA collisions are not detected** but are still possible, we understand that a frame is lost just **because the ACK is not received**.
- The **CSMA/CD is more evolved** (currently **implemented on Ethernet**):
  1. **Check if the channel is busy.**
  2. **If channel is idle, send a frame**.
  3. **While transmitting, check for possible collisions.**
  4. **If collision is detected**, stop transmitting and **wait for a random period** $K \in \{0, \ldots, 2^n-1\}$ where n **is the number of detected collisions** on the current frame (**binary exponential backoff**).
- Since number of possible periods increases, **the probability to successfully send the frame increases with the number of collisions**.

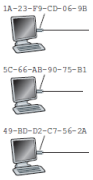
Link Layer
Taking-turns

- **Polling protocol**: there is **one master node that selects in a round-robin way one node per time** allowed to transmit (up to the max throughput). This process is iterated every time transmission stops (e.g., Bluetooth).
  - There are **no collisions**.
  - There is a **polling delay** (time to select nodes).
  - The approach is **centralized**, there is a single-point-of-failure.
- **Token-passing protocol**: there is **no master node, nodes exchange a special frame called token**, if a node receive the token it is allowed to transmit, then the token is passed to another node.
  - There are **no collisions**.
  - The approach is **decentralized**.
  - There are **problems if some node forgets to release the token** (monopolizing the link).

Link Layer
MAC Addresses

- At the link-layer, **devices are identified by MAC addresses**:
  - **Each network interface has a specific MAC address** (it was designed to be fixed but it can be changed).
  - **Each manufacturer has its own MAC**.
- The **MAC address** (or physical address) **is a link-layer address composed by 6 bytes** (2^48 possible addresses) often represented in hexadecimal notation:

  1A:23:F9:CD:06:9B or 1A-23-F9-CD-06-9B
- **MAC addresses are local** (while IPs are global):
  - All interfaces are associated to a MAC address, but this is only used inside a LAN.

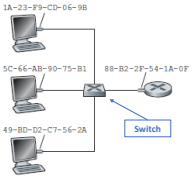


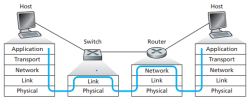
Link Layer
MAC Addresses

- **In a LAN, 2 interfaces (A and B) communicate as follows:**
  - **A includes B's MAC address into the frame** and transmits it.
  - **B receives the frame and compares its own MAC with the destination MAC** of the frame.
  - **If the 2 MACs are equal, the frame is accepted**, otherwise the frame is rejected (the rest of the stack is not involved).
- There is also the possibility **to send broadcast messages** (that are accepted regardless of the MAC), for LANs that use 6-byte addresses (such as Ethernet and 802.11), **the broadcast address is a string of 48 consecutive 1s** (that is, FF-FF-FF-FF-FF-FF in hexadecimal notation).
- In a LAN it is quite possible (if not frequent) for an interface to receive frames directed to another interface, **the role of the MAC address is to filter out unintended frames** without disturbing the host.

Link Layer
Switches

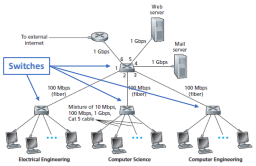
- **Switches are the link-layer equivalent of the routers**:
  - There is **no routing** algorithm implemented.
  - **Only MAC addresses are used**, the IP addresses are not considered.
- The role of the switch is to **receive incoming link-layer frames and forward them** onto outgoing links:
  - The **switch is transparent** to the hosts and routers in the subnet.
  - A switch also has **buffers on interfaces**.
- **Switches have forwarding tables** that associate MAC addresses to interfaces.
  - The **table is updated automatically and dynamically** (self-learning) as new devices are discovered.





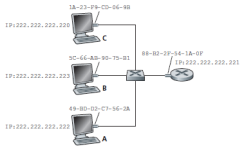
Link Layer
Switched LAN

- It is typical for **LANs to use one or more switches** connecting multiple local devices.
- Differently from routers, **switches are faster and plug-and-play**:
  - There is **no routing** algorithm involved.
  - **Only 2 layers** of the stack are considered.
- On the other hand, switched LANs are **limited in size** and must be **tree-structured**:
  - **MAC addresses are hard to group** (forwarding tables in switches may grow rapidly).
  - There is no routing, **loops are difficult to avoid**.



Link Layer
Address Resolution Protocol (ARP)

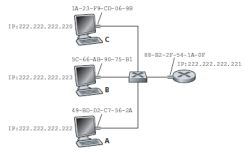
- Since upper-layer protocol works with IP addresses **we need to translate IP into MAC**.
- The **Address Resolution Protocol** (ARP) manages conversion between IP and MAC addresses.
- **Each interface is endowed with an ARP module having an ARP table** that associates each IP in the LAN to a MAC address **with a specific time to live** (TTL) value after which the entry is delated (typically 20 minutes).
- Since MAC addresses are local, also **ARP works only on local networks** (LANs).



IP Address	MAC Address	TTL
222.222.222.221	88-B2-2F-54-1A-0F	13:45:00
222.222.222.223	5C-66-AB-90-75-B1	13:52:00

Link Layer
Address Resolution Protocol (ARP)

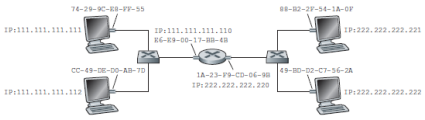
- Assume that **host C** (222.222.222.220) **wants to send messages to host A** (222.222.222.222). To do so, we need to know also the associated MAC.
- Before to send the message, if **A is not present into the table** an ARP packet is sent in **broadcast** to all devices of the network searching for the right IP.
- All nodes receive this packet but **only the searched IP** (222.222.222.222) **answers** with a direct message (not broadcast).



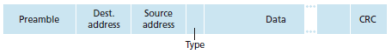
IP Address	MAC Address	TTL
222.222.222.221	88-B2-2F-54-1A-0F	13:45:00
222.222.222.223	5C-66-AB-90-75-B1	13:52:00

Link Layer
Address Resolution Protocol (ARP)

- What happens **if the IP is outside the network** (non-local)?
- The router connecting the 2 networks **must have at least 2 interfaces** (2 IP, MAC and ARP tables) each one inside the specific subnets.
- **Frames headed outside the subnet are sent to the first interface** of the router, moved to the second interface, **and headed toward the right host by using the second ARP table**.



Link Layer
Ethernet Frame



- **Ethernet** frame is composed by the following fields:
  - **Data field** (46 to 1500 bytes): **contains the IP datagram**. The maximum limit is given by the maximum transmission unit (MTU) of Ethernet, if datagram exceeds this size it is fragmented.
  - **Destination address** (6 bytes): contains **the MAC address of the destination** adapter.
  - **Source address** (6 bytes): contains the **MAC address of the source** adapter that transmits the frame onto the LAN.
  - **Type field** (2 bytes): specifies the **network-layer protocol** used for in this frame (there could be alternative to IP, for example, ARP packets have a specific type - 0x0806).
  - **CRC** (4 bytes): contains the **CRC number**.
  - **Preamble** (8 bytes): is a **"wake up" block of bits used to synchronize the clocks** of destination and source adapters.

Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

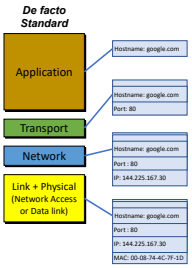
Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

DIETI UNINA

Stack Overview
There and Back Again

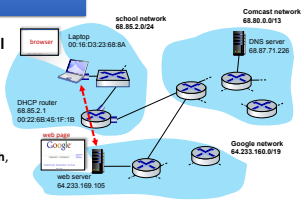
- As we have just seen, **several protocols in the TCP/IP stack collaborate** in allowing messages to travel back-and-forth a network.
- **Each layer-specific protocol adds a piece of information** to messages (through encapsulation and decapsulation) regulating one or more aspects of the communication (services offered).
- It is somehow difficult to get the "big picture" out of the single protocols, we will now recap the whole process by proposing a **web page request scenario**.



Example of the different addresses/identifiers considered in each layer of the stack.

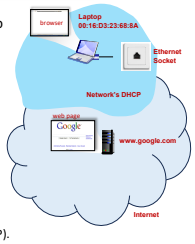
Stack Overview
Web Page Example

- Let's assume that **a user (Bob) wants to access the google.com web page from the institutional network** (school).
  - To do so, Bob **connects the laptop to the school's network through an Ethernet cable**.
- In this example we assume a **typical configuration** for the school's network:
  - Ethernet **sockets (on walls) are connected to a switch**, which is **connected to the school's DHCP-enabled router**.
  - The **school's router is connected to an ISP** (e.g., comcast.net) that provides also **DNS service**.
- For the sake of simplicity, we also assume there is **no NAT service** from the router, all **connections are Ethernet**, and **packets are never lost**.



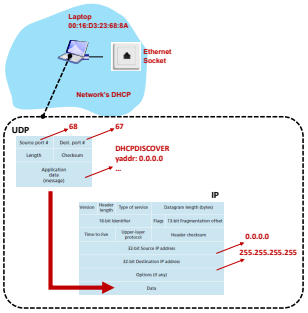
Stack Overview
Web Page Example

- On start, the **user knows very little about the network** topology, the way the LAN is connected to Internet, or the devices involved.
- What Bob knows is that:
  - **There is a www.google.com website somewhere on Internet** that Bob wants to reach.
  - **The local network (LAN) is somehow connected to Internet**.
  - The **laptop can be connected to the LAN** through an Ethernet cable (there is a socket on the wall).
  - The **LAN has an active DHCP**.
- **Once the laptop is physically connected** to the network (through the Ethernet cable) it has to:
  - **Join the network** (get network information from DHCP).
  - **Get the IP address of the website** (DNS).
  - **Get the Google's webpage** (HTTP).



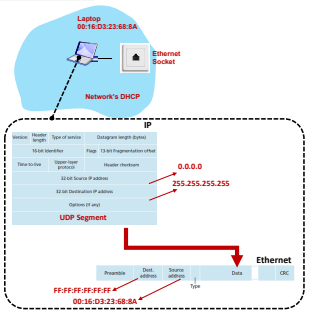
Stack Overview
DHCP

- The first step is for the laptop to join the network by **requesting a host IP, gateway and DNS** to the DHCP:
  1. The **operating system (OS) on the laptop creates a DHCP discovery message** and puts this message **within a UDP segment** with destination port 67 (DHCP server) and source port 68 (DHCP client).
     - The **UDP segment is then placed within an IP datagram** with a broadcast IP destination address (255.255.255.255) and a source IP address of 0.0.0.0 (no host IP yet).



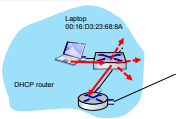
Stack Overview
DHCP

2. The **IP datagram** containing the DHCP discovery message **is then placed within an Ethernet frame** having FF:FF:FF:FF:FF:FF (broadcast) as destination MAC addresses (it will reach all devices on the switch and, hopefully, the DHCP) and laptop's MAC address 00:16:D3:23:68:8A as source.



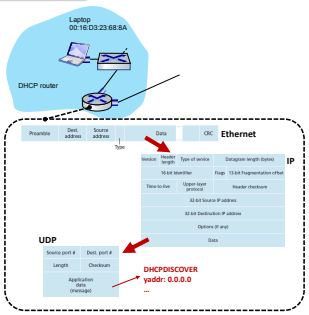
Stack Overview
DHCP

3. The **Ethernet frame containing the DHCP message is sent to the Ethernet switch**. The **switch broadcasts the frame** on all outgoing ports (including the port connected to the router).



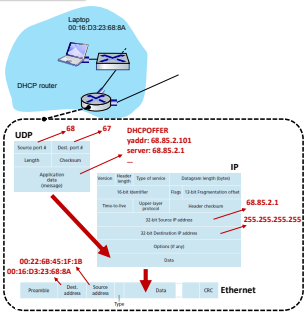
Stack Overview
DHCP

3. The **Ethernet frame containing the DHCP message is sent to the Ethernet switch**. The **switch broadcasts the frame** on all outgoing ports (including the port connected to the router).
4. The **router receives the Ethernet frame** containing the DHCP discovery on its interface (with MAC address 00:22:6B:45:1F:1B), **the message is decapsulated**:
   - The **frame is accepted because it has broadcast destination MAC** address, then the IP datagram is extracted.
   - The **datagram is accepted because it has broadcast IP** destination address, then the UDP segment is extracted.
   - The **UDP segment is demultiplexed** and the payload is received by the DHCP process on the router.



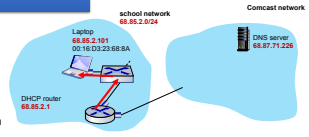
Stack Overview
DHCP

5. Let's now assume that **the DHCP router can allocate IP addresses in the CIDR block 68.85.2.0/24** (which is a subnetwork provided by the ISP):
   - The **DHCP server offers the IP** address 68.85.2.101 to the laptop.
   - The **DHCP server creates a DHCP offer message** containing:
     - The **offered IP address** and mask (68.85.2.101/24).
     - The **IP address of the DNS** server (68.87.71.226).
     - The **IP address for the gateway** (68.85.2.1).
   - The message is **encapsulated**:
     - A **UDP segment** is created with source port 67 and destination port 68.
     - The **segment is put into an IP datagram having broadcast as destination** address and 68.80.2.1 as source address.
     - The **segment is put into an Ethernet frame having 00:22:6B:45:1F:1B as source MAC** address (the router's LAN-interface) and **00:16:D3:23:68:8A as destination MAC** address (the laptop).



Stack Overview
DHCP

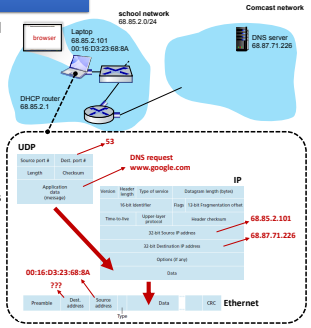
6. The Ethernet **frame containing the DHCP offer is sent** (unicast) by the router to the switch, which forwards it to the laptop checking the destination MAC address.
7. Bob's **laptop receives the Ethernet frame** with the DHCP offer and **decapsulates** it:
   - **Frame accepted** due to **correct MAC address**, then IP datagram is extracted.
   - **IP datagram accepted** due to **broadcast IP** address, then UDP segment is extracted.
   - DHCP **offer is demultiplexed** to the client DHCP process of the OS.
   - The **OS accepts the offer and sends back a DHCP request message** (skipped for brevity).
   - When a **DHCP ACK message is eventually received** from a new UDP segment, the **OS sets the received network information** (IP, gateway and DNS). Now the laptop has joined the network.



At the beginning, we only knew the MAC address of the laptop, **now we know**:
- The **IP of the laptop**.
- The **IP of the router** (gateway).
- The **network configuration** (mask).
- The **IP of the DNS**.

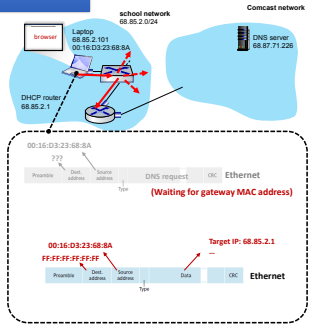
Stack Overview
DNS and ARP

- Now **the user decides to open the browser and to navigate to the Google's home page** (www.google.com), hence the IP of the web server must be retrieved from the DNS:
  8. The **browser invokes an OS routine to get the server's IP** (as in gethostbyname function):
     - The **OS creates a DNS query** message for the "www.google.com" hostname.
     - The **DNS message is encapsulated within a UDP segment** having 53 as destination port (DNS server).
     - The **UDP segment is placed within an IP datagram** having 68.87.71.226 (IP of DNS retrieved from DHCP) as destination address and 68.85.2.101 as source IP address (self).
  9. The IP datagram containing the **DNS query must be encapsulated into an Ethernet frame**. To do so **we need the MAC address of the gateway**, hence an **ARP query** must be created.
     - Note that even if **we know the IP of the gateway** from the DHCP, **we are not aware of the MAC** address.



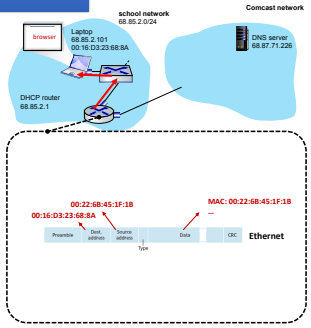
Stack Overview
ARP

10. The **OS creates an ARP query** frame having a **target IP address of 68.85.2.1** (the default gateway) and broadcast (FF:FF:FF:FF:FF:FF) as destination MAC address.
    - The **frame is sent to the switch**, which delivers it **to all connected devices**, including the **gateway router**.



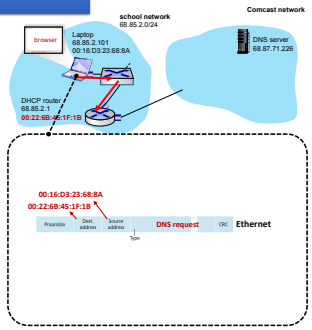
Stack Overview
ARP

11. The **router receives the frame** on its LAN-side interface and finds that the target IP address of 68.85.2.1 in the **ARP message matches the IP address of the interface**:
    - The gateway **router prepares an ARP reply**, indicating that that the MAC address 00:22:6B:45:1F:1B corresponds to the IP address 68.85.2.1.
    - The **ARP reply message is put into an Ethernet frame** having 00:16:D3:23:68:8A as destination address (laptop address).
    - The **frame is sent to the switch**, which delivers it **to the laptop**.



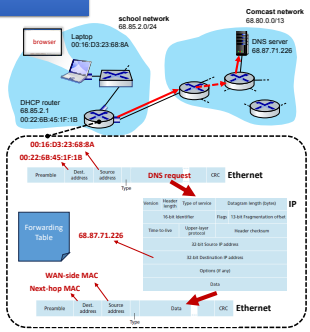
Stack Overview
ARP

12. The **laptop receives the frame** containing the ARP reply message and extracts the MAC address of the gateway router (00:22:6B:45:1F:1B).
13. The **laptop can now send the Ethernet frame containing the DNS query** to the gateway's MAC address.
    - Note that, in this case, the **IP datagram** will have a **destination IP address of 68.87.71.226** (the DNS server), and a **destination MAC address of 00:22:6B:45:1F:1B** (the gateway router).



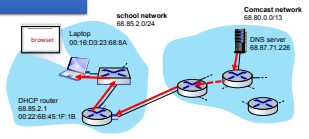
Stack Overview
DNS

14. The gateway **router receives the Ethernet frame** and decapsulates it:
    - The **router extracts the IP datagram** from the frame and retrieve the destination IP of the DNS.
    - The **router looks up the destination IP** (68.87.71.226) and determines from its forwarding table that the **datagram should be sent to the first router of the ISP network** (leftmost router in the Comcast network).
    - The IP **datagram is placed inside a link-layer frame** appropriate for the link connecting the school's router to the leftmost Comcast router and the frame is sent over this link.
15. The **first router of the ISP receives the frame** and extracts the IP datagram:
    - The **router checks the destination address** (68.87.71.226) and retrieves from the forwarding table (created through a link-state or distance-vector algorithm) the outgoing interface.
    - A **new frame is then created and sent to the next router** through the selected interface.
    - **This process may be iterated** several times before the DNS server is reached.



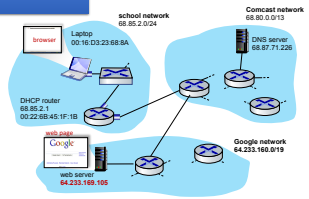
Stack Overview
DNS

16. Eventually the IP datagram containing **the DNS query arrives at the DNS** server:
    - The DNS server extracts the DNS query message, **looks up the name www.google.com from its database** and finds the IP address 64.233.169.105 (Google's server).
      - Notice **that here we are assuming a cached IP**, otherwise additional requests to authoritative servers should be sent.
    - The DNS **server creates a DNS reply message** containing the retrieved IP address, and places it **in a UDP segment**.
    - The **segment is put in an IP datagram** having destination IP of 68.85.2.101 (the laptop) and then into the appropriate frame.
    - This **message will be forwarded back** through the Comcast network to the school's router and from there, via the switch to the laptop.



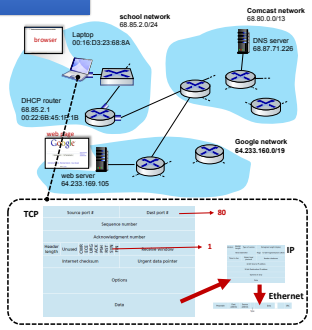
Stack Overview
DNS

16. Eventually the IP datagram containing **the DNS query arrives at the DNS** server:
    - The DNS server extracts the DNS query message, **looks up the name www.google.com from its database** and finds the IP address 64.233.169.105 (Google's server).
      - Notice **that here we are assuming a cached IP**, otherwise additional requests to authoritative servers should be sent.
    - The DNS **server creates a DNS reply message** containing the retrieved IP address, and places it **in a UDP segment**.
    - The **segment is put in an IP datagram** having destination IP of 68.85.2.101 (the laptop) and then into the appropriate frame.
    - This **message will be forwarded back** through the Comcast network to the school's router and from there, via the switch to the laptop.
17. The **OS of the laptop extracts the target IP** address from the DNS message. **Now we know how to reach the Google web server.**



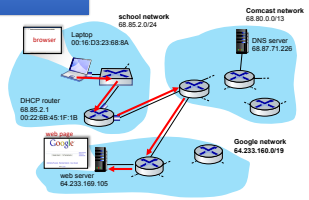
Stack Overview
HTTP

- Knowing the IP of the server **we can now create a HTTP GET request** for the google home page:
  18. The **browser creates a TCP socket**, so the **OS performs a three-way handshake** with the Google web server:
      - The **OS creates a TCP SYN segment** with destination port 80 (HTTP).
      - The **segment is encapsulated inside an IP datagram** having a destination IP address of 64.233.169.105 (www.google .com).
      - The **datagram is put inside a frame** with a destination MAC address of 00:22:6B:45:1F:1B (the gateway router) and sent to the switch.



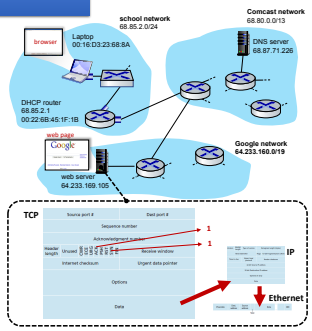
Stack Overview
HTTP

- Knowing the IP of the server **we can now create a HTTP GET request** for the google home page:
  18. The **browser creates a TCP socket**, so the **OS performs a three-way handshake** with the Google web server:
      - The **OS creates a TCP SYN segment** with destination port 80 (HTTP).
      - The **segment is encapsulated inside an IP datagram** having a destination IP address of 64.233.169.105 (www.google .com).
      - The **datagram is put inside a frame** with a destination MAC address of 00:22:6B:45:1F:1B (the gateway router) and sent to the switch.
  19. All **routers (from local, ISP, and Google networks) forward the datagram** to the web server using their forwarding table.
      - Notice that **frames can be modified along the path depending on the specific links**.



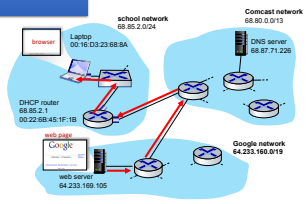
Stack Overview
HTTP

20. The frame containing the **SYN segment eventually arrives at the server**:
    - The **datagram is extracted** from the frame.
    - The **segment is extracted** from the datagram and **demultiplexed** to the welcome socket associated with port 80.
    - A **connection-specific socket is created** between the Google web server and the laptop.
    - A **TCP SYNACK segment is generated**, placed inside a datagram having destination address of 68.85.2.101 (laptop).
    - The **segment is placed into an appropriate link-layer frame** to travel the link connecting www.google.com to its first-hop router.



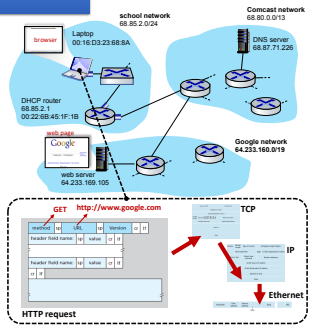
Stack Overview
HTTP

20. The frame containing the **SYN segment eventually arrives at the server**:
    - The **datagram is extracted** from the frame.
    - The **segment is extracted** from the datagram and **demultiplexed** to the welcome socket associated with port 80.
    - A **connection-specific socket is created** between the Google web server and the laptop.
    - A **TCP SYNACK segment is generated**, placed inside a datagram having destination address of 68.85.2.101 (laptop).
    - The **segment is placed into an appropriate link-layer frame** to travel the link connecting www.google.com to its first-hop router.
21. The **TCP SYNACK eventually arrives at the laptop**:
    - The **datagram is demultiplexed** by the OS to the TCP socket created in step 18.
    - The **socket enters the "connection established" state**.



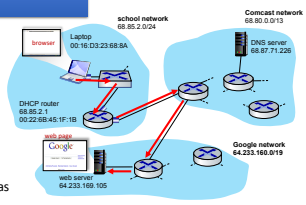
Stack Overview
HTTP

22. The **socket on the laptop now ready to send bytes** to web server:
    - The **browser creates the HTTP GET message** containing the URL to be fetched.
    - The **message is written into the socket**, and the GET request becomes the **payload of a TCP segment**.
    - The TCP **segment is placed in a datagram and then into a frame** (encapsulation).



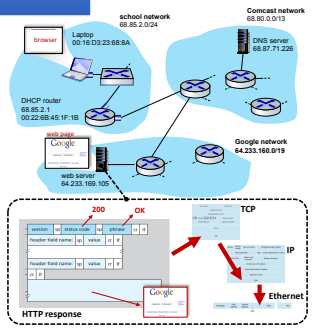
Stack Overview
HTTP

22. The **socket on the laptop now ready to send bytes** to web server:
    - The **browser creates the HTTP GET message** containing the URL to be fetched.
    - The **message is written into the socket**, and the GET request becomes the **payload of a TCP segment**.
    - The TCP **segment is placed in a datagram and then into a frame** (encapsulation).
    - The **message is delivered to www.google.com** (as in steps 18-20).



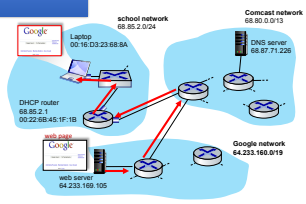
Stack Overview
HTTP

23. The Web **server at www.google.com receives the GET message** from the TCP socket:
    - The **message is decapsulated and demultiplexed**, then the **GET request is interpreted**.
    - The **server creates an HTTP response message**, having **the Google home page in the body**.
    - The **message is written into the TCP socket**.
    - The **server's OS encapsulates the message** into a segment, a datagram, and finally a frame.



Stack Overview
HTTP

24. The datagram containing the **HTTP reply message is forwarded through the 3 networks**, and arrives at the laptop:
    - The **laptop's OS decapsulates the message**, which is **demultiplexed to the browser**.
    - The **browser reads the HTTP response** from the socket.
    - The **browser extracts the html code from the body** of the HTTP response
    - The **browser finally displays the Web page**!



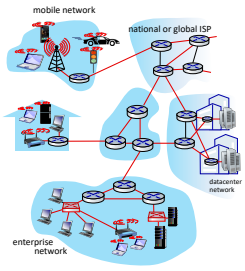
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Network Security
Introduction

- The **network security** is the field that studies **possible attacks** to networks and **possible ways to prevent them**.
- Networking is part of our lives (Internet connection is considered almost as water or power supply) and **a large quantity of our sensible data travels on networks**.
- There are **several types of attacks** having different purposes and different mechanics.
  - Attacks evolve along with the **technology** and the networks' **audience**.



Network Security
Attacks: Malwares

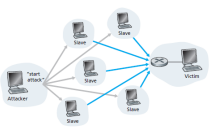
- A **malware** is **malicious software** that can be transferred to a computer through the network (e.g., downloaded file, e-mail attachment, etc.).
- Once malware infects our device it can harm in different ways:
  - Forcing a system to show commercial advertisements (**adware**).
  - Showing false alarm messages to induce users to download malwares (**scareware**).
  - Deleting (**wiper**) or encrypting (**ransomware**) our files.
  - Collecting private information such as passwords, security numbers, etc. (**spyware**).
    - For example, **keyloggers** are software recording (logging) the keys struck on a keyboard.
  - Getting root privileges of our system (**rootkits**).
  - Turn a device into a slave or foothold to attack other devices (**zombie** or **botnet**).

Network Security
Attacks: Malwares

- Todays malware are often **self-replicating**: once it infects one host, from that host **it seeks entry into other hosts over the Internet**, and from the newly infected hosts, it seeks entry into yet more hosts.
- Malware can spread in the form of a virus or a worm:
  - **Viruses** are malware that **require some form of user interaction** to infect the user's device.
    - For example, an **e-mail attachment** containing malicious executable code that self-replicates by sending similar mails to users' contacts.
    - These are typically **disguised as legitimate software** components (**trojan horses**).
  - **Worms** are malware that **can enter a device without any explicit user interaction**.
    - For example, a user may be running **a vulnerable network application to which accepts the worm without intervention**. The worm than scans the network for hosts running a similar application.

Network Security
Attacks: DoS

- **Denial-of-Service** (DoS) attacks are quite common and are designed to render a network, a host, or other piece of infrastructure (e.g., Web servers, DNS, etc.) unusable by legitimate users.
- There are 3 types of DoS attacks:
  1. **Vulnerability attack**: sending suitable messages to vulnerable applications or OSs in order to let them stop or crash.
  2. **Bandwidth flooding**: sending a large quantity of packets to the targeted host, preventing legitimate packets from reaching the server.
  3. **Connection flooding**: establishing a large number of half-open or fully open TCP connections at the target host, so it stops accepting legitimate connections.



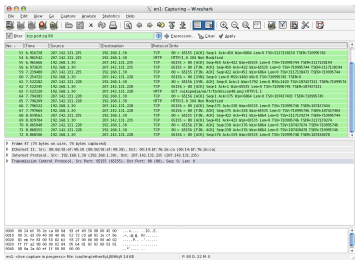
Example of a **DDoS** (Distributed DoS) attack using a **botnet** of zombies (or slaves).

Network Security
Attacks: Packets Sniffing

- **Packets sniffing** involves a passive receiver (**sniffer**) that records a copy of relevant packets from a target host trying to steal sensitive information (aka **eavesdropping**).
  - Sniffers can be deployed in all kind of **broadcast** networks (wired or wireless) simply by copying packets that are meant for different destinations instead of discarding them.
  - As for **non-broadcast** networks, a sniffer can be put into a malware (spyware) and used to infect network devices (e.g., routers) so that all forwarded traffic is also copied.
- Since sniffers are **passive** (no additional traffic is injected into the network) **these are very difficult to detect**.
- To prevent sniffing **cryptography** approaches can be used.

Network Security
Attacks: Packets Sniffing

- There are several sniffers freely available on Internet. A notable example of packet sniffer is **Wireshark** (ex **Ethereal**).
- Wireshark is a **packet/protocol analyzer**, it is mainly used for legitimate purposes (troubleshooting, network monitoring, creation of new protocols, etc.).
- Wireshark is available in different OSs (Linux included).



GUI of Wireshark from Wikipedia

Network Security
Attacks: IP Spoofing

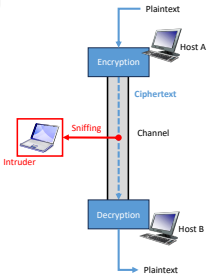
- **IP spoofing** is a technique that allow malevolent hosts to inject into a network packets with false source addresses.
  - It can be used **in combination with applications vulnerability** to attack specific hosts being masqueraded as another user.
  - It can also be used **for DoS attacks** (alternative to botnets) as messages from different source IPs are **more difficult to filter**.
- Spoofing can also be used for **man-in-the-middle** (MitM or MiM) attacks, in which the attacker is placed in between 2 communicating hosts disguised as both.
  - The 2 hosts think they are communicating each other, while **they are actually communicating with the attacker**.
- To prevent spoofing, we can use **message integrity checks** and **end-point authentication**, allowing us to determine if the message has not been modified or if the message originates from the right source.

Network Security
Basics of Network Security

- Given the previous attack types we can now define the set of proprieties that a **secure communication** should guarantee:
  - **Confidentiality**: only the sender and intended receiver should be able to understand the contents of the transmitted message (sniffing avoidance).
  - **Message integrity**: the content of the communication must not be altered, either maliciously or by accident.
  - **End-point authentication**: both the sender and receiver should be able to confirm the identity of the other party involved in the communication (spoofing avoidance).
  - **Operational security**: to rely on a network infrastructure that prevents malicious hosts to sneak into the communication.
- The **first 3 proprieties are software-based** the last one (operational security) typically relies on **specific hardware** (firewalls, intrusion detection systems).

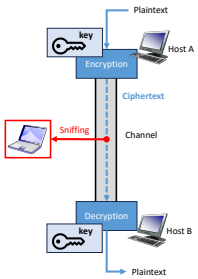
Network Security
Cryptography

- A **cryptographic technique** allows a sender to **disguise data** so that it becomes incomprehensible for an intruder.
  - **Intruders can gain no information** from the intercepted data, but the **receiver must be able to recover the original data** from the disguised data.
- In its initial form the message is called **plaintext** (or cleartext) and is readable for everyone.
- Before to inject the message into the channel a host uses an encryption algorithm to transform the message into a non-readable form called **ciphertext** which must be decrypted when received.



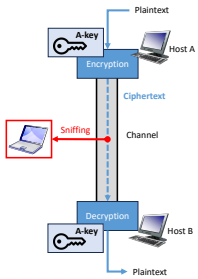
Network Security
Cryptography: Keys

- In many modern cryptographic systems, including those used on the Internet, the **encryption technique is known and standard for everyone** (including the intruder). The **unknown part** of the algorithm are the **encryption/decryption keys**.
- A **key** is an alphanumerical string that must be provided to the encryption/decryption algorithm in order to encrypt/decrypt the messages.
- Encryption and decryption keys can be **identical** (symmetric) or **different** (asymmetric).



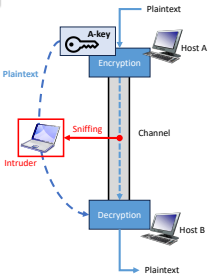
Network Security
Cryptography: Symmetric

- In **symmetric cryptography** there is **only one key** that is used for both encryption and decryption.
- **Encryption**: the plaintext message along with the key is passed to the encryption algorithm to generate a ciphertext that can be safely sent through the network.
- **Decryption**: the cyphertext message along with the key is passed to the decryption algorithm to recreate the initial plaintext message (readable).



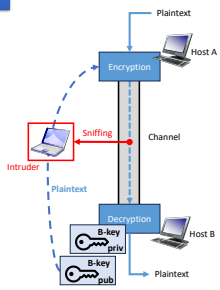
Network Security
Cryptography: Key Exchange

- The **problem** with symmetrical cryptography, or single-key cryptography, is that it requires the secret key to be communicated (**key exchange problem**).
  - Hosts can use a **secure channel** to exchange the key.
  - Hosts can use some **protocol** that allows them to "converge" on a shared key.
- If two parties **cannot establish a secure initial** key exchange, **they won't be able to communicate securely** without the risk of messages being intercepted and decrypted by a third party who acquired the key during the initial key exchange.



Network Security
Cryptography: Asymmetric

- In **asymmetric cryptography** there is a **two-key system** (public and private keys).
  - A message that is **encrypted with one key must be decrypted with the other** and vice versa.
- The idea is that the **public key can be sent over non-secure channels** or shared in public, while the private key is only available to its owner.
- A typical approach is to use **public key for encryption and private key for decryption**:
  - If the intruder sniffs the public key, it is still impossible for him/her to decrypt messages.
  - Host A will use B-key-public to encrypt messages that can only be seen through B-key-private, which is only into B's hands.



Network Security
Certification Authority

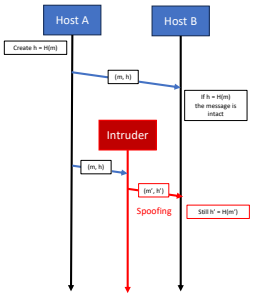
- In public key cryptography it could be useful to **verify if a public key really belong to the entity** with whom you want to communicate.
  - Otherwise, **we could have someone's else key** (attacker) and we could encrypt messages that are readable by illegitimate entities.
- Binding a public key to a particular entity is typically done by a **Certification Authority** (CA), whose job is to validate identities and issue certificates. A CA has the following roles:
  1. A **CA verifies that an entity is who it says it is**. There is no protocol for that, **one must trust the CA** to have performed a suitably rigorous identity verification.
     - It works like a natural selection process: **if a CA is unreliable no on will trust it**.
     - There are **several federal or statal CA** that provide a reasonable reliability, but we still have to trust them.
  2. Once the CA verifies the identity of the entity, **the CA creates a certificate that binds the public key of the entity to the identity**. The certificate contains the public key and a globally unique identifier of the owner (for example**, a name or an IP address**).

Network Security
Message Integrity

- **Message integrity** (also known as **message authentication**) is the problem of checking if:
  1. The message has **not been tempered with**.
  2. The message has been indeed **originated by the expected host**.
- We can create a **check-item** similarly as the checksum or CRC. Typically, a hash function is used to create such item.
  - Remind: a **hash function** is any function that can be used to **map data of arbitrary size into fixed-size values**.
- A **cryptographic hash function** is a function *H* that converts a message *x* into a fixed-size string *H(x)* so that it is very hard (computationally infeasible) to find another message *y* such that *H(y) = H(x)*.

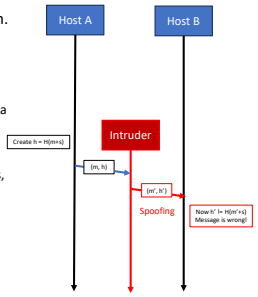
Network Security
Message Integrity

- As in checksum or CRC, **we can attach this hash into the message**:
  1. Host **A creates message m and calculates the hash h = H(m)**.
  2. Host **A appends h to the message m**, creating an extended message (m, h), and **sends the extended message to B**.
  3. Host **B receives the message (m, h) and calculates H(m)**. If H(m) = h, the message is intact.
- **This approach is obviously flawed**. An intruder may **spoof the whole message** (m,h), creating **a new "ad hoc" one** (m',h') that is still consistent with the hashing function H.



Network Security
Message Integrity

- To avoid this, A and B need a **shared secret** *s* (a **shared key or a password**) which is a string known only to them.
  - This basically **works as a symmetric encryption** where *s* is the unique private key.
- Assuming such *s* exists then:
  1. Host **A creates a message m + s** (as a concatenation of message and secret) and **calculates the hash h = H(m + s)** aka a **message authentication code (MAC)**.
  2. Host **A appends the MAC to the message m**, creating an extended message (m, H(m + s)), and sends the extended message to B.
  3. Host **B receives the extended message** (m, h) and knowing s, **calculates the MAC H(m + s)**. If H(m + s) = h, the message is intact.
- As in all symmetric approaches, also here **we need to exchange such secret**.
  - This **secret can be exchanged combining asymmetric cryptography and certificates**.



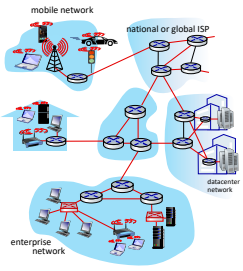
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

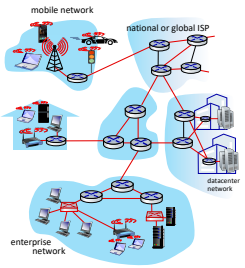
Network Programming
Introduction

- Network programming is the act of **creating network applications** that run on multiple, perhaps different, devices.
  - **Network applications** may be based on **different programming languages, code libraries, or OSs**.
  - **Devices** involved in a network applications **can be quite heterogeneous** (they may have different hardware, architecture, etc.).
- Such applications **rely on the network infrastructure** to communicate.
- Fortunately, the network infrastructure is **standardized**, we no need to worry about networking as most of the processes are "hidden" to us and are **managed (back-box) by sockets**.

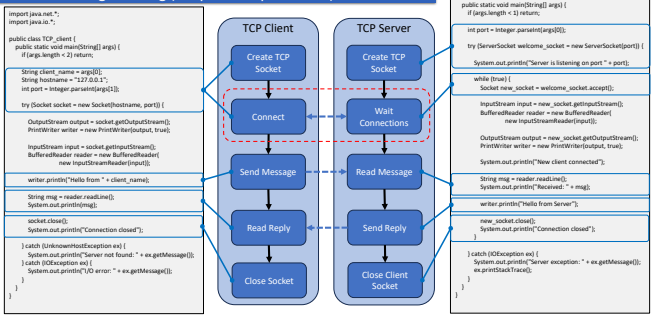


Network Programming
Introduction

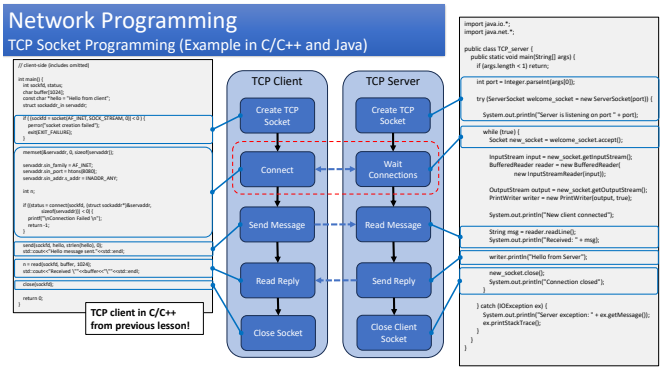
- **Sockets are widely available** in different programming languages and OSs.
- Network programming often requires skills in **multiple programming areas and tools**:
  - Different programming languages.
    - Java, Python, C/C++, etc.
  - Operating Systems.
    - Linux, Windows, etc.
  - ISO/OSI stack and protocols.
    - HTTP, TCP, UDP, etc.
  - Data exchange formats.
    - JSON, YAML, XML, etc.
  - REST APIs.
  - ...



Network Programming
TCP Socket Programming (Simple Example in Java)



Network Programming
TCP Socket Programming (Example in C/C++ and Java)



Network Programming
Data Exchange

- In the previous examples we have just seen simple programs **exchanging strings** (plain text), but this is hardly a realistic case.
- Real network applications typically **exchange structured data** (structures, lists, etc.) so, a more complex representation of the information could be useful.
- Several **data exchange** (or interchange) **languages** (or formats) have been used in the years to provide **a standardized way of representing and exchanging structured data** for network applications.
- Because of their human-readability, **most common** (and open) formats used for data exchange are:
  - XML.
  - YAML.
  - JSON.

Network Programming
Data Exchange: XML

- The **XML** (eXtensible Markup Language) is a language designed to be **easily understandable by both humans and computers** (similar to HTML).
  - Typical file extension: .xml
- The core concept of XML are the **tags**. A **tag** is a markup construct wrapped in angle brackets (<…>), each content in an XML file is surrounded by the **start-tag** (<TAGNAME>) and the **end-tag** (</TAGNAME>).
  - Contents inside tags may be **simple values** (string, numbers, bool, etc.) or **other tags**.
- Differently from HTML, in XML **few tags are pre-defined**, we may **create tags at our discretion**.
  - XML documents are **conventionally** wrapped within <xml> ... </xml> tags, but these are rarely implemented for data exchange.

Network Programming
Data Exchange: XML

- This is a simple example of how to represent data about a person. Here we have:
  - A **root tag** <person> that collects the data about one person. It contains 3 inner tags:
    - The tag <personID> specifies the **ID of the person** (e.g., of a database) and contains a number.
    - The tag <firstName> contains a string with the **name of the person**.
    - The tag <lastName> contains a string with the **surname of the person**.
- It is important to see that **all contents** (no matter how trivial) are wrapped inside start-tag and end-tag.

```
<person>
   <personID> 77 </personID>
   <firstName> John </firstName>
   <lastName> Doe </lastName>
</person>
```

Network Programming
Data Exchange: XML

- It could be uncomfortable to specify all contents within tags. XML offers the possibility to **specify attributes and self-closing tags** to simplify content definition.
- A **self-closing tag** is a **standalone tag** that closes itself:

  <TAGNAME />
- The **attributes** are **name-value pairs** that exist **within a start-tag or self-closing tag**. The value of attributes must be inside double quote ("):

Start-tag with attribute:
```
<TAGNAME ATT_NAME = "ATT_VALUE" >
...
</TAGNAME>
```
or

Self-closing tag with attribute:
```
<TAGNAME ATT_NAME = "ATT_VALUE" />
```

Network Programming
Data Exchange: XML

- Following the previous example, we can use attributes and self-closing tags to represent a person in different ways:
  - Inserting some attributes into the start-tag.
  - Inserting all attributes into a self-closing tag.

```
<person personID = "77" >
   <firstName> John </firstName>
   <lastName> Doe </lastName>
</person>
```

```
<person personID = "77" firstName = "John" lastName = "Doe" />
```

Network Programming
Data Exchange: XML

- We can specify a group of people by nesting multiple person into an additional <people> tag.

```
<people>
   <person personID = "77" >
      <firstName> John </firstName>
      <lastName> Doe </lastName>
   </person>
   <person personID = "78" >
      <firstName> Alice </firstName>
      <lastName> Doe </lastName>
   </person>
</people>
```

Network Programming
Data Exchange: YAML

- The **YAML** (Yaml Ain't Markup Language) is a language designed to be **minimal and human-readable**.
  - Typical file extension: .yaml or .yml
- Differently from XML, in YAML **contents are defined through spaces and indentation** (as in python).
  - Note: **tab-based indentation is not allowed**, only whitespaces must be used.
- In YAML **names and values are separated by column** (NAME: VALUE) and there is a specific syntax to define elements such as **structures, lists or dictionaries**.

Network Programming
Data Exchange: YAML

- This is a simple example of how to represent data about a person. Here we have:
  - A "person" **structure** that contains 3 inner fields:
    - The field "personID" specifies the **ID of the person** (e.g., of a database) as a number.
    - The field "firstName" contains a string with the **name of the person**.
    - The field "lastName" contains a string with the **surname of the person**.
- It is possible to notice that **quotes are not necessary** to discriminate values.

```
person:
   personID: 77
   firstName: John
   lastName: Doe
```

Network Programming
Data Exchange: YAML

- The YAML we can also represent **lists** and **dictionaries**.
- A **list** can be represented in 2 ways:
  1. As a **structure whose elements are preceded by a dash** (-).
  2. As a sequence of comma-separated **elements inside squared brackets** ([E1, E2, …]).
- A **dictionary** is represented as a sequence of **comma-separated name-value couples inside curly brackets** ({ N1: V1, N2: V2, … }).

Network Programming
Data Exchange: YAML

- Extending the previous example, we can define a **list of people containing 2 person structures**.
- We can define a **list of names**.
- We can define **fields of a structure** (the person in this case) **as a dictionary**.
  - This method is **widely used for data exchange**.

```
people:
  - person:
      personID: 77
      firstName: John
      lastName: Doe
  - person:
      personID: 78
      firstName: Alice
      lastName: Doe
```

```
names: [John, Alice, Bob]
```

```
person: {personID: 77, firstName: John, lastName: Doe}
```

Network Programming
Data Exchange: JSON

- The **JSON** (JavaScript Object Notation) is a simple format for data exchange which is **human-readable**, **easily parsed** by machines, and is **based on C-like conventions** for data representation.
  - Typical file extension: .json
- JSON is somehow a **tradeoff between simplicity and effectiveness**, it is considered an **ideal data-interchange language** (one of the most used).
  - C-like convention is also well-known by programmers.

Network Programming
Data Exchange: JSON

- As in YAML, **JSON data** is specified as **a pair of name and value divided by a colon** (:) symbol where the name is a string (inside double quotes ""):

  "DATANAME": DATAVALUE

- **JSON objects** are **groups of data inside curly brackets** where data items are separated by commas ({ "N1": V1, "N2": V2, … }).
- **JSON arrays** are represented as comma-separated **data or objects** within **square brackets** (["N1": V1, "N2": V2, …]).

Network Programming
Data Exchange: JSON

- Differently from YAML, the **values for JSON data** have c-like syntax for different types. Possible values are:
  - **String** (inside double quotes):
    - "name" : "Bob"
  - **Number** (integer, float, double):
    - "age" : 27
    - "weight" : 60.5
  - **Array** (containing generic JSON data):
    - "pets" : ["cat", "dog"]
    - "siblings" : [ ]
  - **Boolean** (true/false):
    - "isAlive" : true
  - **Null** (e.g., not available):
    - "phoneNumber" : null
  - **JSON object**.

Network Programming
Data Exchange: JSON

- This is an example of how to represent data about a person. Here we have:
  - A **root object** that contains the "person" data, whose value is an additional JSON object having 3 fields:
    - The "personID" specifies the **ID of the person** (e.g., of a database) and contains a number.
    - The "firstName" contains a string with the **name of the person**.
    - The "lastName" contains a string with the **surname of the person**.
- In this case the **indentation is not necessary** (as in C), but it is just used for a better visualization.

```
{
  "person": {
    "personID": 77,
    "firstName": "John",
    "lastName": "Doe"
  }
}
```

Network Programming
Data Exchange: JSON

- Also in this case, we can define a list of people by putting multiple JSON objects within a JSON array.

```
{
  "people": [
    "person": {
      "personID": 77,
      "firstName": "John",
      "lastName": "Doe"
    },
    "person": {
      "personID": 78,
      "firstName": "Alice",
      "lastName": "Doe"
    }
  ]
}
```

Network Programming
Data Exchange: JSON

- Also in this case, we can define a list of people by putting multiple JSON objects within a JSON array.
- We may also simplify the syntax by removing the "person" data.
  - Notice that JSON array may contain JSON data as well as JSON objects.

```
{
  "people": [
    {
      "personID": 77,
      "firstName": "John",
      "lastName": "Doe"
    },
    {
      "personID": 78,
      "firstName": "Alice",
      "lastName": "Doe"
    }
  ]
}
```

Network Programming
JSON Examples (Java and C++)

Outsourced…

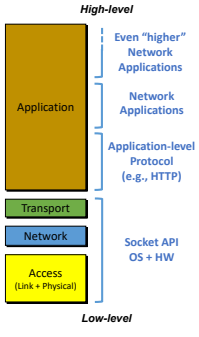
Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

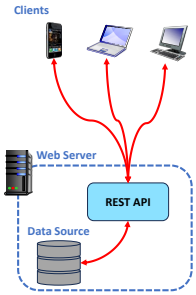
Network Programming
Designing Network Applications

- We can design **applications that use UDP or TCP** transport protocols but for the application-level protocol we can choose between 2 alternatives:
  - **Standard protocol**.
  - **Proprietary (custom) protocol**.
- A possible approach (quite common) is to design network applications "on top" of the **standard HTTP protocol** (as web-based applications):
  - **HTTP is well known**: several network elements already works with HTTP (servers, browsers, etc.) and many programming languages already have HTTP-related APIs.
  - **HTTP is versatile**: several functions can be implemented through a **combination of standard HTTP methods** and **header lines**.



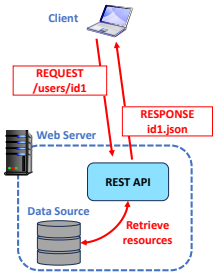
Network Programming
REST

- **REST** (Representational State Transfer) is a **set of architectural principles** that guides programmer in defining **modular, scalable and flexible web-based applications**.
- Web applications following such principles are typically called **RESTful APIs**.
- In theory, **REST principles are not tied to specific standards, programming languages, or technologies**, but these are strongly related to HTTP.



Network Programming
REST Architectural Elements

- A **REST API works as an interface** between multiple (and different) **clients** and the shared **resources** (e.g., a database).
- The **main REST elements** to be identified are:
  - **Resources**: the **pieces of information** that hosts are willing to manipulate. Resources are **identified by URIs** (Uniform Resource Identifiers).
    - We can see **URI as a generalized form of URL** that also include resource names.
  - **Representations**: how resources are structured and represented in a **specific format** (e.g., plain text, XML, YAML, JSON, etc.).

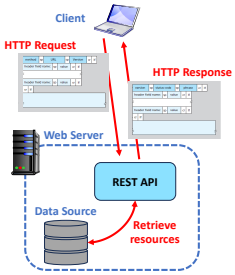


Network Programming
The Six REST Principles

1. **Client-server**. REST rely on client-server applications where the two **hosts are independent**:
   - **Client** application **only knows the URI** of resources to be requested.
   - **Server** application only **passes resources** via HTTP.
2. **Statelessness**. The state of the conversation is not maintained, each request must be self-contained.
3. **Cacheability**. Resources **should be cacheable** on the client or server side.
   - Responses should also contain information about whether caching is allowed or not for a specific resource.
4. **Uniform interface**. All API requests for the **same resource should look the same**, no matter where the request comes from.
5. **Layered system architecture**. RESTful APIs should consider that **messages can be forwarded through different intermediaries** (there is no direct link between client and server), but this should not be perceived by the client nor the server.
6. **Code on demand** (optional). Responses can also contain **executable code** (such as Java applets). In these cases, the code should only **run on-demand**.

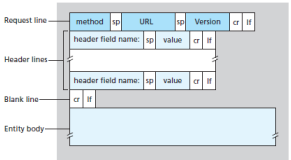
Network Programming
REST API in Practice

- It is possible to notice that **REST principles closely resemble some HTTP features**. In a way, REST defines how to "boldly" use HTTP in a network application.
- The general idea is to manipulate standardly-represented resources through standardly-represented messages.
- We rely on **HTTP** to implement client/server communication:
  - The client access resources through HTTP **requests**.
  - The server provides answers through HTTP **responses**.



Network Programming
HTTP remind

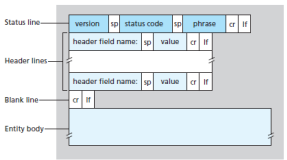
- The **request message** format include the following fields:
  - The **method**: specifies the requested command to be executed by the server.
  - The **URL**: is used to identify the object on which we want to operate.
  - The **version**: specifies the HTTP version (e.g., HTTP/1.1).
  - The **header lines**: contain the parameters of the request, the number and the type of these lines are not fixed. Each line include the **name** and the **value** of the parameter.
  - The **body**: is method-specific and contains data that are potentially associated with the command.



The fields are separated by special characters:
- The sp is space character.
- The cr is carriage return (\r).
- The lf is line feed (\n).

Network Programming
HTTP remind

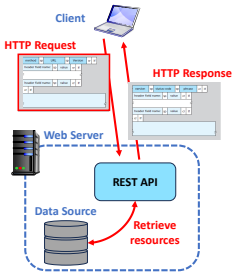
- The **response format message** is similar to the request one.
- Instead of a request line, we have a **status line** that reports the outcome of the command that includes:
  - The **version**: reports the HTTP version of the server's response.
  - The **status code**: a code (number) that specifies the outcome of the command.
  - The **phrase**: contains the result of the request.



The fields are separated by special characters:
- The sp is space character.
- The cr is carriage return (\r).
- The lf is line feed (\n).

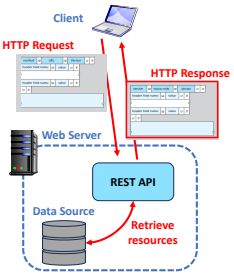
Network Programming
REST API in Practice

- HTTP **requests** in REST:
  - Resources can be **identified** through the **[URL]** field of the request.
  - The **operations on resources** can be specified into the **[Method]** field of a request.
    - **CRUD** (Create, Read, Update, Delete) **operations are typically considered**, which can be implemented through the 4 HTTP methods:
      - **Get**: retrieve the resource (read).
      - **Post**: modify the resource (update).
      - **Put**: add a new resource (create).
      - **Delete**: remove a resource (delete).
  - Possible **resource and options** can be specified into the **[Body] and [Header]** respectively.



Network Programming
REST API in Practice

- HTTP **response** in REST:
  - The **information about the request** are provided via the **[Status code]** and **[Phrase]** fields. Commonly used messages are:
    - 200 for successfully handled resource,
    - 201 for successful creation of a new resource,
    - 404 for resource not found,
    - 405 for method not supported,
    - etc.
  - Possible **resource and options** can be specified into the **[Body] and [Header]** respectively.



Network Programming
Resource Representation

- **HTTP URL is used to identify resources**. Common formats are:
  - [http(s)]://[Domain name of REST API][:Port]/[API version]/[Path to resource]
    - Example: http://myapi.example.com/v1/users/1
  - [http(s)]://[Domain name][:Port]/[REST API]/[API version]/[Path to resource]
    - Example: http://example.com/myapi/v1/users/1
- Paths for **resources and sub-resources** are commonly specified using the following pattern:
  - /[resource name]/[resource id]/[sub-resource name]/[sub-resource id] ..
    - Example: /users/1/nickname
- Resources can be **represented in different data exchange formats**. Most common is JSON, but **multiple languages can be used contemporary**.
  - For example, selected **format can be specified** by the "Content-type" header line.

Network Programming
REST Example (Java)

Outsourced…

Network Programming
Guidelines

- Resource naming and usage **guidelines**:
  - Use **plural nouns not verbs**:
    - Good: /users
    - Bad: /doSomethingOnUsers
  - Use **HTTP methods instead of CRUD terms** (or similar) into names:
    - Good: /users (with GET method)
    - Bad: /getAllUsers
  - Identify **unique resources** with IDs:
    - Good: /users/1
    - Bad: /users?id=1
  - You may use **queries** to select or sort resources:
    - Good: /users?nickname=Bob&sort=age
    - Bad: /usersNamedBobSortedByAge