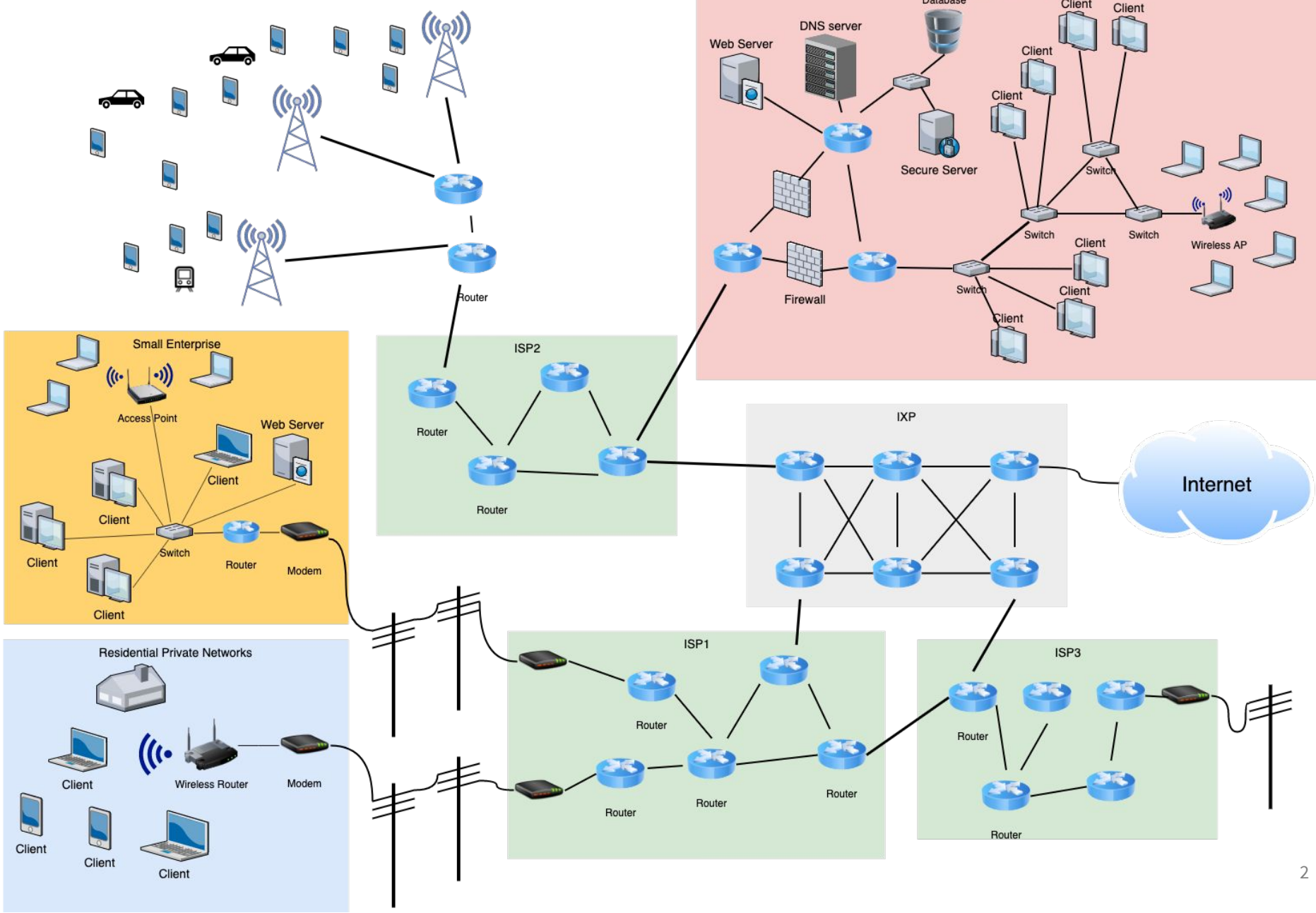
Networking

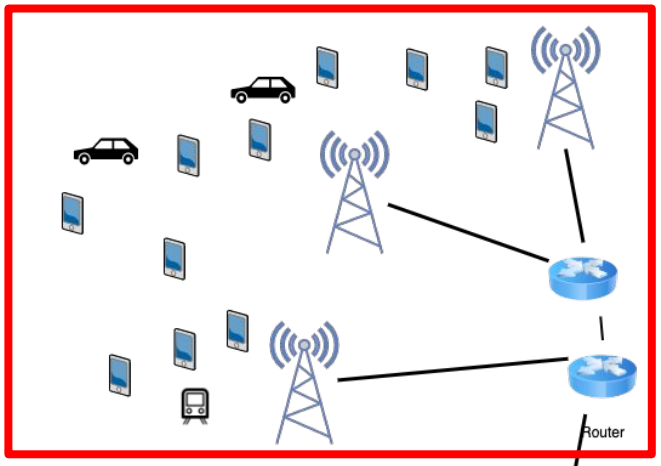
Semana 12 - Jueves 21 de noviembre de 2019

Redes

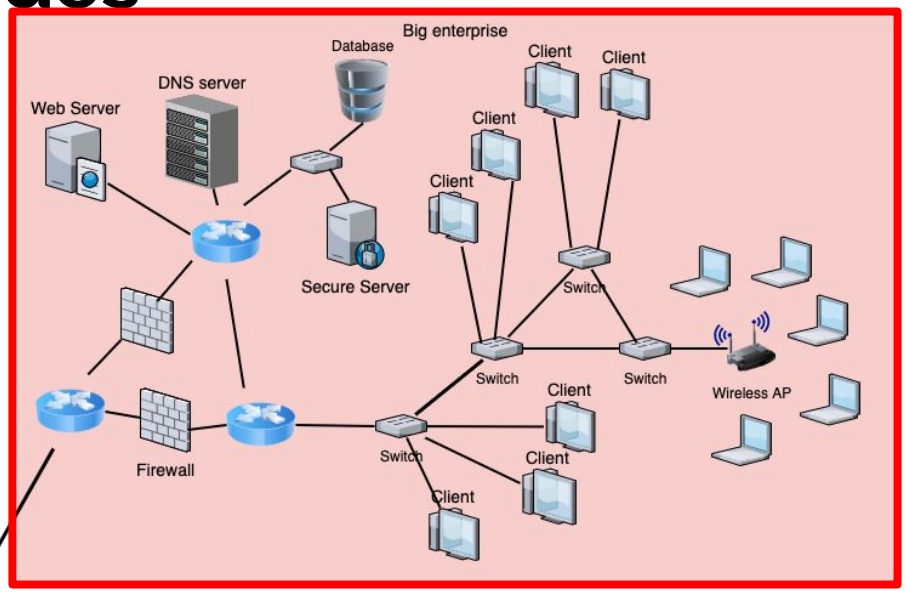

Router
Big enterprise
Database
DNS server
Web Server
Secure Server
Client
Switch
Wireless AP
Firewall
Small Enterprise
Access Point
Web Server
Client
Switch
Router
Modem
ISP2
Router
IXP
Internet
Residential Private Networks
Wireless Router
ISP1
ISP3

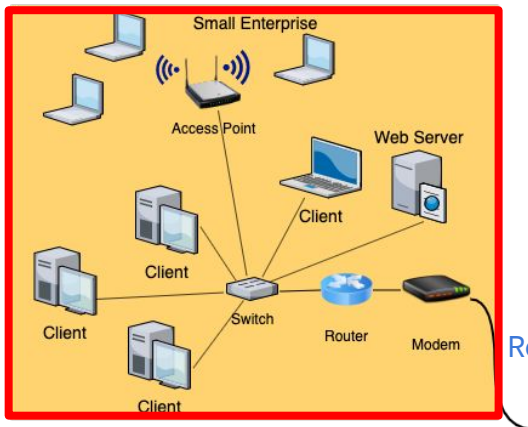
Subredes

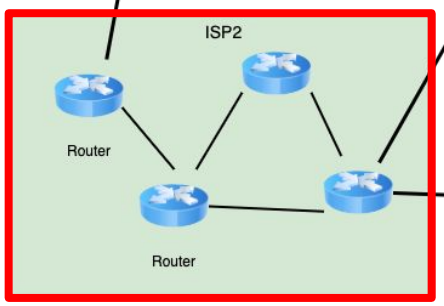
Red: conexión móvil

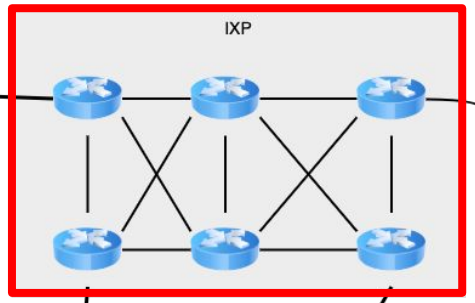


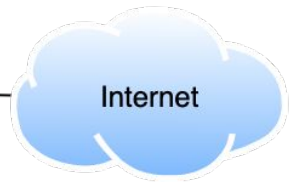
Red: empresa mediana-grande





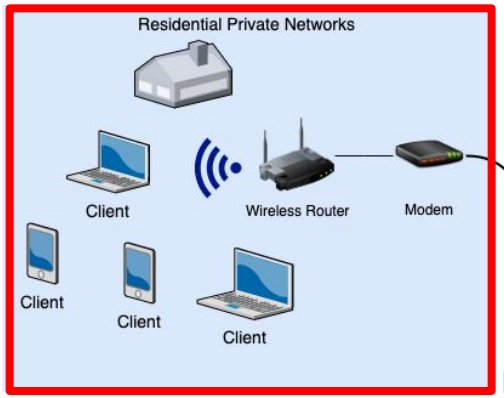


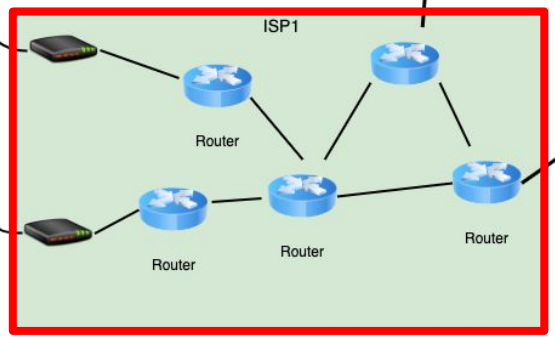


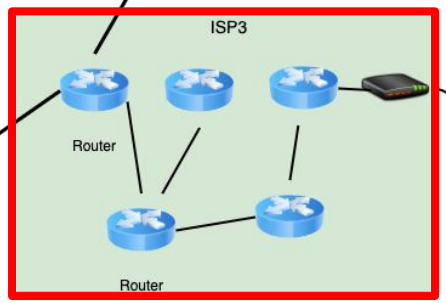


IXP o PIT: Punto de intercambio de Internet

Red: empresa pequeña





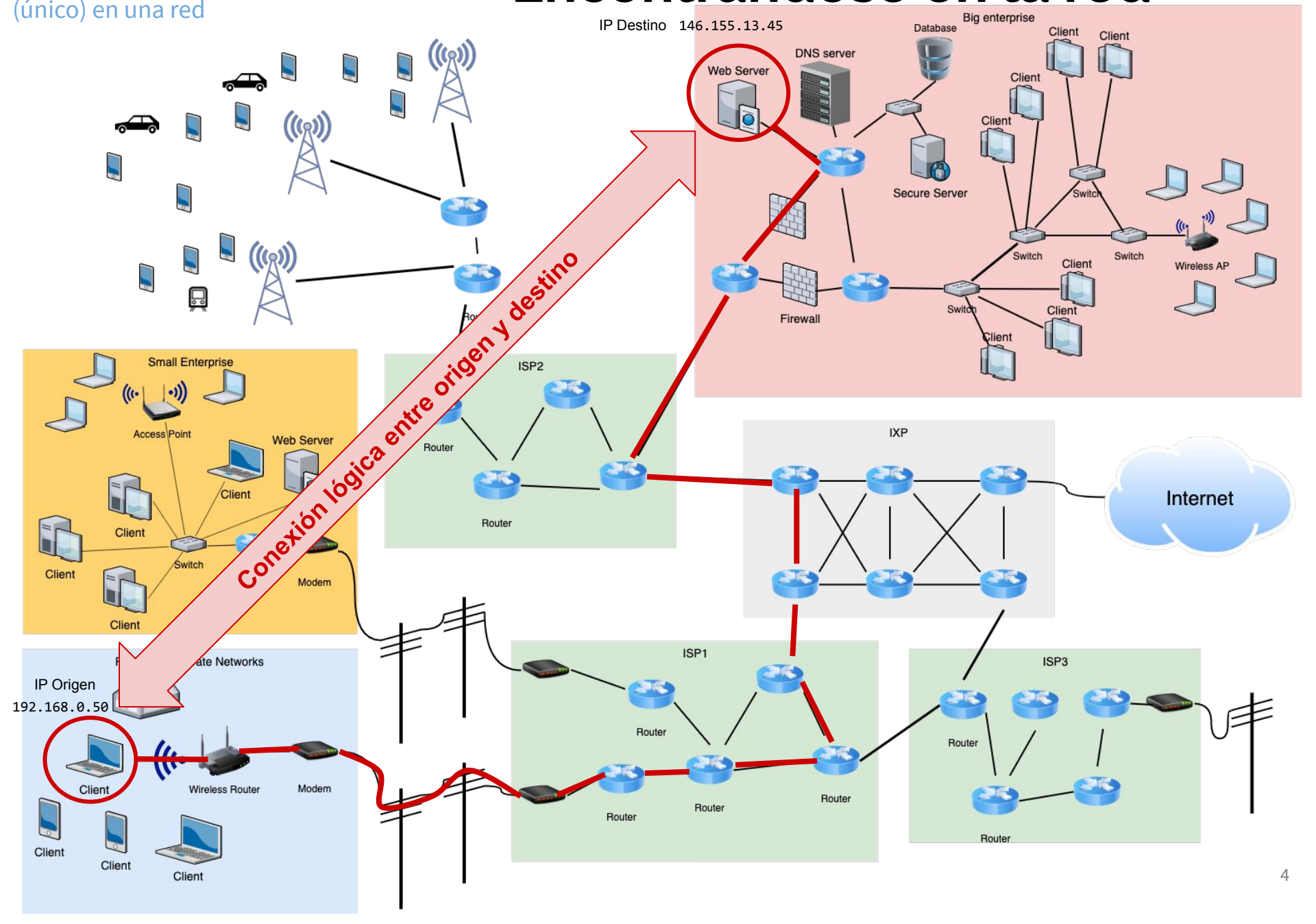


Red domiciliaria

ISP: Internet Service Provider (VTR, Movistar, Claro, Entel, etc)

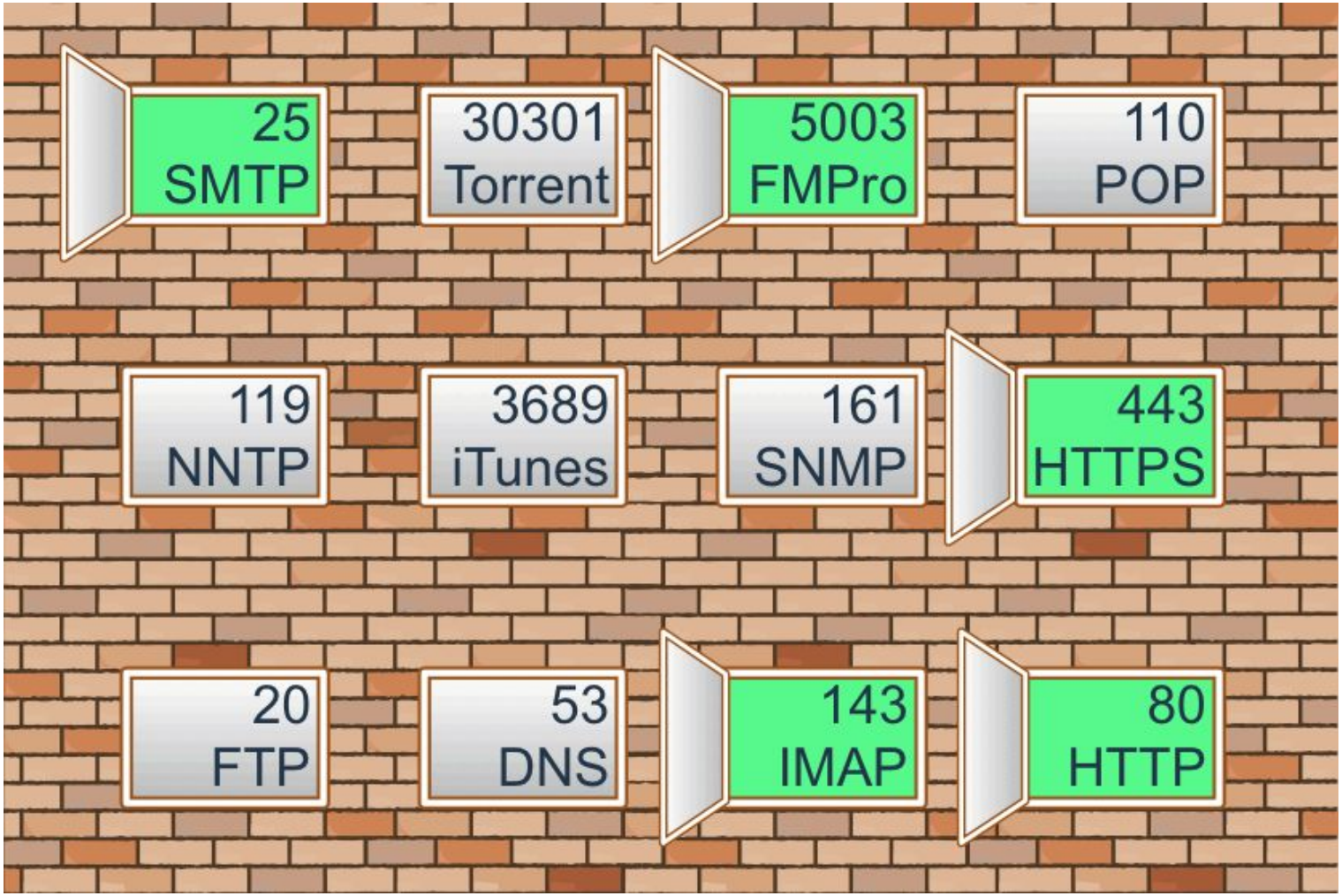
Dirección IP: permite encontrar un miembro (único) en una red

Encontrándose en la red

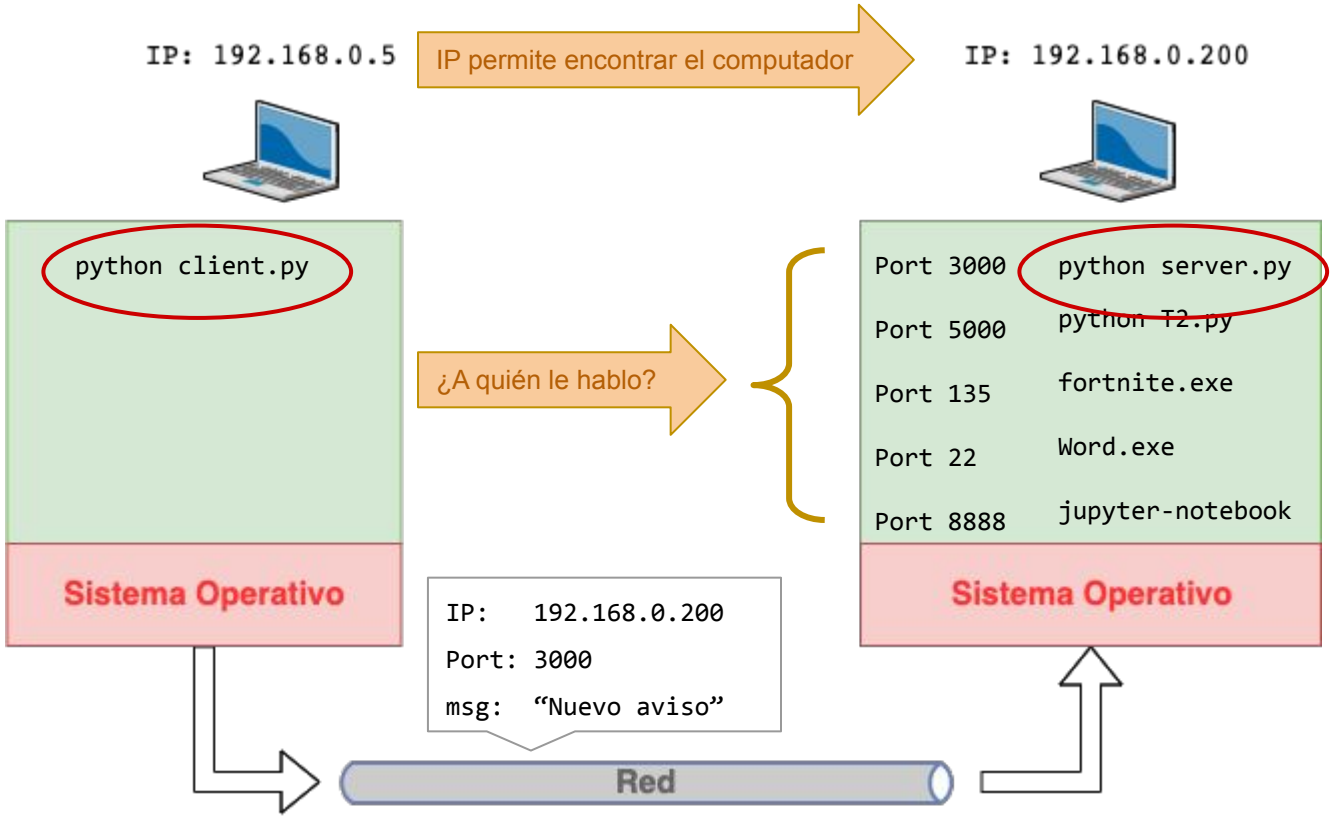


Puertos e IP

146.155.13.45



Conectando dos computadores



7: Capa de Aplicación
¿Qué mensaje envío?

6: Capa de Presentación

5: Capa de Sesión

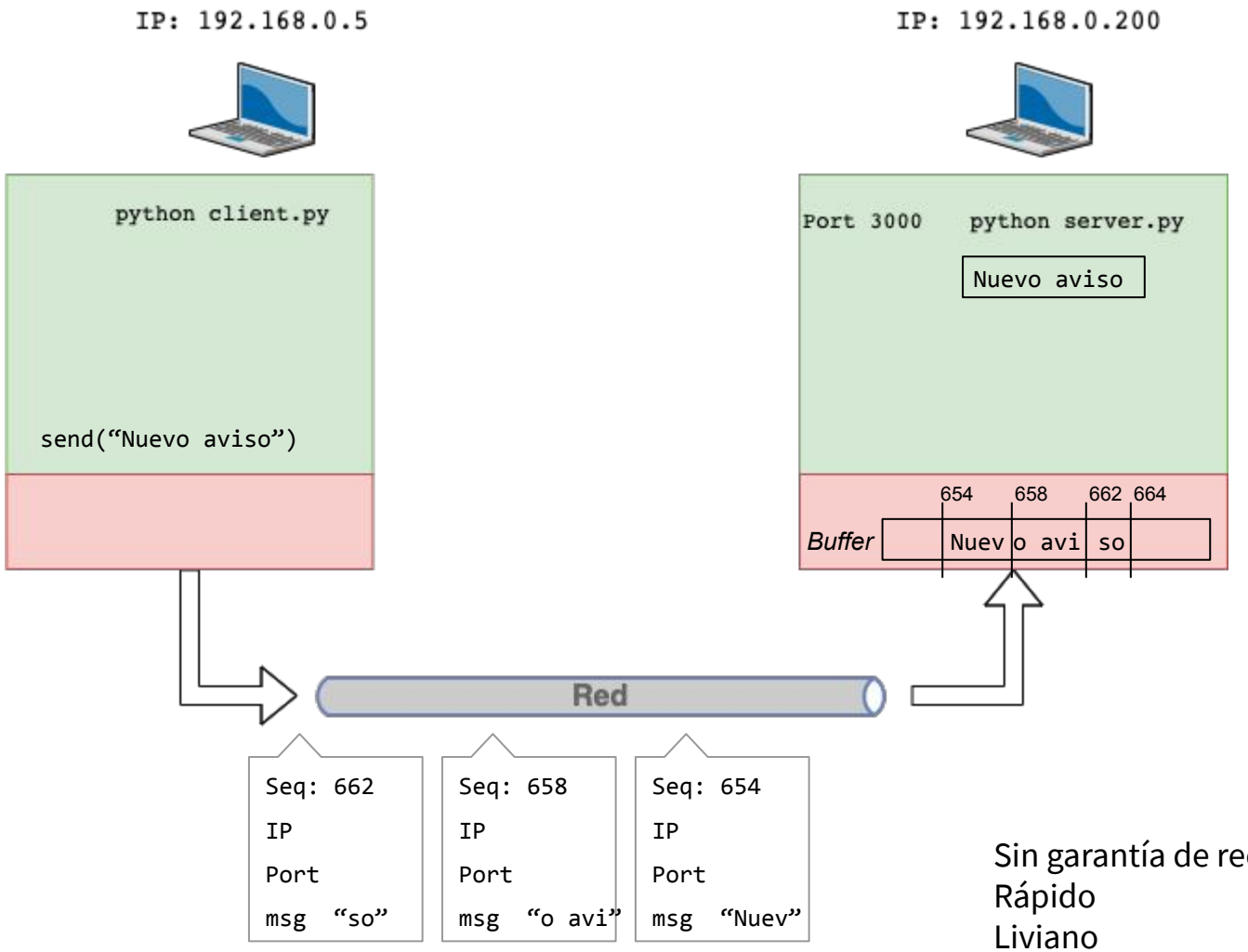
4: Capa de Transporte
¿cómo y a quién envío el mensaje?

3: Capa de Red
¿cómo encuentra el destinatario?

2: Capa de Enlace
¿cómo entrego datos a mi vecino?

1: Capa Física
¿cómo envío un byte?

Transporte: Protocolo UDP



7: Capa de Aplicación
¿Qué mensaje envío?

6: Capa de Presentación

5: Capa de Sesión

4: Capa de Transporte
¿cómo y a quién envío el mensaje?

3: Capa de Red
¿cómo encuentra el destinatario?

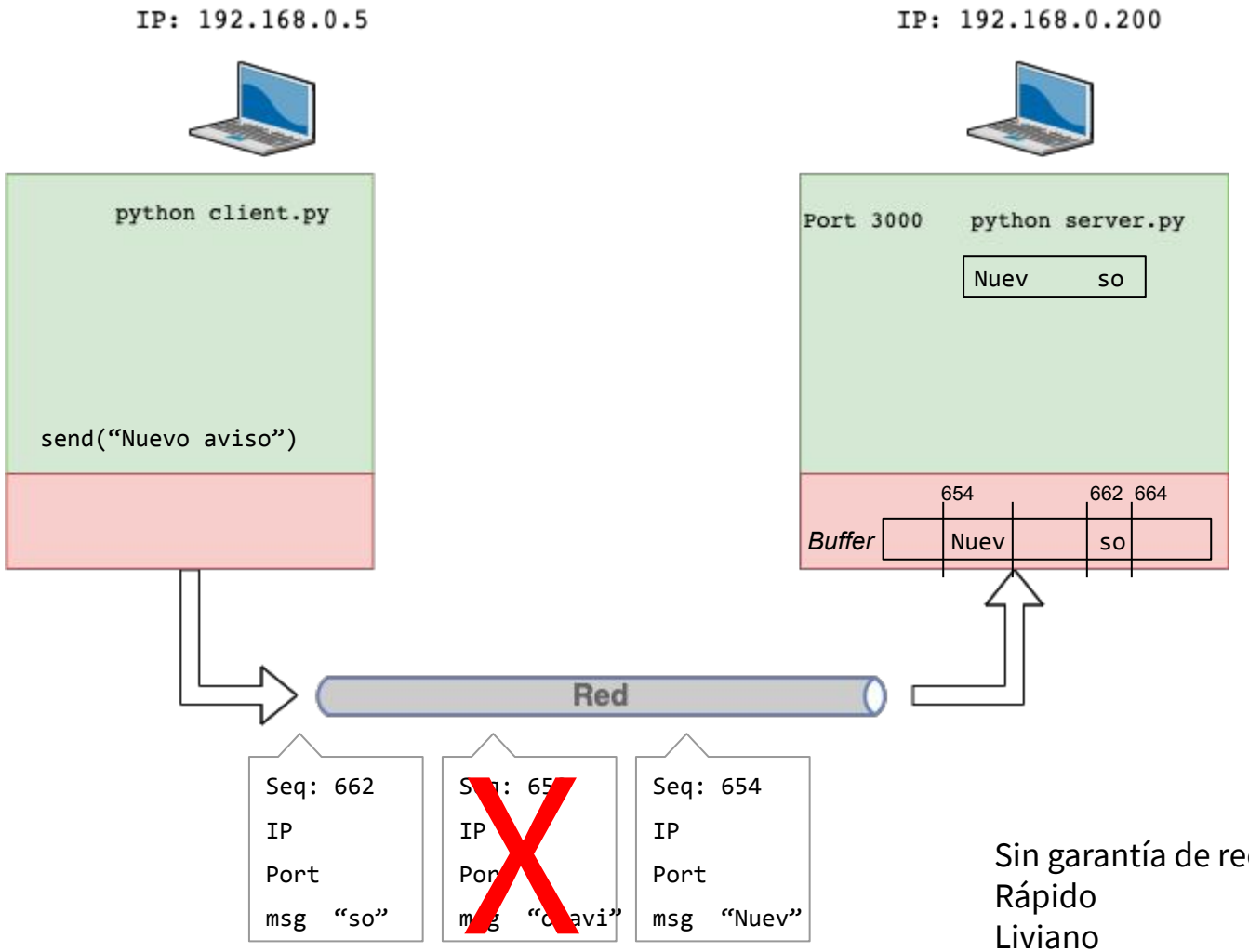
2: Capa de Enlace
¿cómo entrego datos a mi vecino?

1: Capa Física
¿cómo envío un byte?

Sin garantía de recepción
Rápido
Liviano

Transporte: Protocolo UDP

Si un paquete no llega, mensaje puede quedar incompleto



7: Capa de Aplicación
¿Qué mensaje envío?

6: Capa de Presentación

5: Capa de Sesión

4: Capa de Transporte
¿cómo y a quién envío el mensaje?

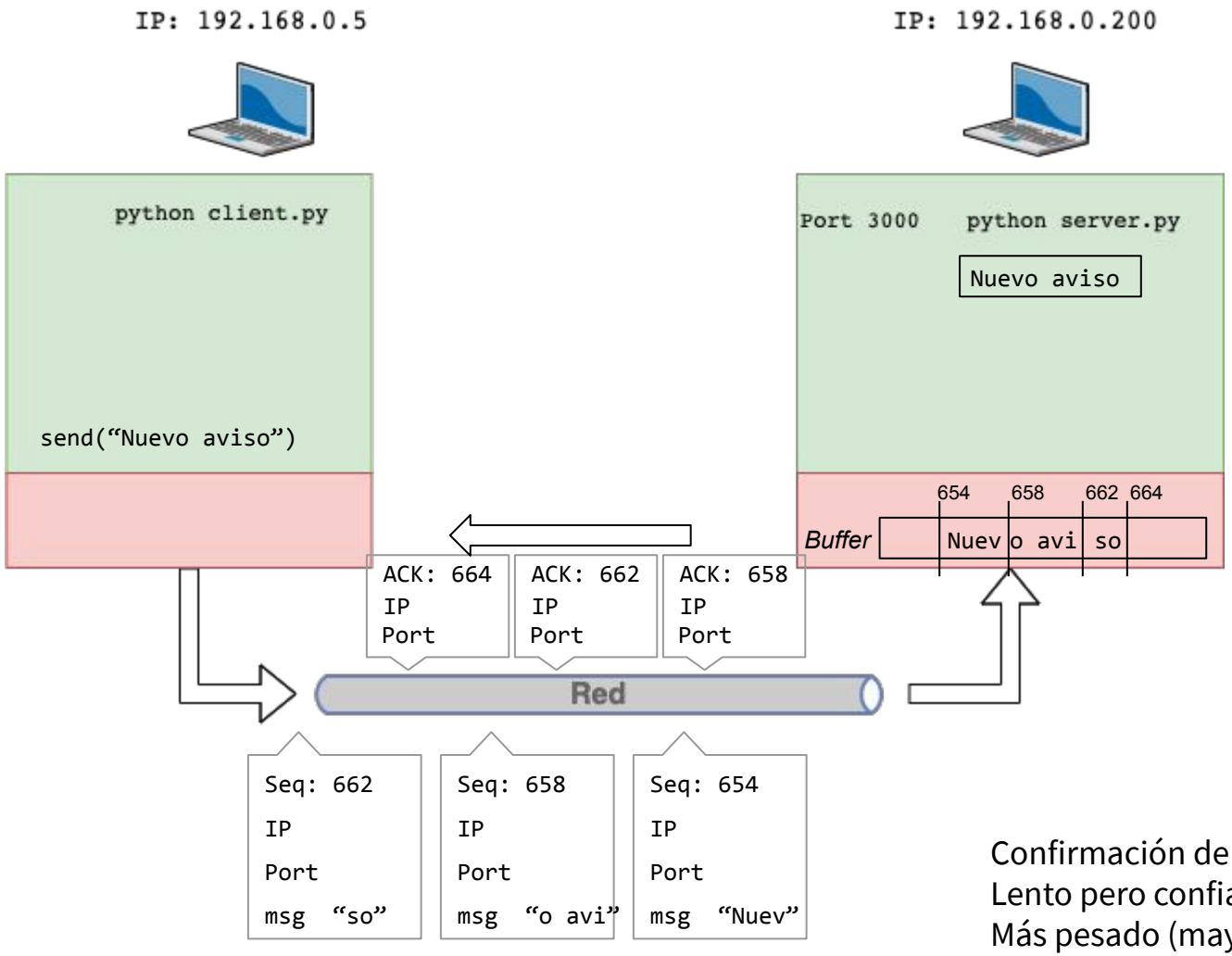
3: Capa de Red
¿cómo encuentra el destinatario?

2: Capa de Enlace
¿cómo entrego datos a mi vecino?

1: Capa Física
¿cómo envío un byte?

Sin garantía de recepción
Rápido
Liviano

Transporte: Protocolo TCP



7: Capa de Aplicación
¿Qué mensaje envío?

6: Capa de Presentación

5: Capa de Sesión

4: Capa de Transporte
¿cómo y a quién envío el mensaje?

3: Capa de Red
¿cómo encuentra el destinatario?

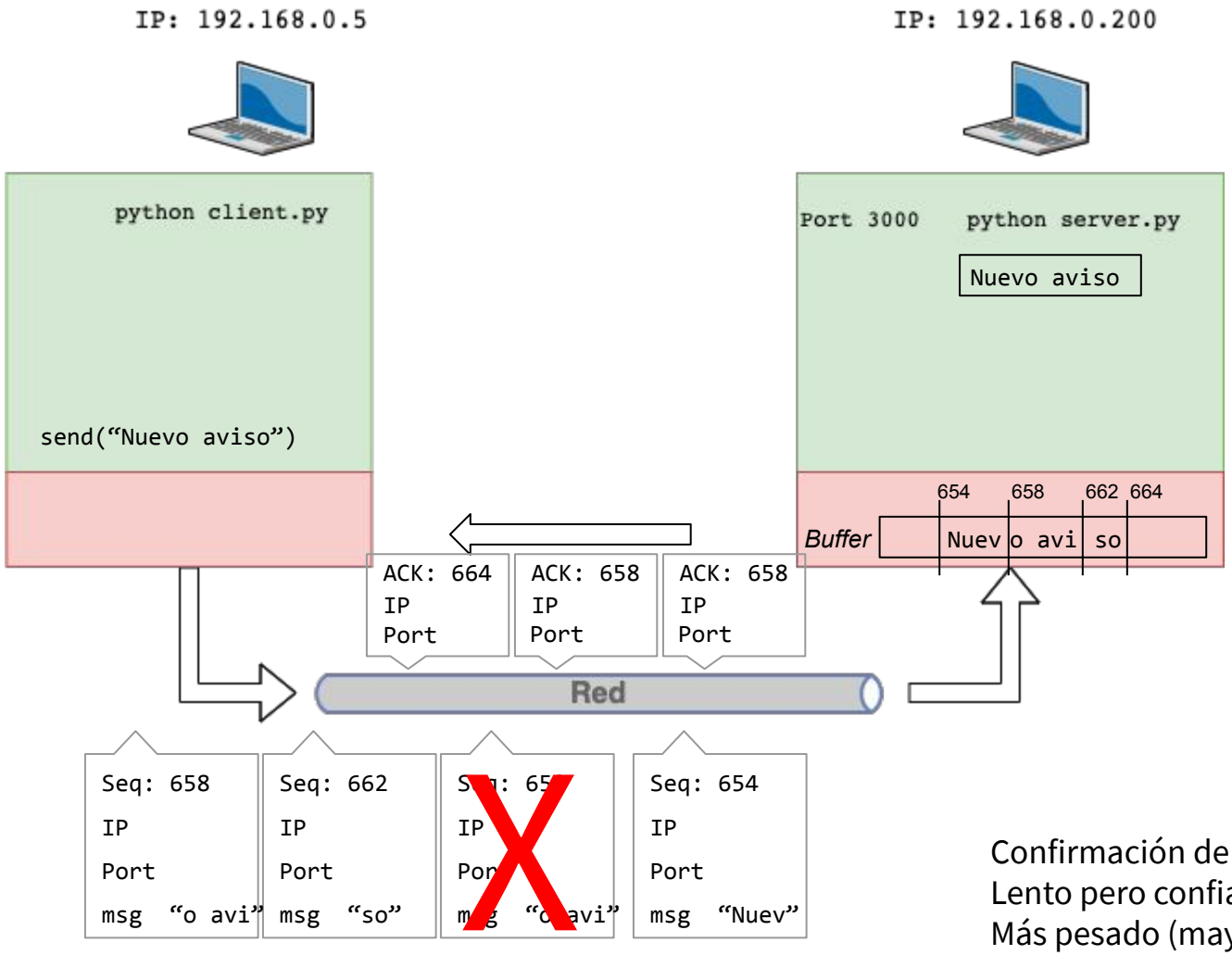
2: Capa de Enlace
¿cómo entrego datos a mi vecino?

1: Capa Física
¿cómo envío un byte?

Confirmación de recepción (ACK)
Lento pero confiable
Más pesado (mayor *overhead*)

Transporte: Protocolo TCP

Si un paquete no llega, se descubre gracias a los ACK, y se reenvía



7: Capa de Aplicación
¿Qué mensaje envío?

6: Capa de Presentación

5: Capa de Sesión

4: Capa de Transporte
¿cómo y a quién envío el mensaje?

3: Capa de Red
¿cómo encuentra el destinatario?

2: Capa de Enlace
¿cómo entrego datos a mi vecino?

1: Capa Física
¿cómo envío un byte?

Confirmación de recepción (ACK)
Lento pero confiable
Más pesado (mayor *overhead*)

Protocolo de aplicación

- ¿Cómo se envían mensajes en mi aplicación?
- ¿Qué deben tener los mensajes?
- ¿Cómo identificar los mensajes?

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Servidor

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Cliente y servidor crean sockets TCP

Servidor

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Servidor espera conexión.
Cliente solicita conexión.

Servidor obtiene socket adicional para el cliente.

Servidor

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Cliente envía 4 bytes con el tamaño, y luego el archivo.
Servidor recibe el tamaño, y lo usa para recibir *chunks* de 4096 bytes hasta completar el archivo.

Servidor

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Servidor

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Servidor guarda archivo, y responde con mensaje para el cliente.

Cliente recibe el mensaje

Ejemplo cliente-servidor

Cliente

```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

sock.connect((host_recep, puerto_recep))
print("Conexión establecida.")

with open('enviar.bin', 'rb') as binf:
    datos = binf.read()
    largo = len(datos)
    sock.sendall(largo.to_bytes(4,
byteorder='big'))
    sock.sendall(datos)

print("¡Archivo enviado!")

print("Respuesta:", sock.recv(4096).
decode('utf-8'))

# Cerramos el socket.
sock.close()
```

Cliente y servidor cierran sockets

Servidor

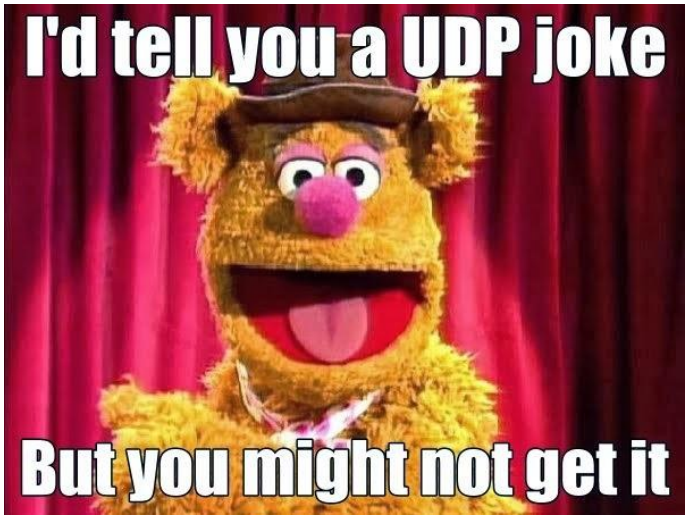
```
sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
sock.bind((host, puerto))
sock.listen()
print('Escuchando...')

sock_cl, _ = sock.accept()
print("Conexión entrante aceptada.")

largo = int.from_bytes(sock_cl.recv(4),
byteorder='big')
datos = bytearray()

while len(datos) < largo:
    leer = min(4096, largo - len(datos))
    recibidos = sock_cl.recv(leer)
    datos.extend(recibidos)

with open('recibido.bin','wb') as binf:
    binf.write(datos)

print("¡Archivo recibido!")
sock_cl.sendall("Gracias".encode('utf-8'))

# Cerramos los sockets.
sock_cl.close()
sock.close()
```

Protocolos de transporte

TCP

"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I'll tell you a TCP joke."
"Ok, I will hear a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."
"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."
"I'm sorry, your connection has timed out.
...Hello, would you like to hear a TCP joke?"

UDP



Cierre

Diagrama de flujo de AC

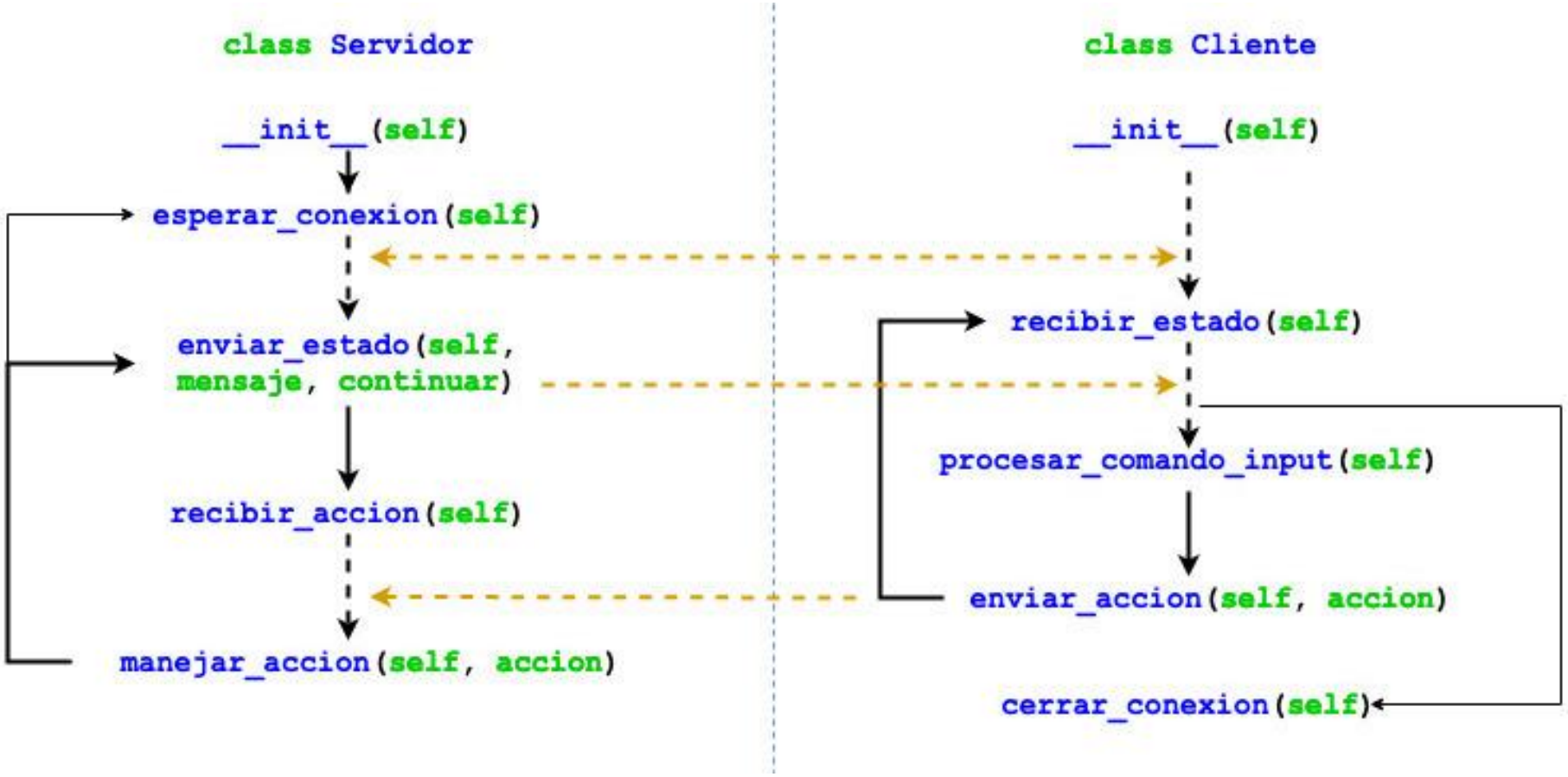
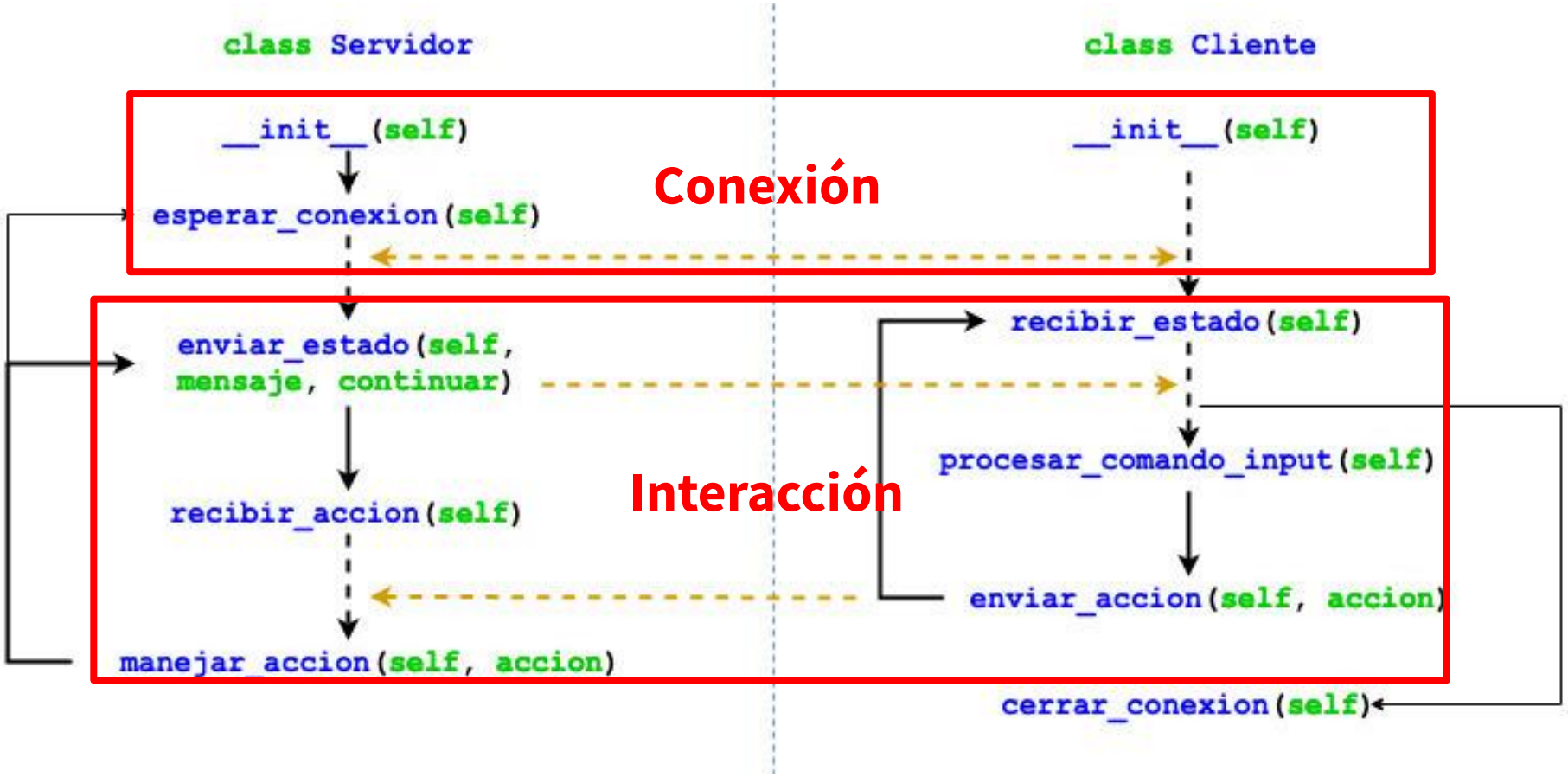


Diagrama de flujo de AC



Networking

Nuestro primer acercamiento a la estructuración de comunicación **entre** programas.

La comunicación por *sockets* es una de las más básicas que encontraremos para comunicar programas.

Su uso enseña un montón de detalles que considerar cuando hay efectivamente comunicación: direcciones, manejo de *bytes* y estructuración de protocolos.

Conexión inicial: Servidor

```
class Servidor
    def __init__(self):
        self.host = "localhost" # Valor arbitrario
        self.port = 8080 # Valor arbitrario
        self.socket_servidor = \
            socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket_servidor.bind((self.host, self.port))
        self.socket_servidor.listen()
        print("Servidor iniciado.")
        self.socket_cliente = None
        Self.juego = None

    def esperar_conexion(self):
        print("Esperando cliente...")
        socket, _ = self.socket_servidor.accept()
        self.socket_cliente = socket
        print("¡Servidor conectado a cliente!")
        self.interactuar_con_cliente()
    ...
if __name__ == "__main__":
    servidor = Servidor()
    while True:
        try:
            servidor.esperar_conexion()
        except KeyboardInterrupt:
            print("\nServidor interrumpido")
            break
```

Crear socket hábil de escuchar conexiones

Acepta conexión y comienza a interactuar con ese cliente.

Servidor una vez creado, espera de a un cliente por siempre.

Conexión inicial: Cliente

```
class Cliente:

    def __init__(self):
        self.host = "localhost" # Valor arbitrario
        self.port = 8080 # Valor arbitrario
        self.socket_cliente = \
            socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            self.socket_cliente.connect((self.host, self.port))
            print("Cliente conectado exitosamente al servidor.")
            self.interactuar_con_servidor()
        except ConnectionRefusedError:
            self.cerrar_conexion()

    ...
if __name__ == "__main__":
    Cliente()
```

Crear *socket*

Intenta conectarse al servidor, y si lo logra comienza a interactuar.

Interacción (general)

1. Se envía el estado inicial al cliente.

Servidor

```
def interactuar_con_cliente(self):
    self.enviar_estado('', True)
    while self.socket_cliente:
        accion = self.recibir_accion()
        self.manejar_accion(accion)
```

4. Servidor recibe acción y maneja el estado del juego a partir de ella.

5. Dependiendo del resultado en el juego, se envía el estado generado de vuelta al cliente.

```
def interactuar_con_servidor(self):
    while True:
        mensaje, continuar = self.recibir_estado()
        print(mensaje)
        if not continuar:
            break
        accion = self.procesar_comando_input()
        while accion is None:
            print('Input invalido.')
            accion = self.procesar_comando_input()
        self.enviar_accion(accion)
    self.cerrar_conexion()
```

6. Cliente continúa el loop.

2. Cliente recibe estado, imprime y ve si debe continuar.

3. Tras obtener la acción del usuario, se envía al servidor.

Cliente

Interacción: Cliente → Servidor

```
class Cliente:
    ...
    def procesar_comando_input(self):
        input_usuario = input('-> ')
        if input_usuario in ['\\juego_nuevo', '\\salir']:
            return {'tipo': input_usuario}
        if ' ' in input_usuario:
            division = input_usuario.split(' ')
            if len(division) == 2 and division[0] == '\\jugada' \
                                    and division[1].isnumeric():
                return {'tipo': division[0], 'data': int(division[1])}
        return None
```

El cliente recibe el comando del usuario, lo reconoce y procesa para luego enviarlo al servidor. Aquí se usa un diccionario con la info, pero no es la única forma.

Luego ocupa **`json`** para serializar y luego enviarse al servidor.

```
    def enviar_accion(self, mensaje):
        mensaje_json = json.dumps(mensaje)
        mensaje_codificado = mensaje_json.encode('utf-8')
        tamano_mensaje_bytes = len(mensaje_codificado).to_bytes(4, byteorder='big')
        self.socket_cliente.sendall(tamano_mensaje_bytes + mensaje_codificado)
```

Interacción: Cliente → Servidor

```
class Servidor:
    ...
    def recibir_accion(self):
        bytes_largo_respuesta = self.socket_cliente.recv(4)
        largo_respuesta = int.from_bytes(bytes_largo_respuesta, byteorder="big")
        bytes_respuesta = bytearray()
        while len(bytes_respuesta) < largo_respuesta:
            resto = largo_respuesta - len(bytes_respuesta)
            bytes_respuesta += self.socket_cliente.recv(min(1024, resto))
        respuesta_json = bytes_respuesta.decode('utf-8')
        respuesta = json.loads(respuesta_json)
        return respuesta


    def manejar_accion(self, accion):
        tipo = accion['tipo']
        if ...

        elif tipo == '\\jugada':
            ...
            jugada = accion['data']
```

Luego el servidor debe recibir siguiendo el proceso inverso que fue usado para enviar la info. Es decir, debe usar **`json`** y procesar el largo variable que puede tener.

Como se envió un diccionario, era relativamente fácil sacar la información necesaria que faltaba en el manejo de la acción.

Networking

Su uso enseña un montón de detalles que considerar cuando hay efectivamente comunicación:

- Cómo realizar la comunicación.
- Establecer un orden en que se enviarán mensajes entre servidor y cliente.
- Establecer en qué formato se enviarán mensajes: *strings*, diccionarios, objetos, … ¿?
- Mantener consistencia entre ambos programas para que sigan lo establecido.
- Idear un flujo claro de ejecución de los programas, de tal forma que puedan interactuar fácilmente.