UD 3 – Comunicaciones en Red

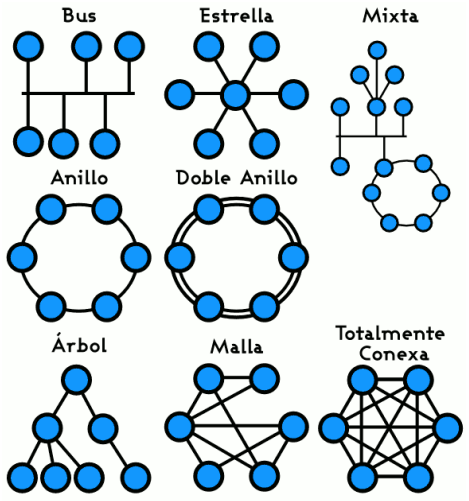
Programación de Servicios y Procesos

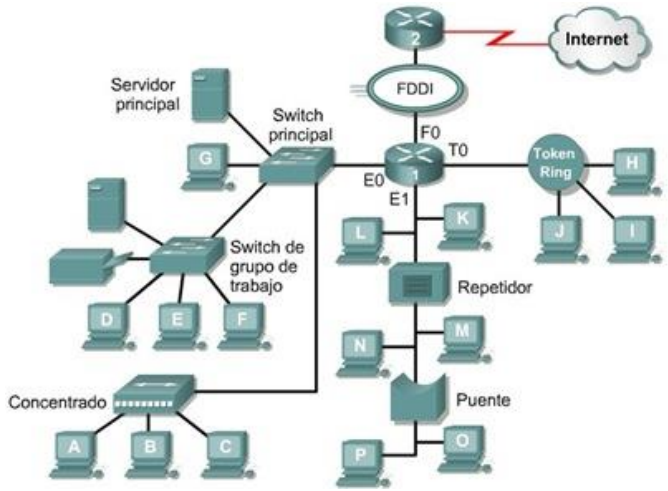
Curso 2024-25

Profesor: Agustín González-Quel

Conceptos fundamentales de Redes

- Red de ordenadores: Conjunto de ordenadores conectados entre sí.
- Objetivo: transmitir y compartir información.
- Tipos de redes
  - LAN, WAN
  - Privadas y públicas
  - Topologías: Cliente Servidor / Estrella, etc.
  - Cable, inalámbricas
  - …

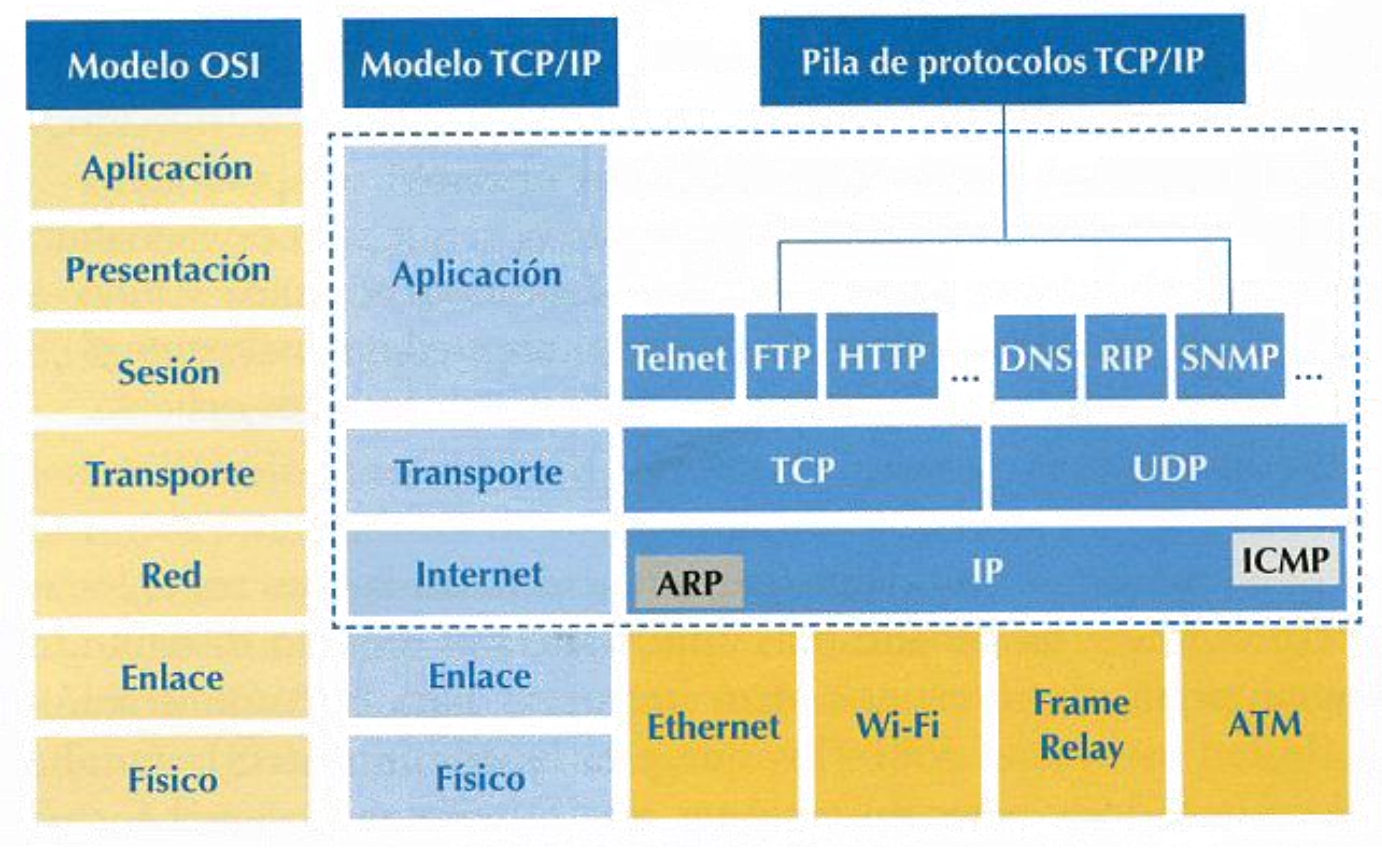




- Nomenclatura
  - Servidor: hardware o software que proporciona funcionalidad a otros programas o dispositivos, llamados "clientes".
  - Cliente: Programa o dispositivo que hace uso de la funcionalidad ofrecida por un servidor.
  - Dirección IP: identificador de un equipo en una red tipo TCP/IP
  - Puerto: número asignado para identificar de forma exclusiva un punto de conexión y para dirigir los datos a un servicio específico.
  - Protocolo: conjunto de reglas que definen la comunicación entre dos partes.

Protocolos OSI vs Protocolos TCP/IP

- El modelo teórico OSI ha derivado en el modelo TCP-IP.
- Pasa de 7 a 4 niveles
- Protocolos de nivel Sesión
  - Sockets
- Protocolos nivel de Transporte
  - TCP
  - UDP
- Protocolos de nivel Aplicación
  - FTP
  - HTTP
  - DNS
  - SMTP
  - DHCP
  - POP/IMAP
  - Websocket

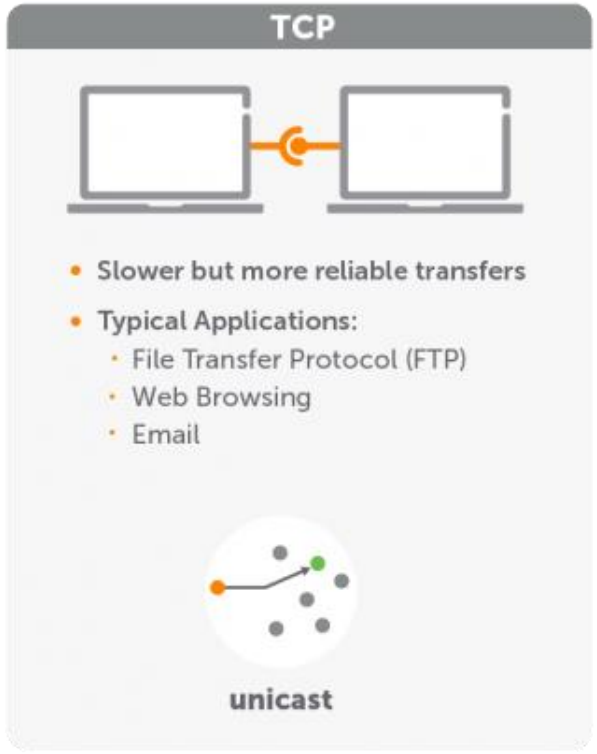


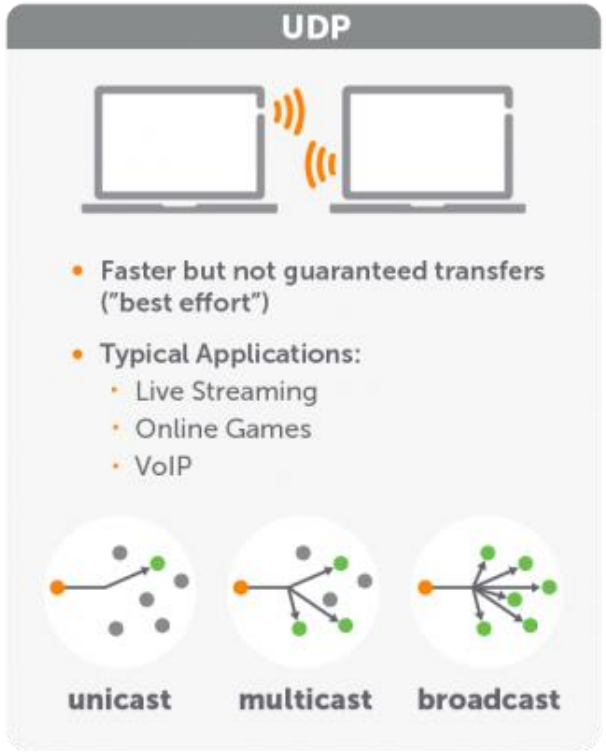
Protocolos de Transporte: TCP y UDP

TCP

- Envía segmentos
- Conexión previa: 1 to 1
- Fiable: garantía de entrega
- Precisa más recursos
- Más lento

FTP, HTTP, SMTP, etc.





UDP

- Envía datagramas
- Sin con
- Sin conexión previa, 1 a muchos.
- Sin garantía de entrega
- Ligero, más rápido

VoIP, videostream

UDP vs TCP

https://www.wowza.com/blog/udp-vs-tcp

TCP	UDP
A connection-oriented protocol.	A connectionless protocol.
Uses specific handshake protocols (generally, SYN, SYN-ACK, ACK).	No handshake.
Guarantees the delivery of data to the destination router, thus making it reliable.	Doesn't guarantee the delivery of data to the destination.
Treats communication stream as a sequence of bytes.	Messages contain packets that are considered independent of one another.
Messages make their way across the internet from one computer to another.	UDP isn't connection-based, so one program can send lots of packets to another.
Packet sequence is verified.	Data is processed in order of arrival.
Slower speed of transmission due to reordering and retransmission.	Faster because integrity is checked at the arrival time using checksum.
Performs error checking and attempts error recovery.	Performs basic error checking and discards erroneous packets without attempting error recovery.
Offers extensive error-checking mechanisms using flow control and acknowledgment of data.	Has only a single error-checking mechanism, which is used for checksums.
Acknowledges segments.	Doesn't acknowledge specific segments.
Header size is 20 bytes.	Header size is 8 bytes.
TCP is heavy. It needs three packets to set up a socket connection before data can be sent.	UDP is lightweight. There is no tracking of connections, ordering of messages, etc.

Adapted from https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html.

Sockets y Websockets

Mecanismo de comunicación de datos sobre TCP o UDP

Permiten la comunicación entre equipos, identificando los extremos por IP + puerto.

Tipos de sockets

- Stream, soportan TCP, etc
- Datagram, UDP
- Raw
- Websocket: protocolo sobre TCP que usando puertos 80 y 443 ofrecen servicios de comunicación para aplicaciones en entorno HTTP.

Ejemplos de sockets

Librería socket: Envía y recibe bytes

```
s = “datos”
s.encode() # para enviar                    s.decode() # para recibir
```

Primitivas

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # TCP
sock.bind((UDP_IP, UDP_PORT)) # Conecta el socket a un puerto (Server, comienza escucha)

        sock.sendto(MESSAGE.encode(), (UDP_IP, UDP_PORT)) # Envia un dato al socket (cliente)
        data, addr = sock.recvfrom(1024)
        data = sock.recv(1024)
```

TCP solamente:

```
        sock.listen(1) # num de clientes. Abre el canal para recibir datos.
        conn, addr = sock.accept() # Bloquea la línea y establece conexión. Solo para TCP
        sock.send(MESSAGE)
        sock.sendall(MESSAGE) # Envía datos a todos los clientes conectados
```

Recursos

Código e Ejemplo

Socket sobre UDP: u3-01-socketServer.py / u3-01-socketClient.py

Socket sobre TCP: u3-02-socketServerTCP.py / u3-02-socketClientTCP.py

Información adicional

https://es.wikipedia.org/wiki/Socket_de_Internet

Manejo de JSON: librería JSON

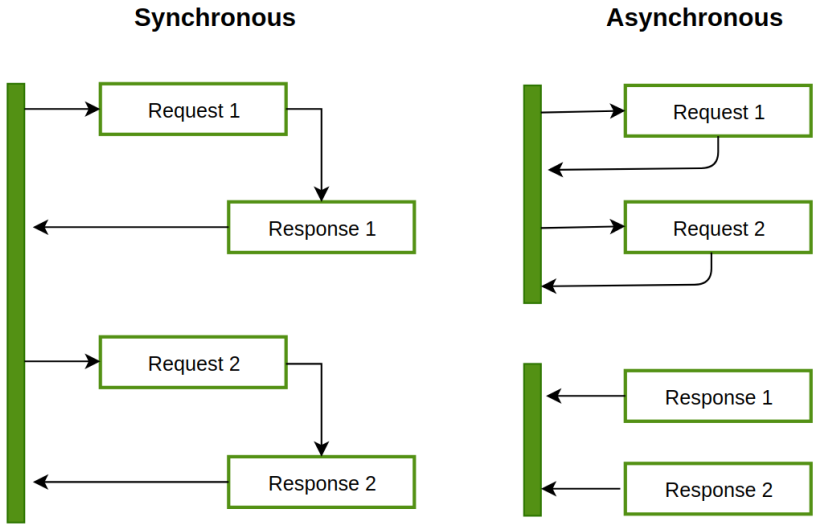
- La librería JSON permite convertir dict a/desde string
- Es ampliamente usada para el manejo de objetos json.

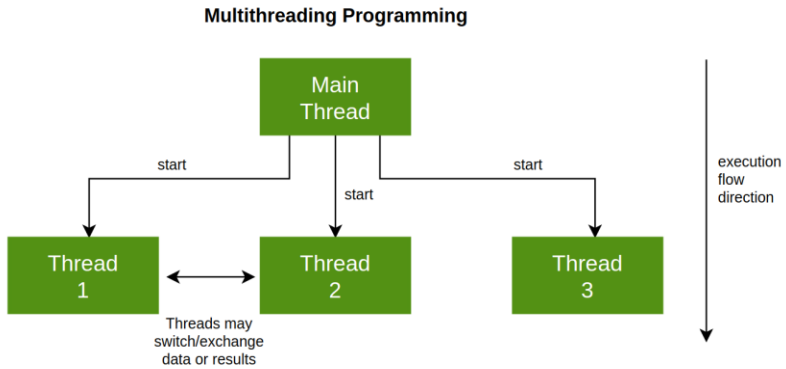
```
import json
a = json.dumps({'name':'Pepe','edad':32})
a
        '{"name": "Pepe", "edad": 32}'
type(a)
        <class 'str'>
b = json.loads(a)
b
        {'name': 'Pepe', 'edad': 32}
type(b)
        <class 'dict'>
```

Concurrencia sin multihilos ni multiproceso

asyncio

Sincrono vs Asíncrono vs Multi-threading





- Síncrono: Se espera a que cada tarea que se solicita termine su ejecución y devuelva un valor.
- Asíncrono:
  - Se lanzan ejecuciones simultáneas.
  - El flujo de ejecución puede continuar
  - Usando primitivas (await) puedo controlar qué co-routina toma la CPU.
- Multihilo: Se descompone la ejecución en vías paralelas que pueden colaborar o no.

Fuente: https://www.baeldung.com/cs/async-vs-multi-threading

Qué es la librería asyncio

- Librería de Python que permite ejecución asíncrona.
  https://docs.python.org/3/library/asyncio.html
- La programación asíncrona nos permite romper el flujo secuencial de un programa, pero manteniendo toda la ejecución en un mismo proceso, incluso en un mismo hilo.
- Posibles usos
  - Envío de una petición que consume mucho tiempo en procesarse.
  - Espera de una petición externa de la red.
  - Proceso que periódicamente hace unas comprobaciones
  - Servicio que se expone a una llamada exterior
- Elementos
  - Co-rutinas: declaradas con la sintaxis async/await es la forma de escribir funciones.

- Más información
  https://superfastpython.com/python-asyncio/

Funcionamiento

- El sistema lanza un programa que puede contener código asíncrono.
- Se estructura sobre co-rutinas que se lanzan en paralelo PERO··· solo 1 se ejecuta en cada momento.
- La rutina que está ejecutándose solamente cede el control por algún mecanismo como
  - Termina
  - Espera en una función asíncrona (como await.sleep)
  - Espera respuesta de I/O externa → Input/Print no cede control.

Asíncrono vs Síncrono

```
import time

def say_after(delay, what):
    time.sleep(delay)
    print("{} at {}".format(what, time.strftime('%X')))

def main():
    print(f"started at {time.strftime('%X')}")
    say_after(3, 'hello')
    say_after(1, 'world')
    print(f"finished at {time.strftime('%X')}")

main()
```

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print("{} at {}".format(what, time.strftime('%X')))

async def main():
    task1 = asyncio.create_task(say_after(3, 'hello'))
    task2 = asyncio.create_task(say_after(1, 'world'))

    print(f"started at {time.strftime('%X')}")
    await task1
    await task2
    #await asyncio.Future()  # run forever
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

Ejemplos:

a01-asyncio-HolaMundo.py

Sentencias

`asyncio.run(func())`

- Lanza una función con comportamiento asíncrono.
- Punto de arranque de todo programa asíncrono y solamente se llama una vez.

`task1 = ayncio.create_task(func())`

- Crea una tarea y la lanza internamente.
- La tarea termina y devuelve valor cuando desde otra función se espera con await.

`result = await task1`

- Espera la terminación de la tarea

`await asyncio.gather(task1, task2)`

- Espera la terminación de una serie de tareas

`await asyncio.Future()`

- Bucle infinito que espera en el bucle de tareas indefinidamente.

`asyncio.sleep(s)`

- Suspende solamente la co-rutina actual.

`task1.done()`

- Devuelve True si está completa

Websockets

Por qué websockets

- Nace como un mecanismo de comunicación sobre HTTP – nivel de aplicación
- Oculta al desarrollador los detalles de la conexión.
- Incorpora mecanismos que suplen problemas de HTTP:
  - HTTP solo tenemos respuesta si hacemos una petición.
  - WS puedes hacer peticiones entre cualquiera de las partes sin que se haya solicitado
    - Alternatica: *polling* desde HTTP, mayor sobrecarga.
    - Aplicación típica: chat
- Los navegadores modernos soportan Websockets
- Existen implementaciones en diversos lenguajes, las más común es en Javascript
- Usaremos la librería websockets de Python.

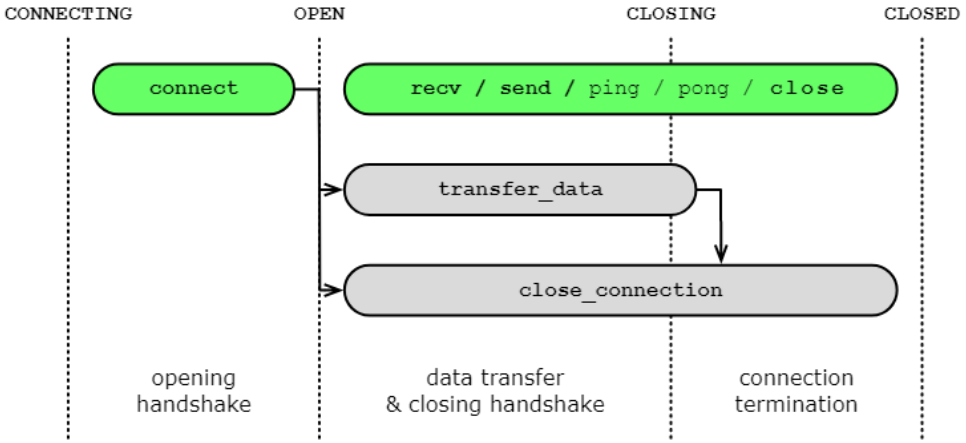
Para saber más:

- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
- https://en.wikipedia.org/wiki/WebSocket

Librería websockets

- Librería para construir clientes y servidores para comunicación sobre websocket sobre el modelo asíncrono de programación de Python
- El servidor queda como una tarea-corrutina del proceso que se activa cuando recibe una llamada
  - Este detalle es transparente para el programador.
- Al llamar a la función de conexión, el cliente abre conexión y se prepara para trasmitir.
  - El hecho de usar un gestor de contexto hace que las conexiones se cierren adecuadamente al salir del bucle.
- Comienza el intercambio de mensajes
  - Si no hay actividad, por diseño de HTTP1.1 las conexiones se cierran a los 30 segundos aprox.

- https://websockets.readthedocs.io/en/5.0/intro.html
- https://github.com/python-websockets/websockets



Referencia server

```
import websockets
```

```
websockets.serve(servidorWS, IP, PUERTO)
```

- Crea un servidor en la IP y PUERTO especificados y carga en el servidor la función que maneja la conexión (servidorWS).

```
await server.serve_forever()
```

- Ejecuta un bucle infinito de escucha que implementa la función servidorWS

```
asyncio.get_event_loop().stop()
```

- Permite salir del bucle infinito del servidor al finalizar todas las corounitas del sistema

```
websocket.send(cad)
```

- Envía la cadena cad por el websocket

```
cad = await websocket.recv()
```

- Escucha el websocket hasta que recibe un mensaje que coloca como cadena de caracteres en cad

Referencia cliente

```
import websockets
```

```
uri = "ws://"+IP+":"+str(PORT)
```

```
websockets.connect(uri)
```

- Conecta con un servidor en la URI especificada. Uri formada por protocolo (ws:// o wss://), dirección y puerto
- Se suele usar con gestor de contexto

```
async with websockets.connect(uri) as websocket:
```

```
websocket.send(cad)
```

- Envía la cadena cad por el websocket

```
cad = await websocket.recv()
```

- Escucha el websocket hasta que recibe un mensaje que coloca como cadena de caracteres en cad

Esquema de funcionamiento

Servidor

```
import asyncio
import websockets

IP = "192.168.1.38"
PORT = 9080

async def servidorWS(websocket):
    # Se pone a la escucha y recibe una cadena
    cad = await websocket.recv()

    # Gestiona los datos recibidos
    print("Recibido {}".format(cad))
    ack = "Has enviado {}".format(cad)


async def main():
    async with websockets.serve(servidorWS, IP, PORT):
        await asyncio.Future()  # run forever

if __name__ == "__main__":
    asyncio.run(main())
```

Cliente

```
import asyncio
import websockets

IP = "192.168.1.38"
PORT = 9080

async def clienteWS():
    uri = "ws://"+IP+":"+str(PORT)
    while True:
        # Proceso una información y decido enviar al WS
        cad = input("Envío: ")

        # Conecto al WS
        async with websockets.connect(uri) as websocket:

            # Envío la información
            await websocket.send(cad)

asyncio.run(clienteWS())
```