

# **Introducción a Phaser 3 para Juegos en Red**

**Desarrollo de juegos en el lado del cliente**

Rubén Rodríguez Fernández (@rrunix)

2025-12-09

# Tabla de contenidos

<b>Preface</b>	<b>8</b>
<b>I Introducción a redes</b>	<b>9</b>
<b>1 Introducción a las Redes de Ordenadores</b>	<b>10</b>
1.1 Introducción . . . . .	10
1.2 La Historia de Internet . . . . .	14
1.3 Infraestructura de la red y tecnologías de transmisión . . . . .	18
1.4 Modelos de Referencia de Redes . . . . .	23
1.5 Rendimiento . . . . .	27
<b>2 Capa de Acceso a la Red</b>	<b>31</b>
2.1 Funciones principales de la Capa de Acceso a la Red . . . . .	33
2.1.1 Control de Acceso al Medio (MAC) . . . . .	33
2.1.2 Direccionamiento Físico . . . . .	33
2.1.3 Detección y Corrección de Errores . . . . .	34
2.1.4 Control de tamaño . . . . .	34
2.1.5 Sincronización y Temporización . . . . .	35
2.1.6 Gestión de Topología . . . . .	35
2.1.7 Control de Calidad de Servicio (QoS) . . . . .	35
2.2 Dispositivos de la Capa de Acceso a la Red . . . . .	36
2.3 Protocolos . . . . .	37
2.3.1 Ethernet (IEEE 802.3) . . . . .	37
2.3.2 Wi-Fi (IEEE 802.11) . . . . .	39
2.3.3 Point-to-Point Protocol (PPP) . . . . .	40
2.3.4 Frame Relay . . . . .	40
2.3.5 Address Resolution Protocol (ARP) . . . . .	42
<b>3 Capa de red</b>	<b>43</b>
3.1 Funciones Fundamentales de la Capa de Red . . . . .	44
3.2 Modelos de servicio . . . . .	46
3.3 Dispositivos físicos de la Capa de Red . . . . .	47
3.3.1 Routers (Enrutadores) . . . . .	48
3.3.2 Switches de Capa 3 . . . . .	50

3.4	Protocolos . . . . .	50
3.4.1	Protocolo IP . . . . .	50
3.4.2	Protocolo ICMP (Internet Control Message Protocol) . . . . .	55
3.4.3	NAT (Network Address Translation) . . . . .	56
<b>4</b>	<b>Capa de transporte</b>	<b>59</b>
4.1	Funciones principales . . . . .	59
4.2	Protocolos . . . . .	61
4.2.1	UDP (User Datagram protocol) . . . . .	61
4.2.2	TCP (Transmission Control Protocol) . . . . .	65
4.3	Comparativa de TCP vs UDP para videojuegos . . . . .	74
<b>5</b>	<b>Capa de aplicación</b>	<b>75</b>
5.1	Socket . . . . .	78
5.1.1	Sockets TCP . . . . .	78
5.1.2	Sockets UDP . . . . .	81
5.1.3	Servicios Requeridos y elección de capa de transporte . . . . .	84
5.2	Arquitecturas de Aplicaciones Distribuidas . . . . .	86
5.2.1	Arquitectura Cliente/Servidor . . . . .	87
5.2.2	Arquitectura Peer-to-Peer (P2P) . . . . .	89
5.3	Protocolos . . . . .	93
5.3.1	HTTP . . . . .	93
5.3.2	DNS . . . . .	95
5.3.3	SMTP, IMAP y POP . . . . .	97
5.3.4	QUIC . . . . .	98
5.4	Servicios . . . . .	99
5.4.1	CDNs . . . . .	99
<b>II</b>	<b>Desarrollo en el cliente</b>	<b>100</b>
<b>6</b>	<b>JavaScript para Desarrollo de Videojuegos</b>	<b>101</b>
6.1	Introducción . . . . .	101
6.1.1	Características de JavaScript . . . . .	101
6.1.2	Versiónes de ECMAScript . . . . .	101
6.1.3	JavaScript vs Java . . . . .	102
6.1.4	Características principales de JavaScript . . . . .	102
6.1.5	DOM y BOM . . . . .	102
6.1.6	Librerías y Frameworks JavaScript . . . . .	102
6.2	Configuración del Entorno de Desarrollo con Node.js . . . . .	103
6.2.1	Introducción a Node.js . . . . .	103
6.2.2	Instalación de Node.js . . . . .	103
6.2.3	Gestión de Paquetes con npm . . . . .	103

6.2.4	Creación de un Proyecto de Videojuego . . . . .	103
6.2.5	Instalación de Dependencias . . . . .	105
6.2.6	Configuración de Webpack . . . . .	106
6.2.7	Scripts de Desarrollo . . . . .	107
6.3	El Lenguaje JavaScript . . . . .	107
6.3.1	Características del Lenguaje . . . . .	107
6.3.2	Imperativo y Estructurado . . . . .	107
6.3.3	Lenguaje de Script . . . . .	108
6.3.4	Tipado Dinámico . . . . .	108
6.3.5	Orientado a Objetos . . . . .	108
6.3.6	Funciones como Ciudadanos de Primera Clase . . . . .	108
6.3.7	Modo Estricto . . . . .	108
6.4	Integración con HTML . . . . .	109
6.4.1	Inclusión de JavaScript . . . . .	109
6.4.2	Optimización de Carga . . . . .	109
6.5	Sintaxis Básica . . . . .	109
6.5.1	Mostrar Información . . . . .	109
6.5.2	Comentarios . . . . .	110
6.5.3	Delimitadores . . . . .	110
6.5.4	Tipos de Datos . . . . .	110
6.5.5	Variables . . . . .	111
6.5.6	Operadores . . . . .	112
6.6	Arrays . . . . .	113
6.6.1	Métodos de Array Modernos . . . . .	114
6.6.2	Destructuring de Arrays (ES2015+) . . . . .	115
6.7	Sentencias de Control de Flujo . . . . .	115
6.7.1	Sentencias Básicas . . . . .	116
6.7.2	Valores Falsy . . . . .	117
6.8	Funciones . . . . .	117
6.8.1	Declaración de Funciones . . . . .	118
6.8.2	Parámetros de Función . . . . .	118
6.8.3	Closures y Scope . . . . .	119
6.9	Manejo de Excepciones . . . . .	120
6.10	Programación Orientada a Objetos . . . . .	121
6.10.1	Clases ES2015+ . . . . .	121
6.10.2	Módulos ES2015+ . . . . .	122
6.11	Almacenamiento de Datos en el Navegador . . . . .	123
6.11.1	Local Storage . . . . .	123
6.11.2	Session Storage . . . . .	126
6.11.3	Cookies . . . . .	126
6.11.4	Comparación y Recomendaciones de Uso . . . . .	128
6.11.5	Ejemplo Práctico: Sistema de Guardado . . . . .	129

<b>7 JavaScript Orientado a Objetos para Videojuegos</b>	<b>131</b>
7.1 Introducción . . . . .	131
7.2 Orientación a Objetos en JavaScript . . . . .	131
7.2.1 Diferencias con Java: Clases vs Prototipos . . . . .	131
7.2.2 Función Constructor (Patrón Tradicional) . . . . .	135
7.2.3 Clases ES2015+ (Sintaxis Moderna) . . . . .	137
7.3 Ejercicios Prácticos . . . . .	140
7.3.1 Ejercicio 1: Separación de Arrays . . . . .	140
7.3.2 Ejercicio 2: Procesamiento de Array Bidimensional . . . . .	141
<b>8 Introducción a Phaser 3</b>	<b>142</b>
8.1 Características principales . . . . .	142
8.2 Requisitos para empezar . . . . .	142
8.2.1 Navegador web . . . . .	142
8.2.2 Framework Phaser 3 . . . . .	143
8.3 Estructura básica de un juego en Phaser 3 . . . . .	143
8.3.1 El elemento Canvas . . . . .	143
8.3.2 Configuración inicial del juego . . . . .	143
8.4 Gestión de escenas . . . . .	146
8.4.1 Concepto de escena . . . . .	146
8.4.2 SceneManager . . . . .	146
8.5 Trabajo con imágenes . . . . .	149
8.5.1 Cargar y mostrar imágenes . . . . .	149
8.5.2 Sistema de coordenadas y origen . . . . .	150
8.5.3 Transformaciones básicas . . . . .	150
8.6 Introducción a Phaser 3 . . . . .	151
8.6.1 Características principales . . . . .	152
8.6.2 Requisitos para empezar . . . . .	152
8.7 Estructura básica de un juego en Phaser 3 . . . . .	153
8.7.1 El elemento Canvas . . . . .	153
8.7.2 Configuración inicial del juego . . . . .	153
8.8 Gestión de escenas . . . . .	155
8.8.1 Concepto de escena . . . . .	155
8.8.2 SceneManager . . . . .	156
8.9 Trabajo con imágenes . . . . .	159
8.9.1 Cargar y mostrar imágenes . . . . .	159
8.9.2 Sistema de coordenadas y origen . . . . .	159
8.9.3 Transformaciones básicas . . . . .	160
8.10 Motores de físicas en Phaser 3 . . . . .	161
8.10.1 Arcade Physics . . . . .	161
8.10.2 Impact Physics . . . . .	162
8.10.3 Matter Physics . . . . .	162
8.10.4 Añadir físicas a objetos . . . . .	162

8.11 Sistema de entrada (Input) . . . . .	163
8.11.1 Teclado . . . . .	163
8.11.2 Ratón . . . . .	165
8.12 Detección de colisiones . . . . .	166
8.12.1 Conceptos básicos . . . . .	166
8.12.2 Colisiones básicas . . . . .	166
8.12.3 Trabajar con grupos . . . . .	167
8.12.4 Callbacks y condiciones . . . . .	168
8.12.5 Colisiones entre grupos . . . . .	169
8.12.6 Detectar colisiones específicas . . . . .	170
8.13 Integración con el Patrón Command . . . . .	170
8.13.1 ¿Por qué usar el Patrón Command con Phaser? . . . . .	170
8.13.2 Estructura básica del Patrón Command en Phaser . . . . .	171
8.13.3 Implementación en una escena de Phaser . . . . .	174
8.13.4 Ventajas de esta arquitectura . . . . .	175
8.13.5 Ventajas del Patrón Command con Phaser 3 . . . . .	175
<b>III Desarrollo en el servidor y comunicación</b>	<b>176</b>
<b>References</b>	<b>177</b>

# Listado de Figuras

1.1	Topología jerárquica de Internet . . . . .	11
1.2	Distribución geográfica de los nodos de ARPANET en 1971 (BBC Brasil 2019) . . . . .	15
1.3	Número de dispositivos conectados por década desde ARPANET hasta Internet en 2025 (Ritchie et al. 2023; Analytics 2020) . . . . .	17
1.4	Esquema de tiempo de los eventos más significativos desde la creación de ARPANET hasta 2025. . . . .	17
1.5	Red simplificada (Kurose y Ross 2017) . . . . .	19
1.6	Modelos OSI y TCP/IP. . . . .	24
2.1	Ejemplo de envío de paquetes en una red local donde no se conoce la MAC del destinatario. . . . .	32
2.2	Cabeceras de un paquete de Ethernet. . . . .	38
2.3	Cabeceras de un paquete Wi-Fi . . . . .	41
3.1	Ejemplo simplificado de la estructura de red. . . . .	43
3.2	Ejemplo de envío de datagrama IP entre dos ordenadores en diferentes redes. . . . .	45
3.3	Formato de cabeceras de IPv4. . . . .	51
3.4	Ejemplo de estructuras de subredes. . . . .	52
3.5	Formato de cabeceras de IPv6. . . . .	55
3.6	Ejemplo de NAT con dos Host que se comunican con un servidor web utilizando una única IP pública. . . . .	57
4.1	Ejemplo de envío de 5 paquetes a través de Internet con UDP. “Internet” en este diagrama representa todo el proceso de envío. . . . .	60

# Preface

Apuntes de la asignatura Juegos en Red. Se organizan en 4 partes:

- Introducción a las redes de computadores. Ofrece una visión general de cómo funcionan las redes de computadores, tanto la parte hardware como software, utilizando la categorización de la pila de protocolos TCP/IP. Esta parte además tendrá una serie de visualizaciones y “Juegos” que podréis encontrar en [jergames.dslabapps.es](http://jergames.dslabapps.es)
- Breve introducción a HTML, CSS, JS y Phaser con el que crearemos un juego web que al que se le añadirán funcionalidades de red en las próximas partes.
- Comunicación entre clientes y servidores a través de APIs REST.
- Comunicación asíncrona entre clientes y servidores.

## ! Importante

En la parte de redes de computadores, muchos conceptos debido a la naturaleza introductoria de este curso. Esta simplificación se ve realizada por omisión, por ejemplo, se cuentan las variantes antiguas pero no las más nuevas que son más complejas, o por incompletitud, ignorando alguna parte de tecnologías o protocolos que dificultarían su explicación.

# **Parte I**

## **Introducción a redes**

# 1 Introducción a las Redes de Ordenadores

## 1.1. Introducción

La etimología de Internet es “Interconnected Networks” (Redes interconectadas), lo cual nos da una pista sobre qué es: una red global interconectada de redes más pequeñas que permite la comunicación entre los diferentes dispositivos conectados. Internet opera como un sistema descentralizado compuesto por varias capas de redes, desde las LAN (Local Area Networks), que son los nodos donde nos conectamos, hasta las WAN (Wide Area Networks) que abarcan continentes.

En la Figura 1.1 podemos ver una organización jerárquica de Internet, donde múltiples redes del mismo nivel se agregan para formar el siguiente nivel superior. A medida que ascendemos en la jerarquía, el número de dispositivos que se pueden conectar se incrementa exponencialmente. Las PAN son redes que conectan dispositivos personales como smartphones y sirven para conectar con smartwatches y electrodomésticos, entre otros. Desde las PAN podemos tomar dos rutas principales hasta el Internet, a través de las LAN o directamente con las MAN (5G o 4G). Las LAN son redes que cubren hogares, oficinas u otras unidades donde el número de dispositivos es reducido. La conexión a las LAN puede ser cableada o inalámbrica, por ejemplo con WLAN (Wi-Fi). Una CAN es una agrupación de redes LAN, generalmente en campus universitarios o grandes empresas donde el número de dispositivos es elevado. Las redes CAN, LANs individuales y las redes de telefonía móvil (e.g., 4G/5G) se juntan para dar lugar a las MAN. Las MAN generalmente abarcan ciudades o grupos de ciudades, que interconectadas dan lugar a las WANs. Finalmente, múltiples WANs dan lugar al Internet global. Las VPN (Virtual Private Network) operan como túneles seguros sobre toda esta infraestructura, permitiendo que los dispositivos cambien dinámicamente entre rutas según la tecnología disponible en cada momento.

Para ilustrar el funcionamiento de Internet vamos a utilizar un ejemplo simplificado. Supongamos que María quiere enviar un mensaje desde un Smartphone conectado a Internet a través del WiFi de su casa en Madrid a Takeshi, conectado a una LAN en la universidad de Tokio (dentro de una CAN). Puede que algunos términos no os suenen, no os preocupéis, los iremos viendo a lo largo de la asignatura. El proceso de envío sería el siguiente:

- **1. Origen - LAN Madrid:** El smartphone de María crea el paquete (podéis pensar en él como un mensaje) con la dirección IP de destino de Tokio y la dirección MAC del router WiFi como destino inmediato. El router WiFi recibe el frame Ethernet, examina

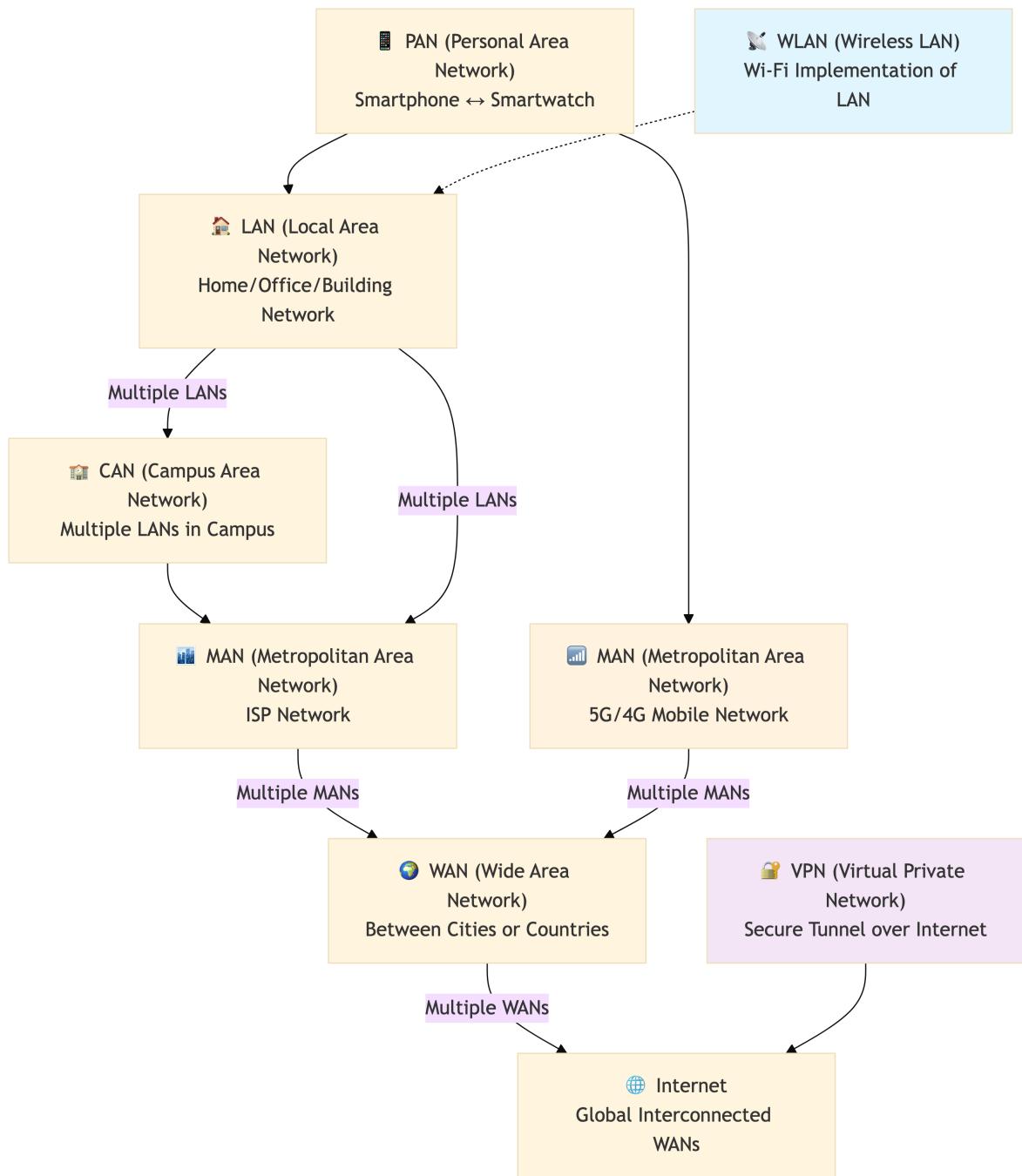


Figura 1.1: Topología jerárquica de Internet.

la dirección IP de destino y se da cuenta de que no pertenece a su red local. Reemplaza la dirección MAC de destino por la de su gateway (ISP) y reenvía el paquete.

- **2. Router local a MAN:** El router del ISP local recibe el frame con su propia dirección MAC como destino. Extrae el paquete IP, examina la dirección IP de destino y determina que debe enviarlo hacia la MAN de Madrid. Encapsula el paquete en un nuevo frame con la dirección MAC del siguiente router como destino.
- **3. MAN a WAN nacional:** El router de la MAN de Madrid recibe el frame dirigido a su dirección MAC, extrae el paquete IP y analiza el destino. Al comprender que Tokio está fuera de España, encapsula el paquete en un nuevo frame con la dirección MAC del router de la WAN española como destino.
- **4. WAN a Internet global:** El router de la WAN española recibe el frame con su dirección MAC, consulta sus tablas de rutas internacionales para Japón y encapsula el paquete con la dirección MAC del siguiente router en la ruta internacional. En cada salto a través del backbone de Internet, los routers intercambian las direcciones MAC (origen y destino) mientras preservan las direcciones IP originales.
- **5. Llegada a Japón - WAN a MAN:** Un router de la WAN japonesa recibe el frame dirigido a su dirección MAC, reconoce que el destino IP está dentro de Japón y encapsula el paquete con la dirección MAC del router de la MAN de Tokio como nuevo destino.
- **6. MAN a CAN:** El router de la MAN de Tokio recibe el frame con su dirección MAC como destino, examina la IP y determina que pertenece a la universidad de Tokio. Encapsula el paquete en un nuevo frame dirigido a la dirección MAC del router gateway de la CAN universitaria.
- **7. CAN a LAN destino:** El router de la CAN universitaria recibe el frame dirigido a su dirección MAC, analiza la IP de destino para identificar qué LAN específica del campus corresponde, y encapsula el paquete con la dirección MAC del router de esa LAN como destino.
- **8. Destino final - LAN universitaria:** El router de la LAN recibe el frame con su dirección MAC como destino, extrae el paquete IP y lo entrega al switch. El switch examina sus tablas ARP para encontrar la dirección MAC correspondiente a la IP de Takeshi, y finalmente envía el frame con la dirección MAC real de Takeshi como destino, completando el viaje desde Madrid.

En este ejemplo simplificado de envío de un mensaje por Internet ya estamos dispuestos para comprender algunos de sus componentes y identificadores. Si os fijáis, hay dos componentes que están presentes a lo largo del ejemplo, los switches y routers. El router es un dispositivo que conecta diferentes redes entre sí usando direcciones IP. Es como un “director de tráfico” que conoce las rutas entre redes distantes. Su funcionamiento a grandes rasgos es el siguiente: Llega un paquete, se identifica a través de la IP destino el camino de salida obteniendo la MAC del

siguiente salto (**hop**), y se envía el paquete. Este proceso, denominado enrutamiento<sup>1</sup>, se repite hasta llegar a la red destino, por eso estos algoritmos, y por ello se determinan **hop by hop**. El switch por otra parte es un dispositivo que conecta equipos dentro de una misma red local usando direcciones MAC. Funciona como un “repartidor inteligente” que conoce exactamente dónde está cada dispositivo en su red. Completando la analogía, Los switches manejan el tráfico local, mientras que cuando necesitan enviar datos fuera de su red, los entregan a los routers. Los routers, a su vez, se conectan a otros routers o switches según el destino.

El procedimiento de envío se realiza con dos identificadores que hemos mencionado durante el ejemplo, la MAC y la dirección IP. La dirección IP funciona como la dirección postal de una casa y permite localizar el dispositivo en las redes (como 192.168.1.100), y la dirección MAC, que es como el DNI del dispositivo: único, asignado por el fabricante y que no cambia nunca. Los routers usan direcciones IP para decidir hacia dónde enviar los paquetes, mientras que los switches usan direcciones MAC para entregar los datos al dispositivo correcto dentro de la red local. Por último, tenemos ARP, un protocolo que nos permite relacionarlas. El protocolo ARP es como un servicio de directorio telefónico: cuando un dispositivo conoce la “dirección postal” (IP) pero necesita el “DNI” (MAC) para hacer la entrega final, envía una consulta ARP preguntando “¿quién vive en esta dirección?”. El dispositivo correspondiente responde con su MAC, permitiendo que la comunicación se complete. ARP traduce entre el mundo de las direcciones (IP) y el mundo de las identidades físicas (MAC).

Una vez vistos los componentes principales de Internet, vamos a realizar unas observaciones. Primero, Internet es un sistema distribuido. Esto quiere decir que es una unión de dispositivos que operan juntos con el fin de ofrecer una funcionalidad. Segundo, Internet tiene una arquitectura descentralizada<sup>2</sup>. Por lo tanto, la caída de alguna parte de Internet no tiene porque implicar la caída de Internet globalmente. Tercero, la ejecución de los procesos de enrutamiento es local. Cada nodo de la red sólo necesita saber cual va a ser el siguiente destino (“hop”). Es decir, no hay una planificación global para el envío de los paquetes. Debido a esto los algoritmos de enrutamiento normalmente se denominan “hop by hop” e incorporan información de tiempo real<sup>3</sup>, por lo que dos mensajes enviados al mismo destino no tienen por qué seguir la misma ruta. Todo esto facilita la escalabilidad del sistema, disminuye la congestión de la red y además proporciona resiliencia a fallos.

Hasta ahora hemos visto un ejemplo simplificado de envío de mensaje, los principales componentes de Internet y algunos conceptos técnicos. Pero aún falta algo. Hemos dicho que Internet es un sistema distribuido formado por redes interconectadas. Pero, ¿Cómo se entienden entre sí?. La respuesta son los protocolos. Un protocolo es una serie de pasos bien definidos que se realizan con un objetivo. En redes de computadores, es como un manual de instrucciones que especifica cómo dos dispositivos deben intercambiar información. Es un conjunto de reglas que define

<sup>1</sup>Enrutamiento es una adaptación al español del término inglés *routing*. Aunque no está reconocida oficialmente por la RAE (2025) y el término normativo sería *encaminamiento*, en estos apuntes se utilizará *enrutamiento* por ser la forma más habitual en el ámbito de las redes y telecomunicaciones.

<sup>2</sup>Algunos componentes de Internet tienen una arquitectura híbrida, como los ISPs grandes y DNS.

<sup>3</sup>Los routers comparten y propagan información de congestión y destinos disponibles con los routers adyacentes.

exactamente cómo deben estructurarse los mensajes, en qué orden enviarlos, qué estructura y formato tienen los mensajes que recibimos, y cómo se debe actuar. Internet funciona gracias a una familia de protocolos organizados en capas, que veremos en el ([intro-network-stacks?](#)).

Por exemplificar los protocolos de red con una analogía, son como las reglas de tráfico en una ciudad: así como los autos necesitan semáforos, señales y carriles para circular ordenadamente sin chocar, los datos en una red necesitan protocolos que definan cómo moverse, comunicarse y llegar a su destino correctamente. Sin estas reglas, tanto el tráfico vehicular como el flujo de datos serían un caos total, con “accidentes” y pérdida de información constante. Desde un punto de vista más formal, podríamos definir un protocolo como:

### 💡 Protocolo

Un protocolo define una serie de tipos de mensaje, su sintaxis y su semántica, así como las reglas de cuándo y cómo enviar/responder los mensajes.

En los siguientes apartados vamos a profundizar en los conceptos introducidos hasta ahora, obteniendo un mejor entendimiento de cómo funciona Internet, cuales son los principales actores involucrados, y cómo podemos realizar nuestras propias aplicaciones que funcionen sobre la red. Los puntos se abordarán de la siguiente manera. Primero, se verán desde un enfoque “informático”, y después contextualizaremos como encaja cada uno de los puntos desde el punto de vista de desarrollo de videojuegos.

## 1.2. La Historia de Internet

Internet, como otros muchos avances de la sociedad, nació como una necesidad de guerra. En concreto, en la Guerra Fría. En una guerra la información y poder comunicarla es poder. El objetivo inicial del germen de Internet, llamado ARPANET, era precisamente la comunicación de información y que estos medios fuesen capaces de sobrevivir a un ataque nuclear. En 1969 la red experimental contaba con 4 host: UCLA, Stanford, UC Santa Bárbara y la Universidad de Utah. Esta red utilizaba un mecanismo para la comunicación de información llamada commutación de paquetes, donde los mensajes se dividían en paquetes más pequeños que podían tomar diferentes rutas hasta llegar a su destino. Así cumplieron los requisitos de tolerancia a fallos en el envío de información a través de la descentralización y duplicidad, y además sentaron la semilla que permitiría una escalabilidad natural. En 1971 ya contaba con 23 host (ver Figura 1.2), y en 1973 se realizó la primera conexión internacional con Noruega y Londres a través de tecnología satelital. En esa época la red se incrementaba a razón de 1 host cada aproximadamente 20 días.

En la década de los 80 ocurrirían 4 eventos que darían forma al Internet que conocemos hoy en día. En primer lugar, en 1983 se adoptó oficialmente la pila de protocolos TCP/IP como estándar para ARPANET, que estableció las reglas de comunicación que aún seguimos

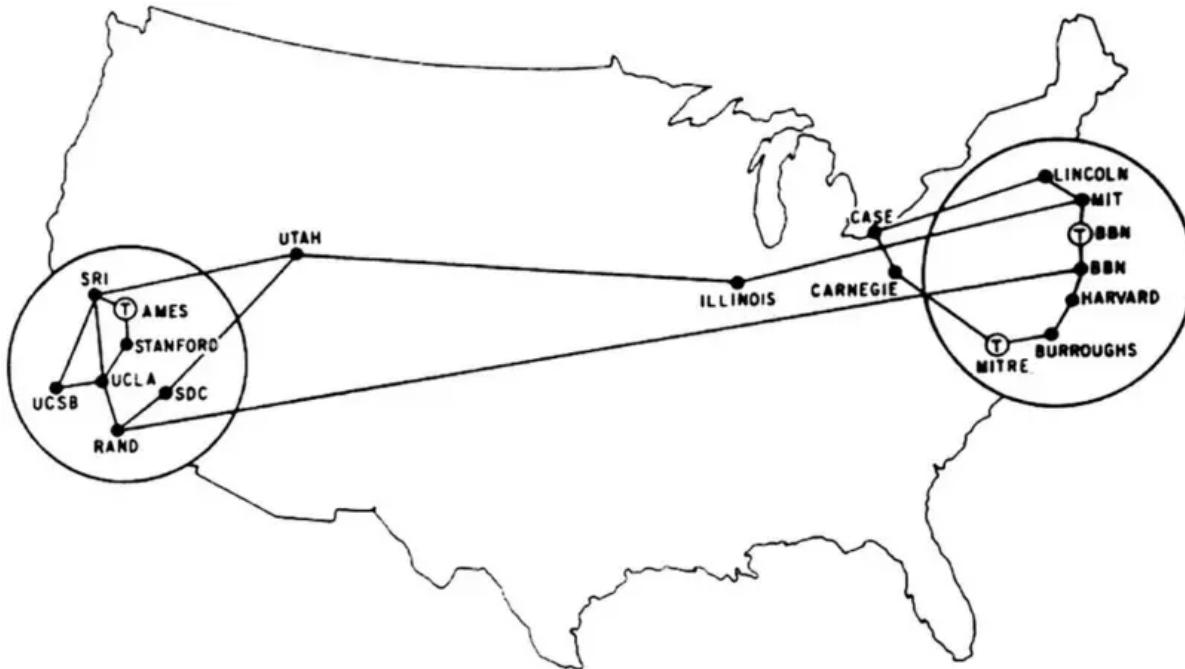


Figura 1.2: Distribución geográfica de los nodos de ARPANET en 1971 (BBC Brasil 2019).

hoy en día. Una de las grandes ventajas de TCP/IP fue que permitió que diferentes tipos de redes se pudiesen comunicar entre sí de manera estándar. Es decir, empezamos a tener redes formadas por redes interconectadas. En este momento fue cuando se empezó a hablar del término “Internet” para describir esta red de redes interconectadas. En segundo lugar, ARPANET se dividió en 1983, creándose MILNET como una red independiente para fines militares, mientras ARPANET continuó creciendo en su uso académico. En tercer lugar, el CERN empezó a interconectar sus ordenadores utilizando TCP/IP, sentando la base para el último evento. En cuarto y último lugar, Tim Berners-Lee, trabajando en el CERN, inventó la World Wide Web en 1989-1990. Propuso un sistema de intercambio de información basado en hipertexto así como las direcciones URL, el protocolo HTTP y el lenguaje HTML, que son omnipresentes hoy en día.

Los años 90 fueron testigos de la transformación de Internet de un proyecto académico a una infraestructura comercial global. El tráfico de ARPANET fue absorbido por Internet y se desmanteló en 1990. En 1991, la World Wide Web fue anunciada públicamente cuando Tim Berners-Lee publicó el primer sitio web. Ese mismo año se creó el primer navegador web gráfico, Mosaic, desarrollado en la Universidad de Illinois en 1993, que revolucionó la experiencia de usuario al permitir la visualización de imágenes junto con texto. La eliminación de las restricciones comerciales sobre el uso de Internet por parte de la National Science Foundation en 1995 marcó un punto de inflexión crucial. Comenzaron a aparecer los primeros proveedores comerciales de servicios de Internet (ISP) como America Online (AOL), que llevó Internet a

millones de hogares. Las empresas empezaron a ver el potencial no solo como un medio de comunicación, sino como una plataforma de negocio, surgiendo los primeros sitios de comercio electrónico como Amazon (1995) y eBay (1995). Yahoo! se estableció como uno de los primeros directorios web populares, mientras que motores de búsqueda como AltaVista comenzaron a indexar la creciente web. A finales de la década, Google fue fundado en 1998, revolucionando la búsqueda en Internet. Finalmente, se completó la transición de Internet de un proyecto gubernamental y académico a una infraestructura comercial global.

El cambio de milenio trajo la adopción masiva de Internet, inicialmente centrada en la conectividad de banda ancha en hogares y oficinas. La llamada “burbuja de las punto-com” explotó en 2000-2001, pero esto no frenó la innovación. Surgió la Web 2.0 a mediados de la década, caracterizada por sitios interactivos y generados por usuarios. Plataformas como MySpace (2003), Facebook (2004), YouTube (2005) y Twitter (2006) transformaron Internet en un medio social y participativo. La revolución móvil comenzó realmente con el lanzamiento del iPhone en 2007, que democratizó el acceso a Internet desde dispositivos móviles. Esto fue seguido por el desarrollo del sistema operativo Android y la proliferación de smartphones. El concepto de “Internet de las Cosas” (IoT) comenzó a materializarse con dispositivos domésticos inteligentes, wearables y sensores conectados. La década de 2010 vio el surgimiento de la computación en la nube con servicios como Amazon Web Services, la popularización de las redes sociales móviles, el auge del comercio electrónico móvil, y el desarrollo de tecnologías como la realidad virtual y aumentada. Más recientemente, la inteligencia artificial, el machine learning, la tecnología blockchain y las criptomonedas han redefinido las posibilidades de Internet.

El número de dispositivos conectados se ha incrementado exponencialmente, pasando de millones en los 90 a miles de millones en la actualidad, marcando el desarrollo de nuevas tecnologías como 5G para soportar el creciente número de dispositivos. En la Figura 1.3 podéis apreciar cómo se han ido incrementando exponencialmente, y esta tendencia está lejos de revertirse. Las redes sociales, herramientas de teletrabajo, VoIP y videollamadas, inteligencia artificial, streaming de video, realidad virtual y aumentada, y otras muchas aplicaciones hacen que no solo se incremente el número de dispositivos conectados, sino también las necesidades de ancho de banda y tiempos de respuesta cada vez más exigentes.

En conclusión, la evolución de Internet (ver resumen en la Figura 1.4) desde sus orígenes militares como ARPANET hasta convertirse en la infraestructura global actual ilustra una transformación extraordinaria que ha redefinido la sociedad moderna. Lo que comenzó en 1969 como una red experimental de 4 hosts diseñada para resistir ataques nucleares, se ha convertido en un ecosistema interconectado de miles de millones de dispositivos. En los siguientes apartados veremos en detalle la tecnología que sustenta Internet y obtendremos el conocimiento necesario para poder realizar aplicaciones y juegos en red.

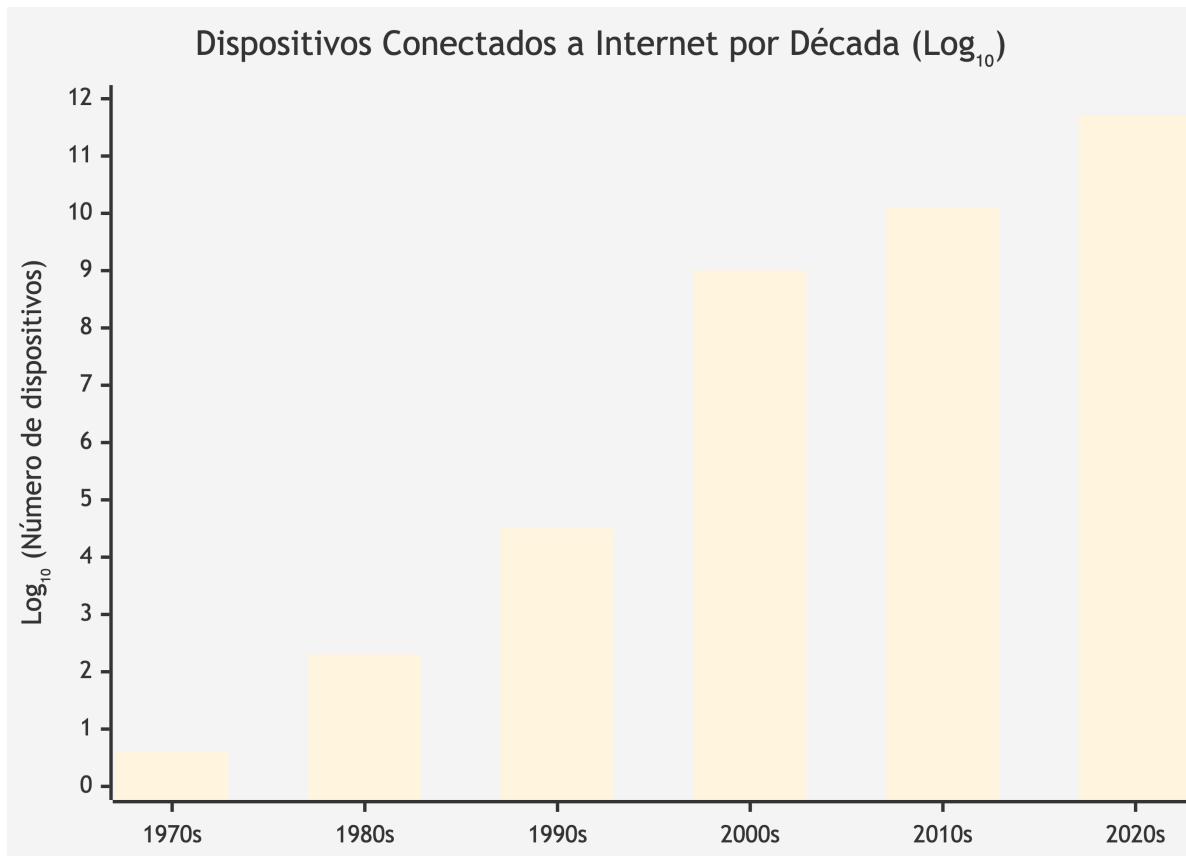


Figura 1.3: Número de dispositivos conectados por década desde ARPANET hasta Internet en 2025 (Ritchie et al. 2023; Analytics 2020).

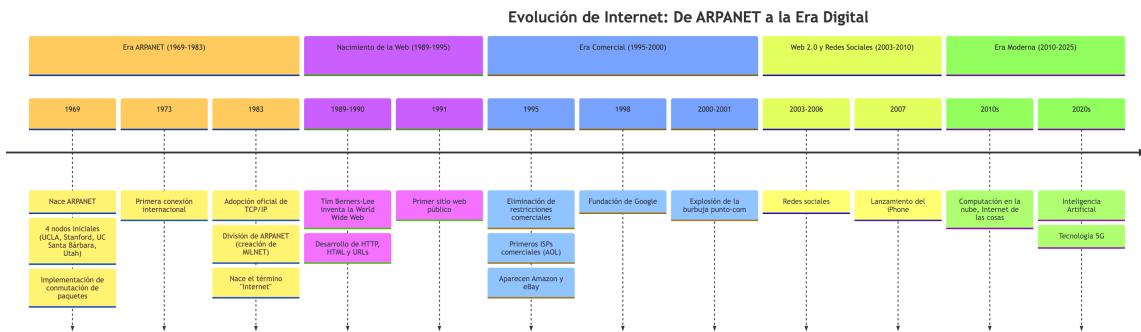


Figura 1.4: Esquema de tiempo de los eventos más significativos desde la creación de ARPANET hasta 2025.

### 1.3. Infraestructura de la red y tecnologías de transmisión

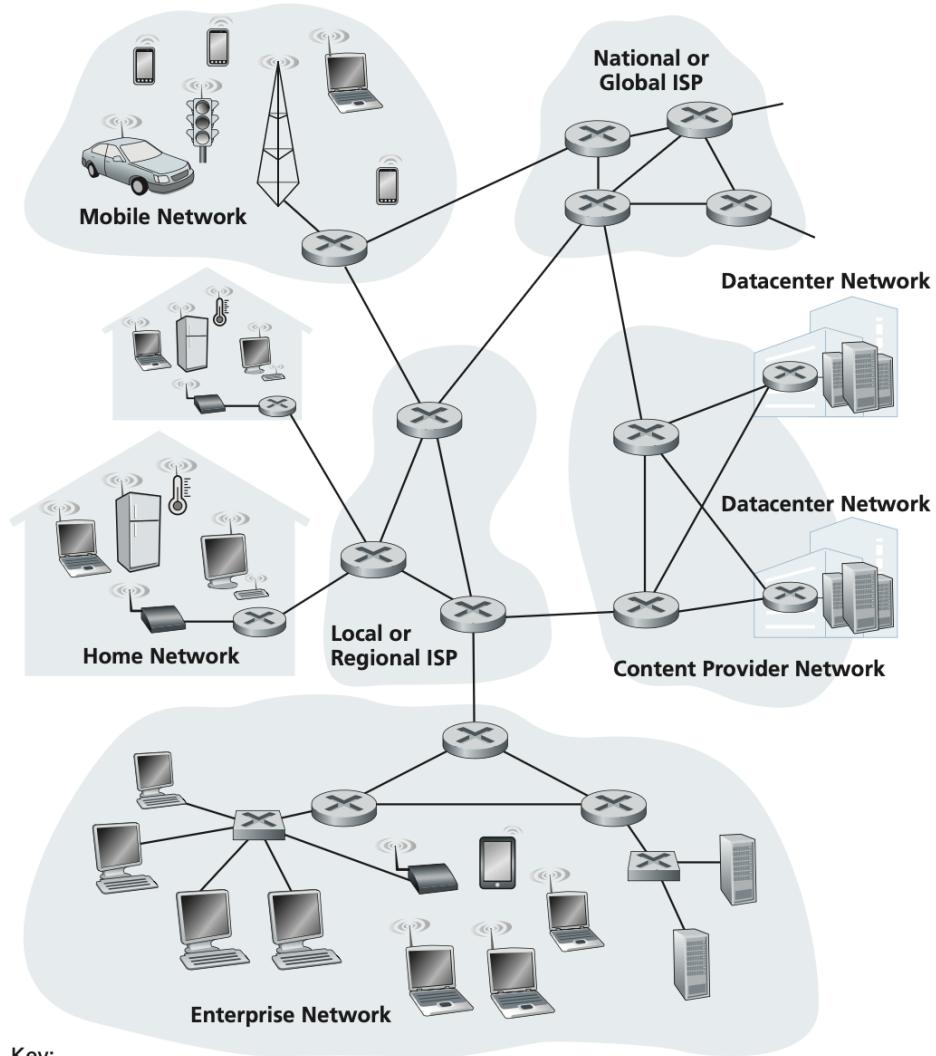
En los capítulos anteriores hemos visto una pequeña introducción a Internet y sus componentes. Ahora pasaremos a ver brevemente la parte física (Hardware) de Internet antes de ver la parte Software en los siguientes capítulos. En la Figura 1.5 tenemos un ejemplo de diagrama donde se muestran los componentes de la red y parte de la taxonomía que veremos en este capítulo.

Empezando por la parte más externa, vamos a hablar de los sistemas terminales (“end systems”). De forma simplificada, podríamos decir que estos son los sistemas que utilizan la Internet, y que el resto de componentes son los que sustentan la red. En esta categoría tendríamos los ordenadores, smartphones, dispositivos inteligentes.. es decir, los componentes conectados. En la jerga de Internet estos componentes se conocen como “host”, por que son los que tienen aplicaciones que funcionan sobre internet. Estos dispositivos se pueden conectar a la red a través de diferentes tecnologías que veremos posteriormente en este capítulo como WiFi o 5g. Los hosts, dependiendo de su uso, también se pueden clasificar como clientes y servidores. Los servidores generalmente ofrecen un servicio que los clientes utilizan. Por ejemplo, cuando hablamos por Whatsapp, nuestro teléfono y el teléfono destino son clientes, y los “ordenadores” de Whatsapp que ofrecen el servidor son servidores. Esto no es clasificación estática y fija, y un cliente puede actuar de servidor también. Este tipo de clasificación la veremos en más detalle en el Capítulo 5.

Moviéndonos a la capa más interna tenemos las redes de acceso (“access network”). Las redes de acceso es la red en la cual se conecta un host con el router (también conocido router de borde, o inglés “edge router”) en el camino hacia el núcleo de la red (core network). El router de borde junto a los hosts también forman parte de lo que se denomina el borde de la red (“edge of the network”). Siguiendo con los ejemplos anteriores, cuando nos conectamos a Internet por WiFi/Ethernet en nuestra casa, universidad, etc, nos conectamos al “router”, que sería el router de borde.

Aquí merece la pena hacer una aclaración técnica sobre el “router” doméstico del ejemplo anterior. En realidad, estos dispositivos son equipos multifunción que integran varias tecnologías: un switch para la red local, un router para el enrutamiento entre redes, y típicamente un punto de acceso WiFi. Cuando nos conectamos por cable o WiFi, técnicamente nos conectamos primero al switch integrado, y cuando la comunicación debe salir hacia Internet, el componente router se encarga del enrutamiento hacia otras redes. Aunque en el uso cotidiano llamamos “router” a todo el dispositivo, es importante entender que internamente realiza múltiples funciones de red.

En el router doméstico, o router de borde, tenemos dos tipos de conexiones principales: la conexión con los hosts y la conexión con el siguiente router. Vamos a ver brevemente los tipos de tecnología para cada caso. Empezando por la conexión host-router, tenemos dos tipos principalmente: conexión cableada tipo Ethernet y conexión inalámbrica WiFi. En la ([tab-ni-infra-host-edge?](#)) podéis ver una comparativa de sus principales características.



Key:

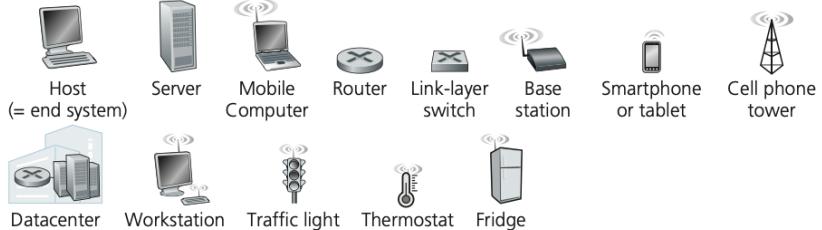


Figura 1.5: Red simplificada (Kurose y Ross 2017).

Tecnología	Medio Físico	Tipo Conexión	Simetría	Velocidad Típica	Alcance	Estado 2025
<b>WiFi 6</b>	Radio 2.4/5/6 GHz	Compartida	Simétrica*	200-400 Mb/s	30-50 m	Estándar
<b>Ethernet</b>	Par trenzado	Dedicada	Simétrica	1000/1000 Mb/s	<100 m	Estándar
<b>4G LTE</b>	Radio móvil	Compartida	Asimétrica	50/15 Mb/s	Varios km	Estable
<b>5G</b>	Radio móvil	Compartida	Asimétrica	300/50 Mb/s	1-5 km	En despliegue

Tabla comparativa de tecnologías de acceso host-router. \*Simétrica en teoría, asimétrica en la práctica.

WiFi 6 es una tecnología de acceso inalámbrico, es decir, no requiere una conexión física entre los dispositivos. Como contrapartida a la flexibilidad de no tener el vínculo físico, el alcance se ve reducido. La máxima distancia entre el router y el dispositivo es típicamente de 30-50 metros, pero puede verse reducida por obstáculos entre ambos. Los estándares WiFi van desde el original 802.11 hasta el más moderno 802.11ax (WiFi 6E), que es capaz de alcanzar velocidades teóricas de hasta 9.6 Gb/s mediante múltiples antenas y técnicas avanzadas. Además, en las últimas versiones se ha incrementado el ancho de banda disponible, incluyendo la banda de 6 GHz, proporcionando espectro adicional para reducir la congestión. La conexión mediante WiFi es simétrica en teoría, aunque en la práctica las velocidades pueden variar según las condiciones del entorno, y del hardware del router y del host. Una de las principales desventajas de los medios inalámbricos es que el medio de transmisión es compartido entre todos los dispositivos, lo que puede causar problemas de congestión cuando hay muchos dispositivos conectados simultáneamente.

Por otra parte, tenemos el acceso tipo Ethernet, que se realiza mediante un cable físico de par trenzado (que explicaremos más adelante). En este caso, el alcance se extiende hasta algo menos de 100 metros. Al ser una conexión física, generalmente no importa qué obstáculos haya entre ambos puntos<sup>4</sup>. Algunas personas han intentado empalmar cables para lograr longitudes superiores a 100 metros, pero esto no funciona adecuadamente. Las causas principales son la degradación de la señal y que los protocolos Ethernet están diseñados asumiendo tiempos específicos de propagación en el cable<sup>5</sup>. Al ser un tipo de conexión dedicada, cuando nos conectamos por cable no tenemos problemas de congestión del medio de transmisión. Las velocidades estándar actuales suelen ser de 1000 Mb/s (Gigabit Ethernet), aunque existen estándares más rápidos como 10 Gigabit Ethernet.

<sup>4</sup>Técnicamente sí pueden afectar ciertos factores, por ejemplo, campos magnéticos intensos o interferencias electromagnéticas sobre el cable Ethernet.

<sup>5</sup>La resistencia del cable se incrementa linealmente con la distancia, aumenta la probabilidad de interferencias electromagnéticas, y se degrada la relación señal-ruido, entre otros factores.

Finalmente, tenemos las tecnologías de acceso móvil como alternativa de conectividad. Tanto **4G LTE** como **5G** utilizan ondas de radio en el espectro móvil licenciado para conectar dispositivos con las torres de telefonía, que actúan como puntos de acceso a la red del operador. Su principal ventaja es el amplio alcance (varios kilómetros para 4G, 1-5 km para 5G según la banda), lo que las hace ideales para ubicaciones sin infraestructura fija o como backup de conectividad. Ambas tecnologías son asimétricas y utilizan un medio compartido, con 4G LTE ofreciendo velocidades típicas de 50/15 Mb/s y 5G alcanzando hasta 300/50 Mb/s en condiciones reales. El 5G representa una evolución significativa al usar un espectro más amplio, incluyendo frecuencias milimétricas, aunque presenta un compromiso entre velocidad y alcance: las frecuencias más altas proporcionan mayor velocidad pero menor penetración. Mientras 4G LTE está completamente desplegado, 5G se encuentra en fase de despliegue activo con cobertura variable según ubicación y operador<sup>6</sup>.

Ahora pasaremos a la conexión del router de borde con el siguiente router. Las tecnologías disponibles

Tecnología	Medio Físico	Tipo Conexión	Simetría	Velocidad Típica	Alcance	Estado
						2025
<b>Dial-up</b>	Par trenzado	Dedicada	Simétrica	56 kb/s	Ilimitado*	Obsoleta
<b>DSL/VDSL</b>	Par trenzado	Dedicada	Asimétrica	50/15 Mb/s	<3 km de central	En declive
<b>Cable HFC</b>	Coaxial/Fibra	Compartida	Asimétrica	300/30 Mb/s	Red local	Estable
<b>FTTH PON</b>	Fibra óptica	Compartida	Simétrica	1000/1000 Mb/s	<20 km	En expansión
<b>FTTH P2P</b>	Fibra óptica	Dedicada	Simétrica	10000/10000 Mb/s	<40 km	Premium
<b>Satelital</b>	Microondas	Compartida	Asimétrica	100/20 Mb/s	Global	Nicho

Para la conexión entre el router de borde y el siguiente router en la jerarquía de red, disponemos de diversas **tecnologías WAN** (Wide Area Network) que han evolucionado significativamente. Las tecnologías más tradicionales como **Dial-up** (56 kb/s) están obsoletas, mientras que **DSL/VDSL** (50/15 Mb/s típicas) se encuentran en declive debido a sus limitaciones de distancia (<3 km de la central telefónica) y asimetría inherente del par trenzado. El **Cable HFC** (Hybrid Fiber-Coaxial) ofrece velocidades superiores (300/30 Mb/s) mediante una combinación de fibra óptica hasta el vecindario y cable coaxial hasta el hogar, aunque mantiene asimetría y medio compartido. Las tecnologías de **fibra óptica** representan el estado del

---

<sup>6</sup>El rendimiento de ambas tecnologías puede degradarse significativamente en áreas de alta densidad poblacional o durante horas pico debido a la congestión del medio compartido.

arte: **FTTH PON** (Fiber-to-the-Home Passive Optical Network) proporciona 1000/1000 Mb/s simétricos con medio compartido y está en expansión activa, mientras que **FTTH P2P** (Point-to-Point) ofrece conexiones dedicadas de hasta 10000/10000 Mb/s para aplicaciones premium. Como alternativa para ubicaciones remotas, la **conectividad satelital** proporciona cobertura global con velocidades de 100/20 Mb/s, aunque con mayor latencia y asimetría, ocupando un nicho específico donde otras tecnologías no son viables.

Finalmente, llegamos a la última capa, denominada el núcleo de la red. El núcleo de la red es una compleja jerarquía de redes interconectadas que trabajan conjuntamente para proporcionar conectividad global. Por contextualizar las tres partes de la red mencionadas hasta ahora vamos a ver un ejemplo. Supón que una persona A (host) envía un mensaje (carta) a otra persona B (otro host). La persona A deposita la carta en correos, que sería el router de frontera. Todo el proceso del envío de la carta desde correos (router de frontera de A) hasta llegar al buzón de B (router de frontera de B) sería el núcleo de la red.

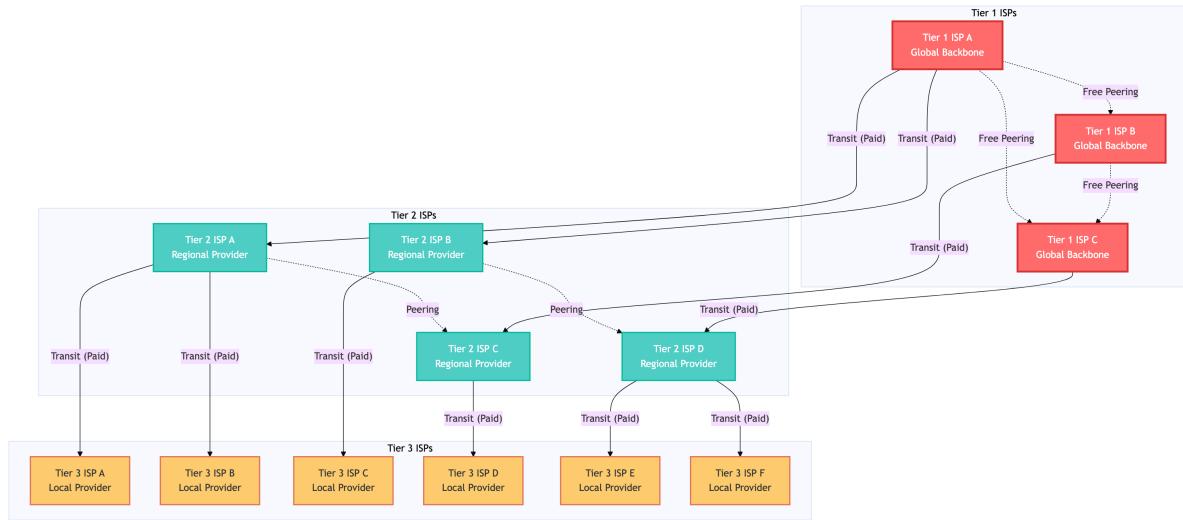
Después de haber ejemplificado su estructura, vamos a indagar en cómo está estructurado el núcleo de la red. Primeramente hablaremos de su estructura, que comentamos en la introducción tiene una estructura descentralizada, lo que permite que el sistema sea mas robusto y escalable. Los componentes de esta red que nos proporcionan interconexión con otras redes se denominan ISP (proveedores de servicio de Internet, del inglés “Internet Service Providers”). Los ISPs se organizan en tres niveles, cada uno con características y roles específicos en el ecosistema global de conectividad.

Los proveedores de Nivel 1 (Tier 1 en inglés) forman la élite de Internet, operando las redes troncales globales de más alta capacidad. Estas organizaciones incluyen empresas como Cogent, AT&T, Verizon, TeliaSonera y Telefónica. Los proveedores Tier 1 mantienen infraestructuras que abarcan continentes enteros con enlaces de 10-100 Gb/s y routers de rendimiento extremo capaces de procesar millones de paquetes por segundo. Entre los proveedores ISP Tier 1 se pueden mandar mensajes sin costo alguno mediante acuerdos de “peering” gratuito. Esto mantiene la exclusividad del estatus Tier 1, ya que se deben alcanzar acuerdos con todos los Tier 1 existentes antes de ser considerado Tier 1.

Los ISP de Nivel 2 (Tier 2 en inglés) operan redes regionales o nacionales más pequeñas que se conectan a Internet a través de uno o más proveedores Tier 1. Pagan a los Tier 1 por “tránsito” - el servicio de llevar su tráfico a destinos que no pueden alcanzar directamente. Sin embargo, los Tier 2 también establecen conexiones directas entre sí cuando es mutuamente beneficioso, reduciendo los costos de tránsito y mejorando el rendimiento para rutas comunes. Estos proveedores sirven como el tejido conectivo esencial de Internet, agregando tráfico de numerosos proveedores más pequeños y proporcionando redundancia y rutas alternativas. Su posición intermedia les permite ofrecer servicios especializados y soporte más personalizado que los grandes Tier 1, mientras mantienen conexiones globales a través de sus relaciones de tránsito.

Los ISP de Nivel 3 son los proveedores de acceso que conectan directamente a usuarios finales - hogares, pequeñas empresas, y organizaciones locales. Estos proveedores compran conectividad

a Internet de ISP de niveles superiores y generalmente no mantienen conexiones directas entre sí. Su valor radica en el conocimiento local, servicio personalizado, y la infraestructura de “última milla” que lleva Internet directamente a los usuarios finales.



## 1.4. Modelos de Referencia de Redes

Una vez visto la parte más hardware de Internet, vamos a pasar a introducir la parte software. En concreto, vamos a hablar de cómo se estructura la parte software de la comunicación en red. Primero, vamos a introducir dos conceptos software muy importantes: Las arquitecturas por capas (“Layered architectures”) y la encapsulación.

Las arquitecturas por capas es una forma de estructurar una aplicación software en capas (componentes) donde cada capa tiene una responsabilidad específica y bien definida. Cada capa proporciona servicios a la capa superior y utiliza los servicios de la capa inferior, creando una jerarquía organizada. Esta organización permite que cada capa se pueda desarrollar, modificar y mantener de forma independiente, siempre que mantenga la misma interfaz con las capas adyacentes.

En el contexto de las redes de comunicación, esta aproximación arquitectónica es fundamental porque permite dividir la complejidad de la comunicación en red en problemas más pequeños y manejables. Por ejemplo, una capa puede encargarse únicamente del enrutamiento de datos, mientras que otra se ocupa exclusivamente de la detección y corrección de errores. Esta separación de responsabilidades hace que el sistema sea más modular, escalable y fácil de debuggear.

### Encapsulación

La encapsulación, por su parte, es el proceso mediante el cual cada capa añade su propia información de control (headers) a los datos que recibe de la capa superior, creando una nueva unidad de datos que pasa a la capa inferior. De esta manera, cada capa trata los datos de las capas superiores como una carga útil (payload) a la que simplemente añade su propia información de control, sin necesidad de entender o modificar el contenido interno de esos datos.

Bajo estos dos conceptos se definen los dos modelos más importantes: el modelo OSI y el modelo TCP/IP. En la Figura 1.6 podemos ver los modelos OSI y TCP/IP divididos en sus diferentes capas y cuál es la equivalencia entre ambos.

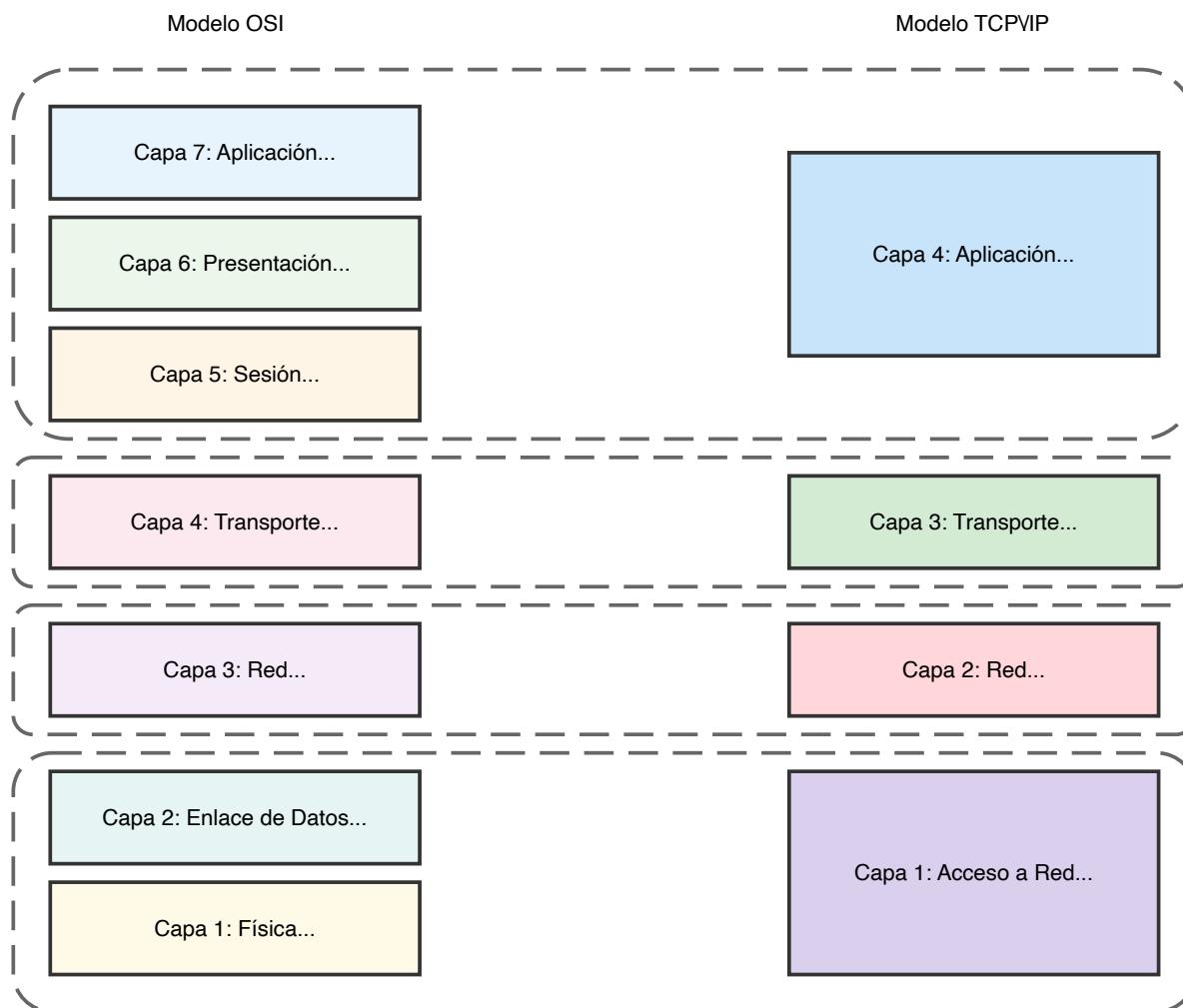


Figura 1.6: Modelos OSI y TCP/IP.

El modelo OSI, desarrollado por la Organización Internacional de Normalización (ISO) en 1984, es un modelo conceptual de siete capas que describe cómo diferentes sistemas de red pueden comunicarse entre sí. El modelo TCP/IP, también conocido como modelo de Internet, es el modelo práctico utilizado en Internet, desarrollado por DARPA con cuatro capas que corresponden aproximadamente a las capas OSI, pero con una estructura más simple y práctica. Aunque en la práctica se usa más el modelo TCP/IP, OSI sigue siendo fundamental para entender los principios de las comunicaciones de red. A continuación, explicamos cada nivel de funcionalidad, mostrando cómo se mapean entre ambos modelos:

**Nivel de Aplicación** OSI: Capas 7, 6 y 5 (Aplicación, Presentación y Sesión) TCP/IP: Capa de Aplicación

En el modelo OSI, este nivel se divide en tres capas separadas, mientras que TCP/IP las agrupa en una sola capa más práctica:

- Funcionalidad de Aplicación (OSI Capa 7): Es la capa más cercana al usuario final. Aquí residen las aplicaciones de red como navegadores web (HTTP/HTTPS), clientes de correo electrónico (SMTP, POP3, IMAP), transferencia de archivos (FTP) y servicios de nombres de dominio (DNS). Esta capa proporciona servicios directamente a las aplicaciones del usuario.
- Funcionalidad de Presentación (OSI Capa 6): Se encarga de la traducción, cifrado y compresión de datos. Convierte los datos del formato de aplicación al formato de red y viceversa. Maneja diferentes representaciones de datos (ASCII, EBCDIC), cifrado/descifrado y compresión/descompresión.
- Funcionalidad de Sesión (OSI Capa 5): Establece, mantiene y termina las sesiones de comunicación entre aplicaciones. Controla los diálogos/conexiones entre ordenadores, implementa checkpoints para recuperación en caso de fallo y gestiona el control de acceso.

En TCP/IP, todas estas funcionalidades están integradas en la Capa de Aplicación, que incluye protocolos como HTTP/HTTPS para web, SMTP para correo electrónico, FTP para transferencia de archivos, DNS para resolución de nombres, y muchos otros que proporcionan servicios directos a los usuarios. Esta aproximación más práctica evita la complejidad de separar artificialmente funciones que a menudo están estrechamente relacionadas.

**Nivel de Transporte** OSI: Capa 4 (Transporte) TCP/IP: Capa de Transporte Este nivel es prácticamente idéntico en ambos modelos. Proporciona transferencia de datos confiable entre sistemas finales, maneja el control de flujo, la corrección de errores y la segmentación/reensamblado de datos.

Los protocolos principales son:

- TCP (Transmission Control Protocol): Ofrece comunicación confiable con control de flujo, corrección de errores y garantía de entrega ordenada.
- UDP (User Datagram Protocol): Ofrece comunicación rápida pero sin garantías de entrega, ideal para aplicaciones en tiempo real.

### **Nivel de Red/Internet** OSI: Capa 3 (Red) TCP/IP: Capa de Internet

Ambos modelos manejan esta funcionalidad de manera muy similar. Se encarga del enrutamiento de paquetes a través de múltiples redes, determinando la mejor ruta para enviar datos desde el origen hasta el destino. El protocolo principal es IP (Internet Protocol), junto con protocolos auxiliares como:

- ICMP: Para mensajes de control y error.
- ARP: Para resolución de direcciones (en TCP/IP).
- Protocolos de enrutamiento: Como OSPF y BGP.

### **Nivel de Acceso Físico** OSI: Capas 2 y 1 (Enlace de Datos y Física) TCP/IP: Capa de Acceso a la Red

El modelo OSI separa estas funciones en dos capas distintas, mientras que TCP/IP las combina por practicidad:

- Funcionalidad de Enlace de Datos (OSI Capa 2): Proporciona transferencia de datos libre de errores entre nodos adyacentes. Se divide en dos subcapas: LLC (Logical Link Control) y MAC (Media Access Control). Maneja la detección y corrección de errores a nivel de enlace y controla el acceso al medio físico.
- Funcionalidad Física (OSI Capa 1): Define las características eléctricas, mecánicas y funcionales para activar, mantener y desactivar el enlace físico. Especifica voltajes, velocidades de datos, conectores y otros aspectos del medio de transmisión (cable, fibra óptica, radio).

En TCP/IP, la Capa de Acceso a la Red combina ambas funcionalidades, encargándose de la transmisión de datos en la red local específica, incluyendo tecnologías como Ethernet, WiFi, y otros protocolos de acceso al medio.

### **Diferencias Clave Entre los Modelos**

- Complejidad: OSI tiene 7 capas vs 4 en TCP/IP, siendo OSI más detallado teóricamente pero TCP/IP más práctico.
- Uso real: TCP/IP es el modelo usado en Internet, mientras que OSI es principalmente un modelo de referencia educativo.
- Flexibilidad: TCP/IP agrupa funcionalidades relacionadas, evitando separaciones artificiales que raramente se implementan por separado en la práctica.
- Evolución: TCP/IP evolucionó con Internet, mientras que OSI fue diseñado como estándar teórico antes de su implementación masiva.

## 1.5. Rendimiento

Por último, vamos a cerrar esta introducción a las redes de telecomunicaciones describiendo brevemente los factores presentes en su rendimiento. Primero, vamos a conceptualizarlo con un ejemplo simplificado. Supongamos que la red de telecomunicación es una carretera entre dos puntos y los paquetes son los vehículos. ¿Cómo podríamos medir el rendimiento de este sistema? Las dos métricas más sencillas serían el tiempo en recorrer la carretera y la cantidad de vehículos que pueden circular a la vez. La primera métrica se conoce como latencia, y está influenciada en nuestro ejemplo por la velocidad del medio, y la segunda se conoce como la tasa de transferencia efectiva (throughput), que sería el número de carriles de las carreteras. El objetivo, bajo estas dos métricas, sería que los vehículos fueran lo más rápido posible aprovechando todos los carriles, consiguiendo que el número de vehículos que llega sea lo más alto posible.

El ejemplo es muy simple, pero nos ha ayudado a introducir dos conceptos clave, la latencia y el throughput. En este capítulo veremos cuáles son los principales factores que influyen en estos dos conceptos cuando en lugar de tener una carretera, tenemos varias carreteras con conexiones entre ellas y no todos los vehículos van al mismo sitio. Las conexiones entre las carreteras, es decir, las redes, se realiza a través de routers como hemos comentado en los capítulos anteriores.

Primero, nos vamos a centrar en el throughput, que es la cantidad de datos real que podemos transmitir por unidad de tiempo. Generalmente se mide en Mb/s o Gb/s. El throughput a veces se mide de manera instantánea pero también se puede considerar como media de un periodo de tiempo. El throughput está limitado por el componente más “lento” en el camino entre dos puntos. Por ejemplo, si estamos descargando información y el medio tiene un throughput de 1Gb/s pero el servidor solo es capaz de proporcionar 100Mb/s, el throughput resultante será 100Mb/s.

Un término asociado al throughput es el ancho de banda (bandwidth). El bandwidth es la capacidad máxima teórica del canal de comunicación, es decir, la cantidad máxima de datos que puede transmitir por unidad de tiempo en condiciones ideales. Es decir, es el límite físico. Por otra parte, el throughput como dijimos es la cantidad real que obtenemos condiciones reales.

### Nota

Es importante no confundir MB/s con Mb/s (u otros pares como GB/s con Gb/s). En informática se suele hablar en MB/s, es decir, MegaBytes por segundo, mientras que en telecomunicaciones se suele hablar en Mb/s. Es una diferencia importante ya que un MB/s es 8 veces más velocidad que un Mb/s.

Ahora pasaremos a la latencia de red, y los factores que la definen. La latencia es el tiempo total que tarda un paquete en viajar desde el origen hasta el destino. Esta latencia no se mide

únicamente con el tiempo teórico de propagación por el medio, sino que es la suma de varios factores. Primero nos enfocaremos en los factores que afectan a un único paquete:

- **Retardo de procesamiento ( $d_{proc}$ ):** El retardo de procesamiento es el tiempo que tarda un router en procesar el paquete. Esto incluye, comprobar la integridad del paquete (checksum), determinar cuál es el siguiente salto y otros procesos adicionales del protocolo. En los routers modernos este proceso normalmente es de microsegundos en condiciones normales, pero puede incrementarse en caso de congestión o políticas adicionales. Este procesamiento se lleva a cabo en hardware especializado (ASICs), pero en determinadas circunstancias es posible que sea necesario inspeccionar el paquete mediante software, como por ejemplo en Deep Packet Inspection, que se suele utilizar para monitorizar la red por seguridad o para forzar políticas Kurose y Ross (2017).
- **Retardo de cola ( $d_{queue}$ ):** El retardo de cola ocurre una vez se ha procesado el paquete con su correspondiente retardo de procesamiento. En este momento, el paquete es colocado en un buffer con la información necesaria para determinar el siguiente salto. El retardo de cola es el tiempo que tarda el paquete en ser enviado al siguiente salto. Si hay poco tráfico, el retardo de cola será casi nulo, en cambio, si hay mucho tráfico este retardo crecerá considerablemente.
- **Retardo de propagación ( $d_{prop}$ ):** El retardo de propagación es el tiempo que tarda en viajar un paquete por el medio, como puede ser la fibra óptica o 5G, o generalmente, una combinación de varias, ya que de un punto a otro puede haber diferentes medios. El retardo, por lo tanto, es la suma de los retardos de cada uno de los medios. El retardo de un medio, se calcula como  $d/s$ , donde  $d$  es la longitud del medio y  $s$  es la velocidad del medio. Por contextualizar con datos las velocidades de los medios, la fibra óptica y el cable coaxial tienen una velocidad (en promedio) de aproximadamente el 67% de la velocidad de la luz y en el 5G la velocidad de la luz Kurose y Ross (2017). Este retardo está limitado por las leyes de la física.

Estos factores afectan a un único paquete, pero generalmente cuando enviamos algo es demasiado grande como para entrar en un paquete y se divide en varios paquetes, que posteriormente se recomponen en el destino. Por lo tanto, tenemos otro tipo de retardo, que tiene en cuenta la cantidad de información que queremos enviar:

- **Retardo de transmisión ( $d_{trans}$ ):** Este retardo está determinado por el tamaño de la información que queremos enviar ( $L$ ) y la velocidad del enlace ( $R$ ), es decir,  $L/R$ . Generalmente este retardo es predecible y constante, pero puede variar significativamente entre tecnologías de red. La velocidad del enlace es el throughput.

Una vez definidos todos los factores, podemos expresar el retardo total como:

$$d_{total} = d_{proc} + d_{queue} + d_{prop} + d_{trans}$$

Vamos a ver un ejemplo “real” de retardo comparando dos enlaces, uno con fibra y otro con 5G. Haremos la comparación hasta el primer router (router de borde) incluido:

- **Retardo de propagación:** Como comentamos previamente, el 5G se propaga a la velocidad de la luz y la fibra aproximadamente al 67% de la velocidad de la luz. Por lo tanto, el 5G es más rápido.
- **Retardo de procesamiento:** En 5G tenemos retardo debido a la estación de radio, la decodificación y la gestión de los recursos de radio (aproximadamente unos 4ms). En cambio, en la fibra este proceso es mucho más rápido, necesitando aproximadamente unos 0.1ms por salto. La fibra suele ser mucho más rápida.
- **Retardo de cola:** A una estación suele haber conectados cientos de dispositivos, puede haber interferencias y además suelen ser dependientes del clima. Un ejemplo de esto lo podréis haber vivido cuando estáis en un concierto con miles de personas y no funciona bien la conexión debido a la congestión. En el caso de la fibra óptica suele haber menos congestión, el número de usuarios es predecible y los sistemas cuentan con buffers más grandes y eficientes.
- **Retardo de transmisión:** El throughput en 5G es inferior a 1Gb/s, mientras que en fibra pueden llegar actualmente a 10 Gb/s.

### Latencia vs Throughput

La latencia mide cuánto tiempo tarda en llegar la información y el throughput mide cuánta información puede viajar simultáneamente por el canal de comunicación. Volviendo al ejemplo de la carretera: la latencia sería el tiempo que tarda un vehículo en recorrer toda la carretera de extremo a extremo, mientras que el throughput sería la cantidad total de vehículos que pueden pasar por la carretera en un periodo determinado (relacionado con el número de carriles y la densidad de tráfico).

Un concepto asociado a la latencia de suma importancia en las aplicaciones en red, especialmente los juegos interactivos es el jitter. Cuando enviamos varios paquetes podemos calcular una latencia promedio, ya que no todos los paquetes tardarán lo mismo debido a las condiciones de red y diferentes rutas. En aplicaciones altamente interactivas tener una latencia promedio baja es indispensable. Sin embargo, considera este pequeño ejemplo donde se envían 4 paquetes.

- Escenario 1: Los paquetes tardan 50ms, 52ms, 48ms, 51ms
- Escenario 2: Los paquetes tardan 28ms, 68ms, 43ms, 62ms.

En ambos escenarios los paquetes tienen una latencia promedio de 50.25ms. Sin embargo, la variación entre los paquetes es elevada. En el primer caso, la variación es de 1.48ms mientras que en el segundo es de 15.82ms. Esta variabilidad se conoce como jitter. Un jitter alto puede ocasionar voz entrecortada o saltos en videoconferencias o degradación de la calidad en videojuegos. En el caso de los videojuegos, se suelen utilizar buffers para realizar interpolaciones de los elementos de red y así tener un juego más fluido.

Finalmente, vamos a ver un último factor que no se ajusta a los anteriores. Hasta ahora hemos asumido que todos los paquetes que enviamos llegan correctamente a su destinatario. Pero esto no es siempre cierto. Por ejemplo, si un router está congestionado y tiene su buffer lleno, descartará los paquetes. Si un paquete se corrompe debido a alteraciones (e.g., campos electromagnéticos, radiación solar<sup>7</sup>) un router de tránsito lo podrá descartar. Esto forma parte del protocolo de Internet. Otros protocolos, en capas superiores como por ejemplo TCP, tienen en cuenta estas situaciones y reenvían el paquete cuando determinan que no ha llegado a su destino.

**Aplicaciones Prácticas: Videojuegos** Cuando estamos diseñando aplicaciones en red tenemos que tener en cuenta estos retardos, pues pueden hacer nuestra aplicación inutilizable. En el caso de los videojuegos, los requisitos de retardo máximo vendrán dados dependiendo del tipo de juego, por ejemplo Claypool y Claypool (2006):

- Real-Time Strategy (RTS): Tolerancia media (100-200ms) debido a su naturaleza estratégica.
- Turn-Based Games: Tolerancia alta (500ms+) debido a que los turnos son discretos.
- First-Person Shooters (FPS): Baja tolerancia (20-50ms) para juegos competitivos.
- Fighting Games: Tolerancia muy baja (1-3 frames, ~16-50ms).
- Racing Games: Tolerancia baja o moderada (50-100ms) dependiendo del realismo.
- MMORPGs: Tolerancia variable dependiendo de la actividad, por ejemplo combates vs social.

Estos tiempos se miden en RTT (Round Trip Time), que involucra el tiempo entre que se manda el mensaje, se procesa en el servidor, y obtenemos la respuesta de vuelta en el cliente.

---

<sup>7</sup>La radiación cósmica produce cambios de bits en dispositivos electrónicos, que se denominan SEU (Single Event Upset). Estos cambios suelen afectar a las DRAM, SRAM y ASICs. Contrario a la intuición, es algo relativamente frecuente, y ocurre con una tasa aproximada de 1 error por cada 256MB por día a nivel del mar. Cuanto más altitud (o dicho de otra forma, más cerca del espacio), esta tasa se incrementa considerablemente. A nivel de red esto suele ocurrir en los routers.

## 2 Capa de Acceso a la Red

La Capa de Acceso a la Red se encarga de la transmisión física de datos entre dispositivos directamente conectados en una red local, manejando tanto los aspectos físicos de la transmisión como el control de acceso al medio compartido. Esta capa combina las funciones de las capas física y de enlace de datos del modelo OSI, proporcionando una interfaz entre los protocolos de red de nivel superior y el hardware de red específico. Su principal responsabilidad es garantizar que los datos puedan transmitirse de manera confiable entre nodos adyacentes en la red.

El capítulo se divide en un apartado donde veremos las principales funciones de la red y después veremos los protocolos definidos en esta capa. Para guiar el aprendizaje, antes veremos un ejemplo del funcionamiento de la Capa de Acceso a Red. No os preocupéis si no entendéis por ahora todos los conceptos, los iremos viendo a lo largo de este capítulo.

Consideremos una red Ethernet típica con un switch central conectando cuatro computadoras (A, B, C, D) con las siguientes direcciones MAC:

- A (Puerto 1): 00:1A:2B:3C:4D:5E.
- B (Puerto 2): 00:2B:3C:4D:5E:6F.
- C (Puerto 3): 00:3C:4D:5E:6F:70.
- D (Puerto 4): 00:4D:5E:6F:70:81.

Como ejemplo, vamos a ver los pasos para el envío de un mensaje desde A a C. Podéis ver la representación en un diagrama de secuencia en la Figura 2.1. Los pasos serían los siguientes:

1. Inicialización del switch. La tabla de direcciones MAC del switch que asocia MAC y puerto está vacía.
2. A quiere enviar el paquete a C, pero no conoce su MAC. Por lo tanto envía una trama ARP broadcast por Ethernet para descubrir la dirección MAC de C.
3. El switch lo recibe en el puerto 1, aprende que la MAC de A está en el puerto 1, y reenvía el paquete a los demás puertos.
4. El switch recibe la respuesta de C desde el puerto 3. Asocia la dirección MAC de C al puerto 3. Reenvia la respuesta a A, que ya sabe que está en el 1.
5. A envía el mensaje con destinatario la dirección MAC de C.
6. El switch recibe el mensaje, busca en su tabla y comprueba que la dirección MAC coincide con el puerto 3 y lo reenvia.

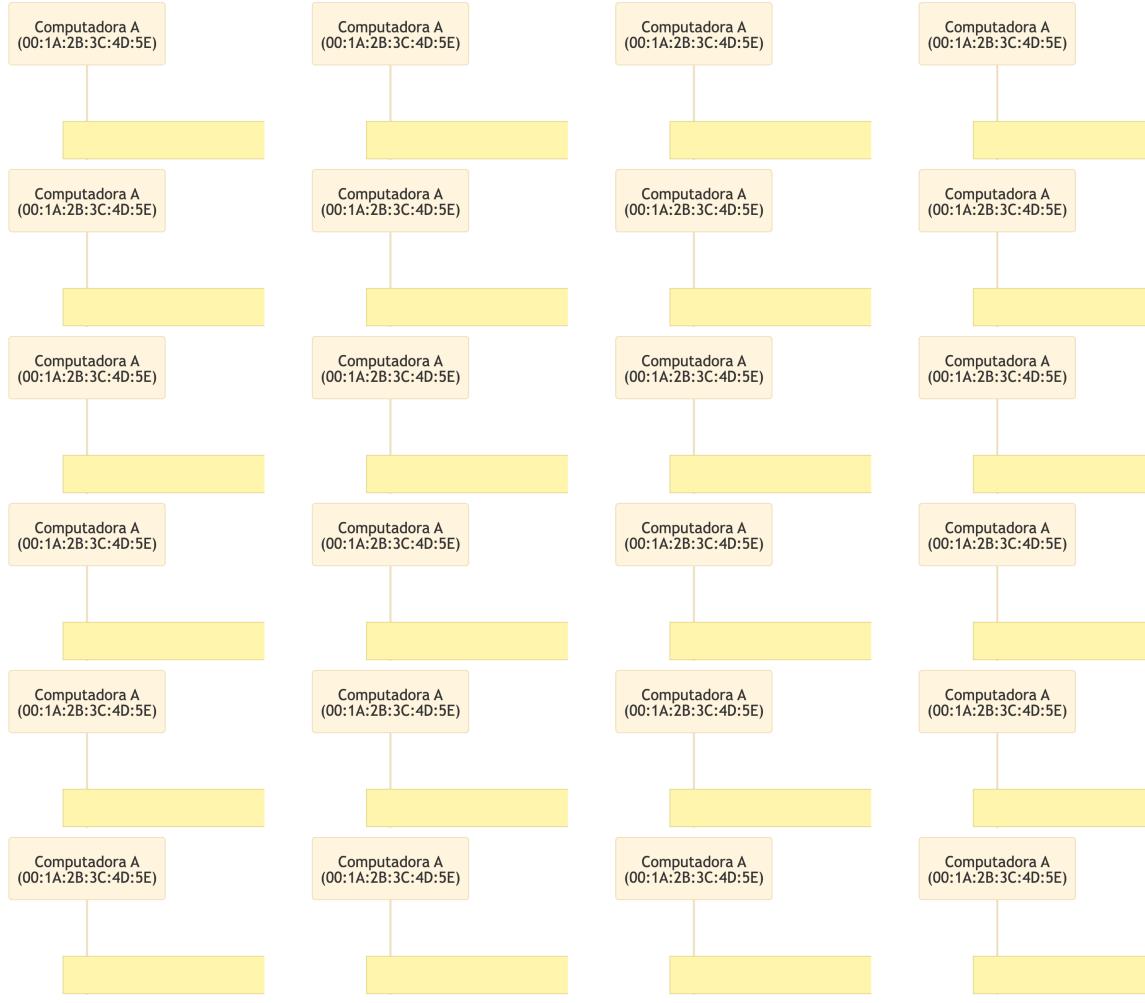


Figura 2.1: Ejemplo de envío de paquetes en una red local donde no se conoce la MAC del destinatario.

## **2.1. Funciones principales de la Capa de Acceso a la Red**

La Capa de Acceso a la Red desempeña múltiples funciones críticas que trabajan en conjunto para garantizar una comunicación eficiente y confiable entre dispositivos en la red local.

### **2.1.1. Control de Acceso al Medio (MAC)**

La función de control de acceso al medio (MAC) es fundamental para coordinar cómo múltiples dispositivos comparten un medio de transmisión común. Esta función implementa diversos algoritmos y protocolos según la tecnología de red utilizada. Su eficiencia determina directamente el rendimiento y la escalabilidad de toda la red local. En redes Ethernet tradicionales de half-duplex (sólo pueden transmitir en una dirección a la vez), los dispositivos emplean el método CSMA/CD (Carrier Sense Multiple Access with Collision Detection), donde primero escuchan el medio antes de transmitir para verificar que esté libre, permiten que múltiples dispositivos accedan al mismo medio compartido, y detectan colisiones durante la transmisión implementando algoritmos de backoff exponencial para programar retransmisiones inteligentes.

Las redes inalámbricas, por el contrario, utilizan CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) debido a que la detección de colisiones es impráctica en el medio radioeléctrico. En este esquema, los dispositivos esperan un tiempo aleatorio antes de transmitir para reducir la probabilidad de colisiones, utilizan mecanismos de acknowledgment para confirmar que la transmisión fue recibida correctamente, e implementan el protocolo RTS/CTS (Request to Send/Clear to Send) para resolver el problema del nodo oculto donde algunos dispositivos no pueden detectar las transmisiones de otros.

Complementariamente, los mecanismos de control de flujo evitan que transmisores rápidos saturen receptores más lentos mediante técnicas como Pause Frames en Ethernet full-duplex (se puede transmitir en ambas direcciones a la vez) que permiten al receptor solicitar pausas temporales, buffer management en switches para absorber ráfagas de tráfico sin pérdida de datos, y rate limiting para controlar dinámicamente la velocidad de transmisión según las condiciones de la red.

### **2.1.2. Direccionamiento Físico**

El direccionamiento físico opera a nivel de hardware y es independiente de los protocolos de capa superior. Utiliza direcciones MAC únicas para identificar cada interfaz de red en el segmento local. Este sistema de direccionamiento es esencial para la entrega precisa de tramas entre dispositivos directamente conectados. Las direcciones MAC son como el DNI del dispositivo, son únicas, estáticas y cada dispositivo tiene una. Están formadas por 48 bits siguen una estructura específica donde los primeros 24 bits constituyen el OUI (Organizationally Unique Identifier) asignado por IEEE a cada fabricante, los últimos 24 bits forman el identificador único del dispositivo asignado por el fabricante, y bits especiales indican si la dirección es individual o

grupal y si está administrada universalmente o localmente. Las MAC se representan mediante octetos 6 octetos separados por dos “：“, como por ejemplo “00:1A:2B:3C:4D:5E”.

#### Dirección MAC

Una dirección MAC es un identificador único asignado a cada tarjeta de red. Están formados por 48 bits donde la primera parte identifica al fabricante, después el dispositivo dentro del fabricante, y por último tiene unos bits especiales.

El sistema soporta tres tipos principales de direccionamiento: unicast para comunicación dirigida a un único dispositivo específico, broadcast utilizando la dirección especial FF:FF:FF:FF:FF:FF para alcanzar todos los dispositivos del segmento simultáneamente, y multicast para dirigir tráfico a grupos específicos de dispositivos identificados por el primer bit configurado en 1.

#### **2.1.3. Detección y Corrección de Errores**

Esta función garantiza la integridad de los datos transmitidos a través del medio físico. Implementa algoritmos como Códigos de Redundancia Cíclica (CRC) para detectar errores de transmisión y mecanismos de retransmisión cuando es necesario. Sin esta función, los datos corruptos podrían propagarse por la red causando problemas de comunicación. Los códigos de CRC son ampliamente utilizados y generan un polinomio matemático basado en los datos originales, agregan el resultado como Frame Check Sequence (FCS) al final de cada trama, y permiten al receptor recalcular el CRC para compararlo con el recibido, detectando efectivamente errores de un solo bit y muchos errores de múltiples bits. Para aplicaciones menos críticas existen checksums simples que realizan una suma aritmética de todos los bytes de datos, son menos robustos que CRC pero requieren menos procesamiento computacional.

Las técnicas más avanzadas incluyen Forward Error Correction (FEC) que no solo detecta sino que corrige errores automáticamente, utiliza códigos como Hamming para corrección de errores de un solo bit y Reed-Solomon para errores en ráfagas, y es especialmente importante en medios inalámbricos donde la interferencia y las condiciones ambientales pueden causar errores frecuentes.

#### **2.1.4. Control de tamaño**

Esta función maneja las limitaciones de tamaño impuestas por diferentes tecnologías de red. Cada tecnología de red define un Maximum Transmission Unit (MTU) específico que determina el tamaño máximo de datos que puede transportar una sola trama, donde Ethernet maneja 1500 bytes de datos, Token Ring típicamente 4464 bytes, FDDI 4352 bytes, y PPP sobre enlaces seriales utiliza valores variables aunque comúnmente 1500 bytes para mantener compatibilidad. Cuando los datos de capas superiores exceden el MTU disponible, la Capa de Acceso a la Red descarta automáticamente el paquete.

### **2.1.5. Sincronización y Temporización**

Esta función coordina el timing entre dispositivos para asegurar la correcta interpretación de las señales digitales. Establece marcos de tiempo comunes para la transmisión y recepción de datos. Es especialmente crítica en redes de alta velocidad donde pequeñas diferencias de timing pueden causar errores de comunicación. La sincronización de reloj es esencial para el funcionamiento correcto de cualquier comunicación digital, abarcando la sincronización de bit para determinar precisamente los límites temporales de cada bit transmitido, la sincronización de trama para identificar inequívocamente el inicio y fin de cada trama de datos, y la sincronización de símbolo necesaria en modulaciones complejas como QAM donde múltiples bits se codifican en un solo símbolo.

### **2.1.6. Gestión de Topología**

Esta función se encarga de descubrir y mantener información sobre la estructura física de la red. Implementa protocolos para detectar enlaces, prevenir bucles y optimizar rutas de comunicación. Permite que la red se adapte automáticamente a cambios en la topología como fallos de enlaces o adición de nuevos dispositivos. El mantenimiento continuo de enlaces se logra mediante mensajes de tipo keepalive que detectan proactivamente fallos de enlace antes de que afecten el tráfico de usuarios. Los protocolos de detección de topología como CDP (Cisco Discovery Protocol) y LLDP (Link Layer Discovery Protocol) permiten que los dispositivos se identifiquen mutuamente y compartan información sobre sus capacidades, mientras que Spanning Tree Protocol previene bucles peligrosos en topologías redundantes que podrían causar tormentas de broadcast.

### **2.1.7. Control de Calidad de Servicio (QoS)**

Esta función prioriza diferentes tipos de tráfico según su importancia y requisitos de rendimiento. Es fundamental para el funcionamiento adecuado de aplicaciones en tiempo real como voz y vídeo. Esto se logra implementando mecanismos de gestión de buffers y scheduling para garantizar que aplicaciones críticas reciban el ancho de banda necesario, y se realiza bajo estándares de clasificación. Los mecanismos de gestión de buffers aseguran que diferentes tipos de tráfico reciban el tratamiento apropiado mediante Weighted Fair Queuing que asigna recursos proporcionalmente según la importancia de cada clase de tráfico, priority queuing que garantiza que el tráfico más crítico siempre tenga precedencia sobre tráfico menos importante, y Random Early Detection que previene congestión descartando proactivamente paquetes menos críticos antes de que los buffers se saturen completamente.

## 2.2. Dispositivos de la Capa de Acceso a la Red

Los **switches** son dispositivos de red que operan en la Capa de Acceso a la Red, específicamente en la subcapa de enlace de datos. Funcionan como elementos centrales que conectan múltiples dispositivos en una red local, creando dominios de colisión separados para cada puerto. Es decir, que la información de dispositivos conectados por diferentes puertos no colisiona entre sí, ya que no están en el mismo medio. Estos dispositivos evolucionaron desde los **bridges** tradicionales (que conectaban solo 2-4 segmentos) hasta reemplazar los **hubs** tradicionales, donde todos los puertos operaban igual con colisiones frecuentes a un sistema donde cada puerto opera independientemente con capacidades full-duplex que permiten transmisión y recepción simultánea, duplicando efectivamente el ancho de banda disponible. Su importancia radica en funcionalidades clave como el aprendizaje automático de direcciones MAC donde construyen dinámicamente tablas que asocian cada dirección con su puerto específico, el reenvío selectivo que envía tramas únicamente al puerto de destino reduciendo tráfico innecesario.

Los switches se categorizan principalmente en dos tipos según sus capacidades de gestión:

- no gestionados: son plug-and-play y se utilizan generalmente en redes pequeñas y domésticas.
- gestionados: ofrecen un control más granular y más capacidades, como seguridad, monitoreo y medidas QoS.

Los **puntos de acceso** son dispositivos fundamentales para la conectividad inalámbrica en la Capa de Acceso a la Red que actúan como traductores entre medios cableados e inalámbricos, manejan la asociación y autenticación de dispositivos inalámbricos, e implementan CSMA/CA para coordinar el acceso al medio y evitar colisiones en el espectro radioeléctrico compartido.

Los **repetidores** extienden el alcance de las redes regenerando señales digitales sin filtrar tráfico ni reducir colisiones, simplemente reciben, amplifican y retransmiten las señales para superar las limitaciones de distancia de los medios físicos. Los amplificadores incluyen RF amplifiers que aumentan la potencia de señales inalámbricas y optical amplifiers que amplifican señales en fibra óptica sin conversión eléctrica, todos debiendo cumplir estrictas regulaciones de potencia de transmisión para evitar interferencia con otros sistemas.

Los **conversores de medio** facilitan la interoperabilidad convirtiendo entre diferentes medios físicos como fibra óptica y cobre, adaptando automáticamente velocidades, y permitiendo extensión de redes existentes o migración gradual a tecnologías más avanzadas.

Tabla de resumen:

Dispositivo	Función Principal	Características Clave	Aplicación Típica
<b>Switches No Gestiónados</b>	Conectividad básica LAN	Plug-and-play, aprendizaje MAC automático, full-duplex	Redes pequeñas y domésticas

Dispositivo	Función Principal	Características Clave	Aplicación Típica
<b>Switches Gestionados</b>	Conectividad LAN avanzada	VLANs, QoS, SNMP, seguridad 802.1X, port mirroring	Redes empresariales
<b>Puntos de acceso</b>	Conectividad inalámbrica	CSMA/CA, traducción cableado-wireless, beamforming	Redes Wi-Fi empresariales
<b>Repetidores</b>	Extensión de alcance	Regeneración de señal, sin filtrado	Superación de límites de distancia
<b>Amplificadores</b>	Amplificación de señal	Aumento de potencia RF/óptica	Enlaces de larga distancia
<b>Conversores de medio</b>	Conversión de medios	Fibra - cobre, adaptación de velocidades	Migración gradual, extensión
<b>Modulador</b>	Conectividad modular	SFP/SFP+/QSFP, intercambiables	Flexibilidad en tipos de conexión

## 2.3. Protocolos

### 2.3.1. Ethernet (IEEE 802.3)

Ethernet es el protocolo dominante en redes cableadas locales. Define tanto el formato de las tramas como los métodos de acceso al medio. Su éxito radica en la simplicidad de implementación, la robustez del diseño, y la capacidad de evolucionar continuamente para satisfacer las demandas crecientes de ancho de banda en entornos empresariales y domésticos. Al principio en Ethernet se tenía una arquitectura de medio compartida, y por ello, se tenían que utilizar técnicas como CSMA/CD para minimizar colisiones. Esto hacía que eficiencia de la red disminuyese. Con la llegada de los bridges y switches, se introdujo una topología de estrella, donde todos están conectados al switch y este crea dominios de colisión independientes, volviendo innecesario el CSMA/CD y mejorando considerablemente la eficiencia de la red.

La estructura de las tramas Ethernet sigue un formato estandarizado que garantiza la interoperabilidad entre dispositivos de diferentes fabricantes. Cada trama comienza con un **preámbulo** de 8 bytes que proporciona sincronización entre el transmisor y receptor, estableciendo el timing necesario para la correcta interpretación de los bits que siguen. Las **direcciones MAC** de destino y origen, de 6 bytes cada una, identifican inequívocamente los dispositivos involucrados en la comunicación, mientras que el campo **Tipo/Longitud** de 2 bytes especifica qué protocolo de capa superior procesa los datos o la longitud de la carga útil cuando es menor a 1536 bytes. El contenido útil reside en el campo de **data/payload** que puede contener entre 46 y 1500 bytes de información útil, con padding automático cuando los datos son menores al mínimo requerido, y finalmente el **Frame Check Sequence** de 4 bytes implementa detección de errores CRC permitiendo al receptor verificar la integridad de toda la trama recibida.

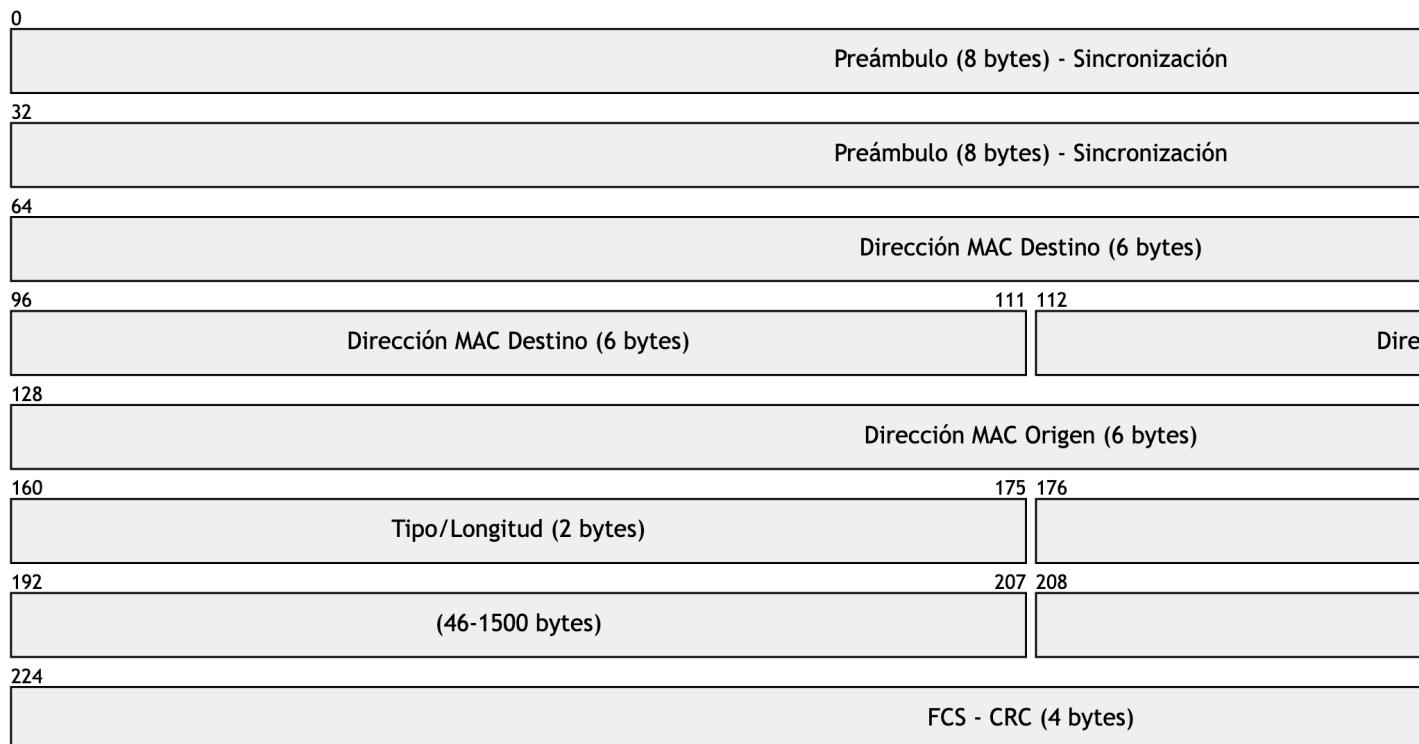


Figura 2.2: Cabeceras de un paquete de Ethernet.

La evolución de Ethernet ha sido extraordinaria, comenzando con 10Base-T que maneja 10 Mbps sobre cable trenzado CAT3/5, evolucionando hasta 10GBase-T con 10 Gbps sobre CAT6A/7. En contextos intensivas, o en nodos centrales, se cuenta con implementaciones de mayor velocidad. Todo esto se ha conseguido a través de diferentes estandares de modelos físicos y la mejora de rendimiento en el hardware. También se ha vuelto posible combinar la transmisión de datos y energía eléctrica por un mismo cable, simplificando los dispositivos de red.

#### Límite de longitud

En los cables de pares trenzados hechos de cobre el límite máximo de trasmisión es de 100 metros. Es decir, que cada 100 metros hay que añadir un dispositivo de red que regenere la señal, como switches o repetidores, para evitar la degradación y pérdida de datos. Esta limitación se debe a la atenuación de la señal y la interferencia electromagnética que se acumulan con la distancia. Para superar esta restricción se utilizan alternativas como fibra óptica (alcanza kilómetros sin regeneración), extensores Ethernet (hasta 300m), o tecnologías inalámbricas. En entornos empresariales, esto determina la ubicación estratégica de closets de telecomunicaciones en la arquitectura de red.

### 2.3.2. Wi-Fi (IEEE 802.11)

El protocolo Wi-Fi maneja la comunicación inalámbrica y debe lidiar con desafíos únicos como la interferencia y la movilidad de dispositivos. A diferencia de las redes cableadas donde el medio físico está claramente definido y controlado, las redes inalámbricas operan en un espectro electromagnético compartido donde múltiples factores pueden afectar la calidad de la transmisión. Esta complejidad ha llevado al desarrollo de sofisticados mecanismos de control y técnicas avanzadas de modulación que permiten comunicaciones confiables incluso en entornos con alta densidad de dispositivos. Uno de los métodos principales es la utilización de CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance).

La trama Wi-Fi (802.11) es considerablemente más compleja que Ethernet debido a los desafíos únicos del medio inalámbrico. El **Frame Control** contiene información crítica sobre el tipo de trama, versión del protocolo, y flags especiales para funciones como gestión de energía y fragmentación. El campo **Duration/ID** implementa el mecanismo de reserva virtual del medio, permitiendo que otros dispositivos sepan cuánto tiempo estará ocupado el canal. La característica más distintiva son las **múltiples direcciones MAC** (hasta 4) que manejan la complejidad de las redes inalámbricas. Address 1 identifica al receptor inmediato, Address 2 al transmisor inmediato, Address 3 proporciona filtrado adicional (a menudo el BSSID del punto de acceso), y Address 4 se utiliza únicamente en sistemas de distribución inalámbrica cuando los access points se comunican entre sí. Los **campos opcionales** reflejan la evolución del estándar: QoS Control permite priorización de tráfico para aplicaciones sensibles al tiempo, HT Control habilita características de alto rendimiento como MIMO y beamforming, y el

**Sequence Control** maneja la ordenación de tramas y detección de duplicados, crítico en un medio donde las transmisiones pueden perderse o duplicarse debido a interferencia.

Al igual que en Ethernet, la evolución de Wi-Fi ha sido enorme. Los primeros estándares, 802.11n (Wi-Fi 4), contaban con velocidades de hasta 600 Mbps, mientras que Wi-Fi 7 (802.11be) promete hasta 46 Gbps. Las mejoras se han enfocado en técnicas para reducir interferencias entre redes y colisiones entre dispositivos, y la inclusión de más bandas:

- 2.4 GHz: mayor alcance y penetración, pero menor velocidad.
- 5 GHz: mayor velocidad pero menor alcance.
- 6 GHz: mejor rendimiento, aunque requiere de hardware específico y tiene el mejor alcance.

### 2.3.3. Point-to-Point Protocol (PPP)

PPP se utiliza para conexiones directas entre dos dispositivos, comúnmente en enlaces seriales y conexiones de acceso telefónico. Este protocolo fue diseñado específicamente para superar las limitaciones de protocolos más antiguos como SLIP (Serial Line Internet Protocol), proporcionando un marco robusto y flexible para comunicaciones punto a punto. Aunque su uso ha disminuido con la proliferación de tecnologías de banda ancha, PPP sigue siendo relevante en conexiones de respaldo, enlaces satelitales, y ciertas implementaciones de VPN donde se requiere control granular sobre la conexión. PPP utiliza un formato de trama mucho más simple que Ethernet o Wi-Fi, reflejando su naturaleza punto a punto donde no hay necesidad de direccionamiento complejo, permitiendo un procesamiento eficiente en enlaces de baja velocidad y dispositivos con recursos limitados.

Las características avanzadas de PPP lo distinguen de protocolos más simples al integrar detección y corrección de errores que garantizan la integridad de los datos transmitidos incluso en enlaces propensos a interferencia, capacidades de autenticación mediante PAP (Password Authentication Protocol) o el más seguro CHAP (Challenge Handshake Authentication Protocol), y configuración automática de direcciones IP que negocia dinámicamente parámetros de red eliminando la necesidad de configuración manual en ambos extremos.

### 2.3.4. Frame Relay

Frame Relay es un protocolo de capa de enlace utilizado en redes WAN que proporciona conexiones virtuales entre sitios remotos. Frame Relay fue desarrollado como una evolución más eficiente de X.25, eliminando muchas de las verificaciones y controles redundantes que hacían lento al protocolo anterior. Aunque ha sido en gran medida reemplazado por tecnologías más modernas como MPLS y VPN sobre Internet, Frame Relay estableció conceptos fundamentales de redes WAN que siguen siendo relevantes en tecnologías contemporáneas.

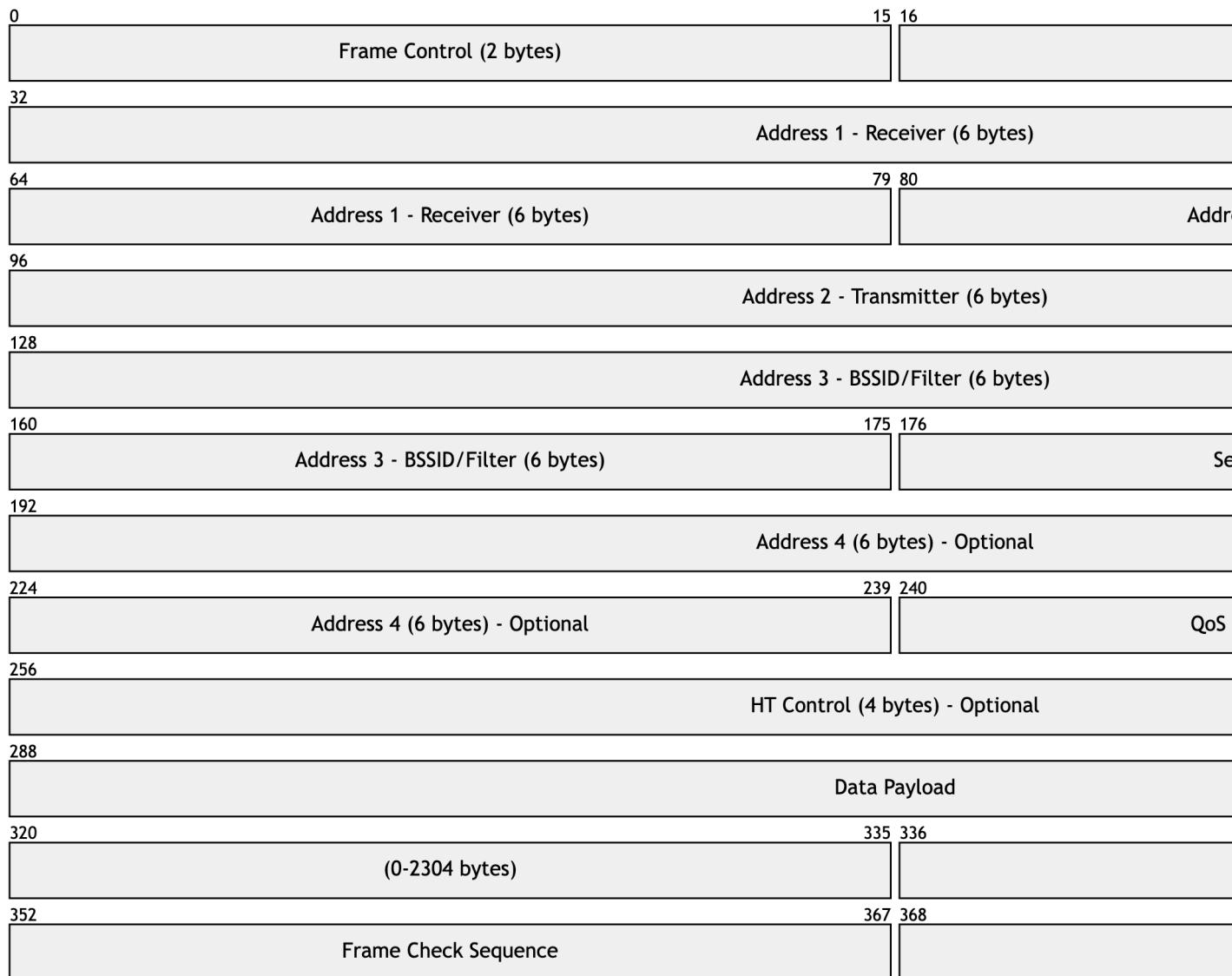


Figura 2.3: Cabeceras de un paquete Wi-Fi

La arquitectura de Frame Relay se basa en conmutación de tramas utilizando identificadores de circuito virtual que permiten múltiples conexiones lógicas sobre una sola interfaz física, simplificando la gestión de conectividad entre múltiples sitios remotos. El protocolo implementa un control de congestión sofisticado.

### **2.3.5. Address Resolution Protocol (ARP)**

ARP es fundamental para la operación de redes IP sobre Ethernet, proporcionando la traducción entre direcciones IP (Capa de Red) y direcciones MAC (Capa de Acceso a la Red). Este protocolo resuelve uno de los problemas más básicos pero críticos en redes: cómo traducir direcciones lógicas que los humanos y aplicaciones entienden fácilmente a direcciones físicas que el hardware de red requiere para la transmisión real.

El proceso ARP opera mediante un mecanismo de solicitud y respuesta que minimiza el tráfico de red mientras proporciona la información necesaria. Cuando un dispositivo necesita comunicarse con otro pero solo conoce su dirección IP, envía un ARP Request como broadcast preguntando “¿Quién tiene la IP X.X.X.X?” a todos los dispositivos del segmento local. El dispositivo que posee esa dirección IP específica responde con un ARP Reply unicast que incluye su dirección MAC, permitiendo al solicitante establecer la asociación necesaria. Para optimizar el rendimiento, estas asociaciones IP-MAC se almacenan en una caché ARP local con temporizadores que eliminan automáticamente entradas obsoletas, evitando repetir el proceso de resolución para comunicaciones frecuentes.

ARP soporta múltiples modalidades de operación que se adaptan a diferentes necesidades de red y escenarios operativos. ARP Estático permite crear entradas manuales permanentes que nunca expiran, útil para dispositivos críticos como gateways y servidores donde se requiere máxima predictibilidad. ARP Dinámico constituye el modo normal de operación donde las entradas se aprenden automáticamente con tiempo de vida configurable, balanceando eficiencia con actualización automática cuando los dispositivos cambian.

## 3 Capa de red

La capa de red es el segundo nivel del modelo de capas TCP/IP y forma el núcleo del sistema de comunicaciones de Internet. Su principal función es proporcionar una comunicación end-to-end entre dispositivos, potencialmente separados por múltiples redes intermedias, independientemente de la tecnología de subyacente. Es decir, la comunicación funciona de igual forma si estamos conectados a través de WiFi, Ethernet o 5G, a pesar de que sean diferentes medios. Esta clara delimitación de capas permite combinar de forma más sencilla diferentes tecnologías y dispositivo hardware.

Como es habitual, vamos a ver un ejemplo simplificado donde un dispositivo quiere mandarle un mensaje a otro dispositivo que no está en la misma red. Este ejemplo simula una situación real como acceder desde casa a un servidor web de Google. Habrá conceptos que no os suenen pero los veremos a lo largo del capítulo. El dispositivo A (tu ordenador en casa), con IP (192.168.1.10) quiere enviarle un mensaje al dispositivo B (servidor web de Google), con IP (142.250.184.3). Durante el ejemplo vamos a realizar una simplificación y utilizaremos siempre la IP del emisor como 192.168.1.10, pero esto no es válido como veremos posteriormente ya que se trata de una IP privada y el Router-A utilizaría NAT. La estructura de la red es la siguiente:

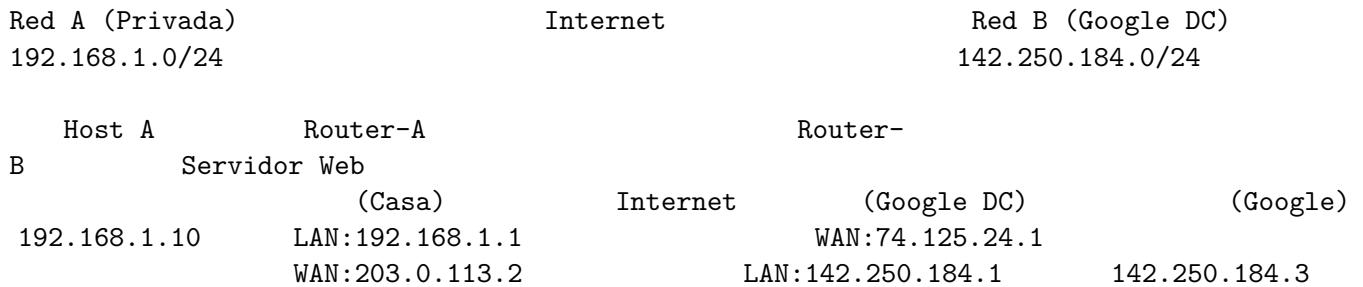


Figura 3.1: Ejemplo simplificado de la estructura de red.

Los pasos de los que constaría este ejemplo simplificado están recogidos en la Figura 3.2 y serían los siguientes:

1. El dispositivo A (192.168.1.10) examina la IP de destino (142.250.184.3). La IP 142.250.184.3 no está en mi red 192.168.1.0/24, por lo tanto enviará el paquete al gateway

(192.168.1.1), es decir, el Router-A. Para ello obtiene la MAC del Router-A y le envía la trama.

2. El Router-A recibe la trama. Ve que la MAC de destino coincide con la suya y extrae el datagrama IP. Lee la IP de destino (142.250.184.3), y como no está en su red local, consulta su tabla de enrutamiento. Determina que debe enviar el paquete a su router del ISP (203.0.113.1). Este router del ISP tendrá en su tabla de enrutamiento una entrada que indica que para llegar a la red 142.250.184.0/24 debe enviar los paquetes al router 74.125.24.1. Router-A actualiza los campos necesarios del datagrama IP y lo encapsula en una nueva trama.
3. El Router-B (74.125.24.1) recibe la trama después de múltiples saltos a través de Internet, extrae el datagrama IP y lee la IP de destino (142.250.184.3). Consulta su tabla de enrutamiento y determina que la red 142.250.184.0/24 está directamente conectada a través de su interfaz 142.250.184.1. Router-B obtiene la MAC del servidor web y le envía el paquete.
4. Finalmente, el servidor web recibe la trama, ve que la MAC de destino es suya, extrae el datagrama IP, comprueba que la IP de destino (142.250.184.3) coincide con la suya, y entrega los datos al protocolo de la capa superior.

Aunque este ejemplo sea una simplificación, nos ayuda a introducir la funcionalidad de la capa de red, en concreto, de los routers y del protocolo IP. Generalmente, entre el Router-A y Router-B habría múltiples routers intermedios, pero proceso seguiría siendo el mismo. En los siguientes apartados profundizaremos en las funcionalidades de la capa de red a través de los routers y el protocolo IP.

### 3.1. Funciones Fundamentales de la Capa de Red

La Capa de Red tiene dos funciones clave: el enrutamiento y el reenvío. El **enrutamiento** representa el proceso global mediante el cual la red determina las rutas óptimas que seguirán los paquetes de datos desde su origen hasta su destino final. Este proceso considera toda la topología de la red y puede tomar desde segundos hasta minutos para converger completamente. Los algoritmos de enrutamiento más comunes son RIP, OSPF y BGP.

En contraste, el **reenvío** constituye un proceso local y extremadamente rápido que se encarga de mover los paquetes desde el puerto de entrada hasta el puerto de salida específico dentro del mismo router. Esta operación debe completarse en microsegundos para mantener el rendimiento de la red, por lo que se implementa directamente en hardware. El proceso se basa exclusivamente en la dirección IP de destino y utiliza únicamente la tabla de reenvío local del router para tomar decisiones inmediatas.

La interacción entre ambos procesos forma un sistema integrado donde los algoritmos de enrutamiento como RIP, OSPF y BGP generan la tabla de enrutamiento con rutas completas,

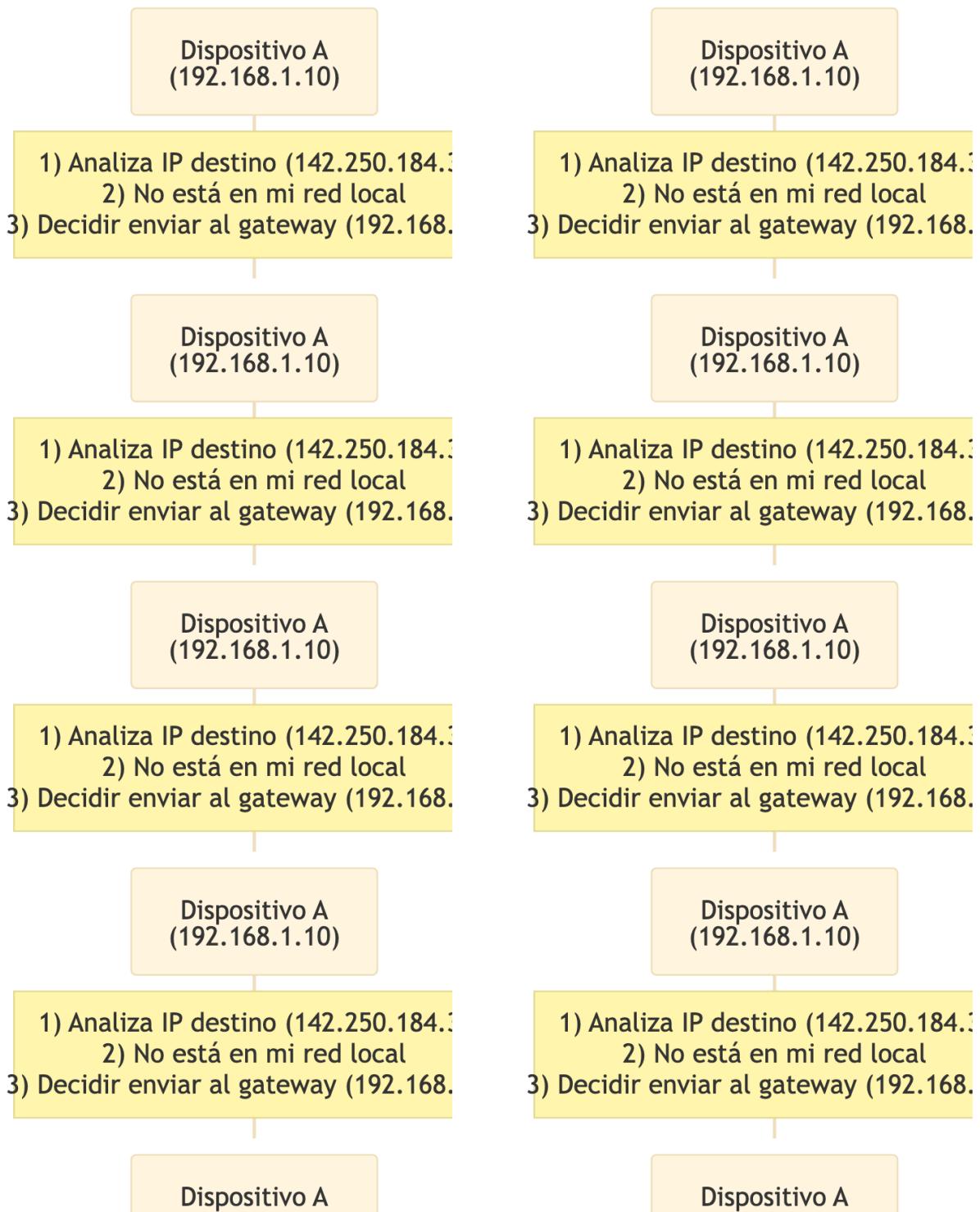


Figura 3.2: Ejemplo de envío de datagrama IP entre dos ordenadores en diferentes redes.

la cual se traduce en una tabla de reenvío optimizada que contiene únicamente la información del siguiente salto (next-hop). Esta tabla de reenvío es la que finalmente permite tomar las decisiones de reenvío paquete por paquete de manera eficiente, creando un flujo continuo desde la planificación estratégica de rutas hasta la ejecución táctica del movimiento de datos.

Las responsabilidades de la capa de red varían según el tipo de dispositivo y su posición en el flujo de comunicación. En el host emisor, la capa de red recibe segmentos de TCP o UDP y los encapsula en datagramas IP añadiendo las cabeceras correspondientes. Durante este proceso, debe fragmentar los datagramas si exceden el MTU del enlace de salida y determinar si el destino es local (dentro de la misma red) o remoto para enviarlo. En el extremo opuesto, el host receptor debe reensamblar los fragmentos cuando sea necesario, verificar la integridad de los datos mediante el checksum de cabecera, extraer los segmentos y entregarlos a la capa de transporte apropiada, además de procesar las opciones de cabecera IP cuando estén presentes.

Los routers intermedios desempeñan un papel diferente pero crucial en este ecosistema. Su función principal consiste en examinar los campos de la cabecera IP, especialmente la dirección de destino, consultar sus tablas de enrutamiento para determinar el siguiente salto apropiado, y reenviar los paquetes por la interfaz de salida correspondiente.

### 3.2. Modelos de servicio

Existen dos paradigmas fundamentales para implementar servicios de capa de red, cada uno con filosofías y mecanismos completamente diferentes. La elección entre estos modelos determina aspectos cruciales como performance, confiabilidad, complejidad y escalabilidad de la red.

**Las redes de circuitos virtuales (VC)** emulan el comportamiento de los circuitos telefónicos tradicionales estableciendo “caminos virtuales” dedicados entre origen y destino. Su funcionamiento se desarrolla en tres fases claramente definidas: primero, el establecimiento de conexión mediante el envío de un mensaje SETUP desde el host origen, donde cada router intermedio reserva recursos como ancho de banda y buffers, crea una entrada en su tabla VC con un identificador único local, y reenvía la solicitud hasta que el host destino confirma con un mensaje ACK. Durante la fase de transferencia de datos, los paquetes solo necesitan llevar el VC ID asignado en lugar de la dirección de destino completa, permitiendo un reenvío rápido mediante consulta a la tabla VC, garantizando calidad de servicio (QoS) y manteniendo una ruta fija para todos los paquetes del flujo. Finalmente, la terminación se realiza mediante un mensaje TEARDOWN que libera los recursos previamente reservados y elimina las entradas de las tablas VC.

Esta arquitectura ofrece ventajas significativas como QoS predecible con garantías de rendimiento, overhead reducido en las cabeceras al usar solo el VC ID, control de flujo extremo a extremo y orden garantizado de los paquetes. Sin embargo, presenta desventajas importantes incluyendo la complejidad en el establecimiento y mantenimiento de conexiones, la necesidad de mantener estado por cada conexión en todos los routers, rigidez ante cambios en la topología

de red y overhead adicional por la señalización requerida. Tecnologías como ATM, Frame Relay, X.25 y MPLS implementan este modelo de circuitos virtuales para aplicaciones que requieren garantías específicas de rendimiento.

**Las redes de datagramas** adoptan un enfoque completamente diferente al tratar cada paquete de manera independiente sin establecer conexiones previas entre origen y destino. Este modelo se caracteriza por la ausencia de estado de conexión en los routers, eliminando la necesidad de un proceso de setup inicial, y basa el reenvío en la dirección de destino completa contenida en cada paquete. Cada router procesa los paquetes independientemente. Como resultado, diferentes paquetes del mismo flujo puedan seguir rutas distintas a través de la red.

El modelo de datagramas presenta ventajas sustanciales en términos de simplicidad de diseño e implementación, robustez excepcional ante fallos de red ya que no depende de estados de conexión preestablecidos, flexibilidad para implementar balanceo dinámico de carga, escalabilidad superior al no requerir mantener estado por cada flujo, y adaptabilidad inmediata a cambios en la topología de red. Estas características hacen que las redes de datagramas sean especialmente adecuadas para entornos dinámicos y de gran escala como Internet.

No obstante, el modelo de datagramas también presenta limitaciones significativas que incluyen la ausencia de garantías de calidad de servicio (QoS), la posibilidad de que los paquetes lleguen fuera de orden al destino debido a las diferentes rutas que pueden tomar, el overhead adicional generado por incluir la dirección de destino completa en cada paquete, y la prestación únicamente de un servicio de mejor esfuerzo (best-effort) sin compromisos específicos de rendimiento. A pesar de estas limitaciones, el modelo de datagramas se ha convertido en el fundamento de Internet debido a su simplicidad, robustez y capacidad de adaptación a las condiciones cambiantes de la red.

Resumen comparativo:

Aspecto	Circuitos Virtuales	Datagramas
<b>Establecimiento</b>	Requerido	No requerido
<b>Estado en routers</b>	Sí, por conexión	No
<b>Direccionamiento</b>	VC ID	Dirección IP completa
<b>Enrutamiento</b>	Ruta fija	Ruta por paquete
<b>QoS</b>	Garantías posibles	Best effort
<b>Recuperación fallos</b>	Difícil	Automática
<b>Escalabilidad</b>	Limitada	Alta
<b>Overhead</b>	Setup/teardown	Por paquete

### 3.3. Dispositivos físicos de la Capa de Red

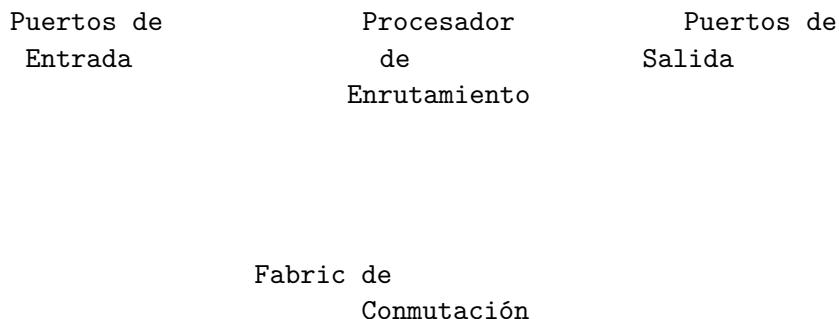
Los dispositivos de capa de red son los componentes hardware que hacen posible la interconexión de redes y la implementación de las funciones de enrutamiento y reenvío. Estos dispositivos

varían considerablemente en complejidad, desde simples switches Layer 3 hasta routers core de alta capacidad.

### 3.3.1. Routers (Enrutadores)

Los routers constituyen la columna vertebral de Internet y las redes empresariales modernas. Su función principal es interconectar diferentes redes y determinar la ruta óptima para el reenvío de paquetes de datos. A diferencia de los switches que operan en la Capa de Acceso a la Red, los routers trabajan en la Capa de Red, tomando decisiones basadas en direcciones IP y manteniendo una visión global de la topología de red.

La arquitectura básica consta de cuatro componentes principales:



En los routers diferenciamos dos planos claramente separados:

- **plano de control:** ejecuta el proceso de enrutamiento mediante software especializado y generan tablas de enrutamiento que contienen rutas completas hacia todos los destinos conocidos.
- **plano de datos:** ejecuta el proceso de reenvío mediante hardware especializado para máxima eficiencia. Utiliza la tabla de enrutamiento generada en el plano de control.

Los puertos de entrada constituyen las puertas de recepción del router y realizan tres funciones críticas organizadas en secuencia. La terminación física proporciona la interfaz con medios de transmisión como cables de cobre o fibra óptica, convirtiendo las señales eléctricas u ópticas en datos digitales. El procesamiento de la capa de enlace maneja protocolos específicos como Ethernet, PPP o Frame Relay, extrayendo el datagrama IP de la trama correspondiente. La función de búsqueda IP consulta la tabla de reenvío usando el algoritmo de coincidencia de

prefijo más largo para determinar hacia dónde dirigir cada paquete. Esta función debe ejecutarse a la velocidad del enlace para evitar crear cuellos de botella en el sistema.

Los puertos de salida gestionan el tráfico que abandona el router mediante un proceso inverso al de entrada. La bufferización y scheduling implementa sistemas de colas sofisticados que aplican políticas de calidad de servicio, decidiendo qué paquetes enviar primero según sus prioridades. El procesamiento de la capa de enlace encapsula el datagrama IP en la trama apropiada para el protocolo del enlace de salida. Finalmente, la terminación física convierte los datos digitales en señales eléctricas u ópticas para su transmisión.

El procesador de enrutamiento funciona como el cerebro del sistema, ejecutando los protocolos de enrutamiento que intercambian información con otros routers para mantener actualizado el conocimiento de la topología de red. También gestiona funciones administrativas como SNMP para monitoreo remoto, procesamiento ICMP para herramientas de diagnóstico como ping y traceroute, y la computación de las tablas de reenvío optimizadas a partir de las tablas de enrutamiento.

Por último, NAT es un protocolo que opera entre ambas capas (lo veremos después). Al principio operaba sólo en el plano de control, tomando un tiempo significativo. En la actualidad, opera en el plano de control para manejar las sesiones y el resto en hardware especializado en el plano de datos.

### 3.3.1.1. Proceso de Reenvío de Paquetes

El proceso de reenvío sigue una secuencia precisa y optimizada que se ejecuta para cada paquete:

1. **Recepción y procesamiento inicial:** El paquete llega al puerto de entrada desde el enlace físico, se procesa la cabecera de la capa de enlace correspondiente y se extrae el datagrama IP.
2. **Verificación de integridad:** Se verifica el checksum de la cabecera IP para detectar posibles errores de transmisión y se comprueba que el valor TTL sea mayor que cero.
3. **Extracción de información de destino:** Se extrae la dirección IP de destino de la cabecera del datagrama para utilizarla en la decisión de reenvío.
4. **Consulta de tabla de reenvío:** Se aplica el algoritmo de coincidencia de prefijo más largo en la tabla de reenvío para determinar la interfaz de salida apropiada y obtener la dirección del siguiente salto.
5. **Modificación del paquete:** Se decrementa el campo TTL en una unidad y se recalcula el checksum de la cabecera IP para mantener la integridad de los datos. Si el TTL llega a cero después del decremento, el router descarta el paquete y envía un mensaje ICMP “Time Exceeded” al host origen, evitando así loops infinitos en la red.

6. **Resolución de direcciones:** Si es necesario, se resuelve la dirección MAC del dispositivo del siguiente salto mediante el protocolo ARP.
7. **Encapsulación y envío:** Se encapsula el datagrama IP en una nueva trama según el protocolo de la capa de enlace del puerto de salida y se transmite por la interfaz física correspondiente.

### 3.3.2. Switches de Capa 3

A medida que las redes locales crecieron en complejidad, surgió la necesidad de dispositivos que combinaran la velocidad del switching con las capacidades del routing. Los switches Layer 3 llenan este nicho específico. La principal diferencia es la implementación a nivel de hardware del procesamiento, haciéndolo mucho más rápido. A modo de comparativa tenéis la siguiente tabla:

Aspecto	Router Tradicional	Switch L3
<b>Reenvío</b>	Software/ASIC	Hardware puro
<b>Latencia</b>	Microsegundos	Nanosegundos
<b>Throughput</b>	Limitado por CPU	Wire-speed
<b>Costo</b>	Mayor	Menor
<b>Flexibilidad</b>	Alta	Limitada

## 3.4. Protocolos

### 3.4.1. Protocolo IP

IP es el protocolo principal de la capa de red en la arquitectura TCP/IP. Define la estructura de datagramas, direccionamiento y mecanismos básicos de entrega. Las características principales de IP son:

- **Sin conexión:** No requiere establecimiento previo.
- **No confiable:** No garantiza entrega, orden, o integridad.
- **Best effort:** Hace el “mejor esfuerzo” por entregar paquetes.
- **Independiente del medio:** Funciona sobre cualquier tecnología de enlace.

Dentro de IP hay dos versiones. IPv4 diseñado en los años 70 y IPv6, como evolución de IPv4 enfocado a solventar las limitaciones de IPv4, en especial el número de IPs disponibles. Empezaremos por IPv4.

El datagrama IPv4 es la unidad básica de información que viaja por Internet (Ver estructura en Figura 3.3). Utiliza una cabecera de longitud variable (mínimo 20 bytes) que contiene la

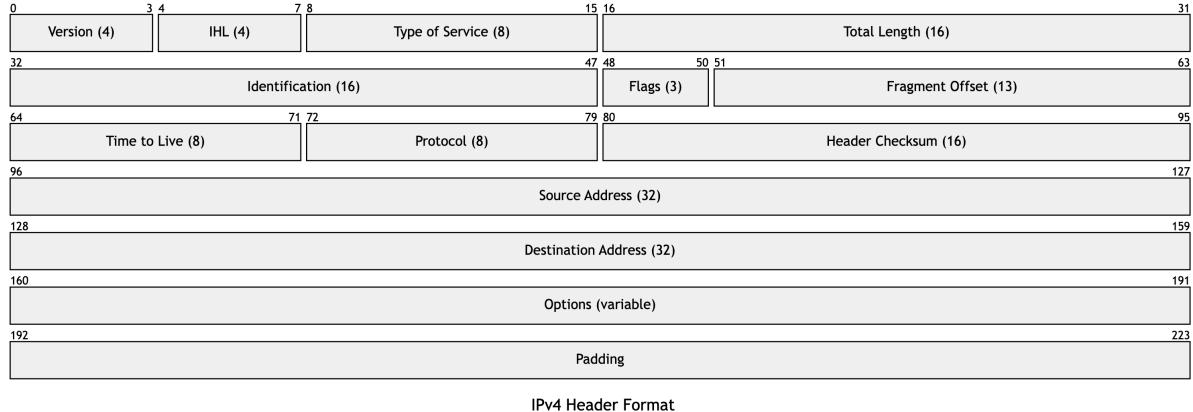


Figura 3.3: Formato de cabeceras de IPv4.

información esencial para el enrutamiento y entrega de paquetes a través de Internet. Los campos más críticos incluyen las direcciones IP de origen y destino que determinan los puntos de comunicación, el campo TTL que previene loops infinitos al decrementarse en cada router, el campo Protocol que identifica el protocolo de capa superior (TCP, UDP, ICMP), y los campos de fragmentación (Identification, Flags, Fragment Offset) que permiten dividir y reensamblar datagramas que exceden el MTU del enlace. El checksum protege únicamente la cabecera, delegando la protección de los datos a las capas superiores, mientras que el campo Total Length especifica el tamaño completo del datagrama para su procesamiento correcto.

El sistema de direccionamiento IPv4, llamadas IP, es un identificador único de un dispositivo dentro de una red. En IPv4 tienen un formato de 32 bits que se organiza en 4 octetos separados por puntos. Por ejemplo, 192.168.1.1 o 10.0.1.50. Debido a la longitud de 32 bits, el número de direcciones IP posibles son  $2^{32}$ , aproximadamente 4.3 miles de millones. Estas direcciones se organizan en dos partes, la parte de red y la parte de host, además tenemos la máscara de red que nos ayuda a distinguir ambas partes. Por ejemplo, 192.168.1.1 con máscara de red 255.255.255.0 o 10.0.1.50 con máscara de red 255.255.0.0, siendo la parte azul la parte de red y la roja la parte del host. Para obtener la dirección de red utilizamos el operador binario AND:  $192.168.1.1 \& 255.255.255.0 = 192.168.1.0$ . En CIDR, que veremos más adelante, esta máscara 255.255.255.0 se representa como /24.

Esta división entre parte de red y parte de host permite representar jerárquicamente la estructura de direccionamiento, como se muestra en la Figura 3.4. Los routers pueden tomar decisiones de reenvío basándose únicamente en la parte de red de la dirección destino, consultando sus tablas locales para determinar la interfaz de salida. Gracias a esta organización, es posible la agregación de rutas, donde varias redes pequeñas se resumen en una sola entrada de mayor alcance. Por ejemplo, dos subredes /28 contiguas (192.168.1.0/28 y 192.168.1.16/28) pueden representarse como un único bloque /27 (192.168.1.0/27), reduciendo de dos entradas a una. Este mecanismo permite que los routers mantengan información consolidada sobre redes remotas sin necesidad

de conocer cada host o subred en detalle, lo que disminuye drásticamente el tamaño de las tablas de reenvío y hace escalable la infraestructura global de Internet.

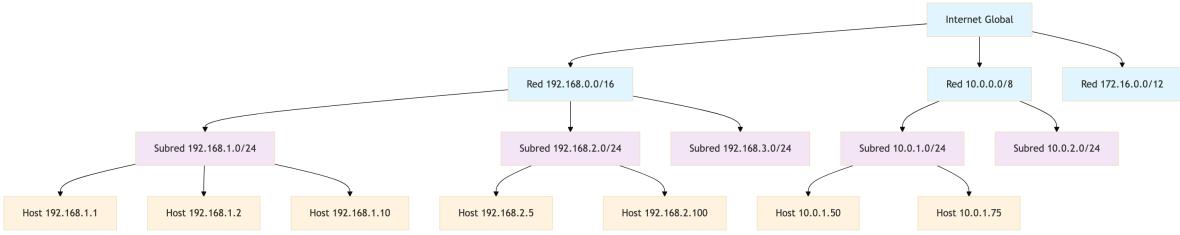


Figura 3.4: Ejemplo de estructuras de subredes.

La estructura de direccionamiento IPv4, que permite distinguir entre red y host para generar una arquitectura jerárquica de redes, inicialmente utilizaba un sistema de clases. En este sistema de clases, las direcciones IPv4 se categorizaban en tres grupos principales según los bits iniciales del primer octeto, determinando la división entre bits de red y host. La Clase A comenzaba con bit 0, la Clase B con bits “10”, y la Clase C con bits “110”, creando saltos enormes entre las capacidades de cada categoría que generaban ineficiencias significativas en la asignación. La estructura de direccionamiento IPv4, que permite distinguir entre red y host para generar una arquitectura jerárquica de redes, inicialmente utilizaba un sistema de clases. En este sistema de clases, las direcciones IPv4 se categorizaban en tres grupos principales según los bits iniciales del primer octeto, determinando la división entre bits de red y host. La Clase A comenzaba con bit 0, la Clase B con bits “10”, y la Clase C con bits “110”, creando saltos enormes entre las capacidades de cada categoría que generaban ineficiencias significativas en la asignación.

Clase	Rango de Direcciones	Primer Bit(s)	Bits de Red	Bits de Host	Redes Disponibles	Hosts por Red	Uso Típico
<b>A</b>	0.0.0.0 - 127.255.255.255	0	7	24	126 <sup>1</sup>	16,777,214	ISPs, gobiernos, organizaciones masivas
<b>B</b>	128.0.0.0 - 191.255.255.255	10	14	16	16,384	65,534	Universidades, empresas medianas

<sup>1</sup>Las direcciones 0.0.0.0/8 y 127.0.0.0/8 están reservadas para funciones especiales.

Clase	Rango de Direcciones	Primer Bit(s)	Bits de Red	Bits de Host	Redes Disponibles	Hosts por Red	Uso Típico
C	192.0.0.0 - 223.255.255.255	110	21	8	2,097,152	254	Empresas pequeñas, oficinas locales

Sin embargo, la rigidez del sistema de clases generaba problemas críticos. Una organización con 1,000 hosts enfrentaba un dilema: elegir una red Clase B desperdiando 64,534 direcciones (99.5 % de ineficiencia) o gestionar múltiples redes Clase C con mayor complejidad administrativa. Esta inflexibilidad aceleró el agotamiento del espacio IPv4 y motivó el desarrollo de alternativas más eficientes.

Para solventar este problema se introdujo CIDR. La innovación fundamental consistió en la notación /x que indica exactamente cuántos bits destinan a la parte de red. Por ejemplo, 192.168.1.0/24 significa que los primeros 24 bits identifican la red, dejando 8 bits para hosts (254 hosts utilizables). CIDR permite asignar direcciones en bloques de cualquier tamaño potencia de 2, eliminando el desperdicio masivo del sistema anterior. Una organización que necesite 500 hosts puede recibir un /23 (510 hosts) en lugar de desperdiciar una Clase B completa. Esta flexibilidad aumentó la utilización del espacio IPv4 del 20-30 % tradicional al 95-98 % actual y la simplificación de las tablas de enrutamiento globales mediante la agregación de rutas.

Para funcionar, CIDR requiere el algoritmo de longest prefix matching para búsquedas en tablas de enrutamiento. Cuando un router recibe un paquete, evalúa todas las rutas que coinciden con la dirección destino y selecciona aquella con el prefijo más específico. En una tabla con rutas 192.168.0.0/16, 192.168.1.0/24 y 192.168.1.128/25, el destino 192.168.1.200 coincide con las dos primeras pero selecciona 192.168.1.0/24 por tener el prefijo más largo (24 bits vs 16 bits). Este mecanismo garantiza que el tráfico tome siempre la ruta más específica disponible.

Independientemente del sistema de direccionamiento utilizado (clases o CIDR), IPv4 mantiene direcciones especiales con propósitos específicos:

- 0.0.0.0/32: This host on this network. Referencia un host sin IP configurada. Se utiliza en el proceso de configuración (DHCP).
- 127.0.0.0/8: Loopback. Los paquetes no salen del host local y se utiliza para servicios y pruebas. Un ejemplo común es localhost, con IP 127.0.0.1.
- 255.255.255.255/32: Limited broadcast. Broadcast a todos los hosts en red local. No atraviesa routers.
- x.x.x.0: Dirección de red. Todos los bits del host a 0 (con la máscara de red). Identifica a la red misma.

- x.x.x.255: Directed broadcast. Todos los bits de host a 1. Broadcast dirigido a una red específica.

En el sistema de clases, las direcciones de red y broadcast seguían patrones fijos según la clase, pero con CIDR se adaptan dinámicamente a la máscara de subred específica utilizada.

Ambos sistemas establecen una serie de rangos, determinadas privadas, que son exclusivas para redes internas. Estas direcciones no son enruteables en Internet público, ya que los routers globales están configurados para descartarlas, evitando conflictos de direccionamiento. La principal ventaja radica en que múltiples organizaciones pueden reutilizar los mismos rangos internamente sin interferir entre sí, conservando el escaso espacio IPv4 público. Para acceder a Internet, estas redes requieren NAT, que traduce direcciones privadas a públicas. Los rangos delimitados son: 10.0.0.0/8 (16.7 millones de hosts, para grandes organizaciones), 172.16.0.0/12 (1 millón de hosts, para empresas medianas) y 192.168.0.0/16 (65,000 hosts, para hogares y oficinas pequeñas).

Por último, en el protocolo IP hay un tamaño máximo para el datagrama. Este tamaño se conoce como MTU (del inglés, Maximum Transmission Unit), y puede variar dependiendo de la tecnología subyacente, por ejemplo, en Ethernet es 1500 bytes y en Token Ring es 4464 bytes. Cuando el tamaño del datagrama es superior al MTU, el datagrama se fragmenta en trozos más pequeños y se desfragmentará posteriormente en el destino. Una consideración importante es que el protocolo IP sí mantiene el orden de la información del datagrama. Es decir, si yo envío un datagrama que se tiene que fragmentar, IP garantiza que al desfragmentarlo la integridad de los datos estará preservada. Cuando decimos que no garantiza el orden es que si primero envío el datagrama A y después otro datagrama B (independientes), puede que la aplicación reciba primero el datagrama B y después el A, y no tendrá forma de saber si uno va antes que el otro.

### 3.4.1.1. IPv6

IPv6 surge como respuesta a las limitaciones críticas de IPv4, principalmente el agotamiento de su espacio de direcciones de 32 bits que solo proporciona  $4.3 \times 10^9$  direcciones únicas. Además, IPv4 presenta problemas de fragmentación ineficiente que requiere procesamiento en routers intermedios, configuración manual compleja sin capacidades de autoconfiguración, implementación de seguridad como complemento opcional (IPSec), y limitaciones en calidad de servicio con campos TOS poco efectivos. Estos desafíos hacen insostenible IPv4 para el crecimiento exponencial de dispositivos conectados a Internet.

IPv6 revoluciona el protocolo con un espacio de direcciones masivo de 128 bits que proporciona  $2^{128} = 3.4 \times 10^{38}$  direcciones, utilizando notación hexadecimal con reglas de compresión para simplificar su representación. La cabecera, ver Figura 3.5, se simplifica a un formato fijo de 40 bytes eliminando el checksum para reducir el procesamiento en routers, e integra características avanzadas como autoconfiguración SLAAC, seguridad IPSec obligatoria, y mejor calidad de servicio mediante campos Traffic Class y Flow Label. Debido a la cantidad de

dispositivos en la red, la migración de IPv4 a IPv6 se realiza de forma gradual mediante estrategias que permiten la interoperabilidad entre ambos protocolos.

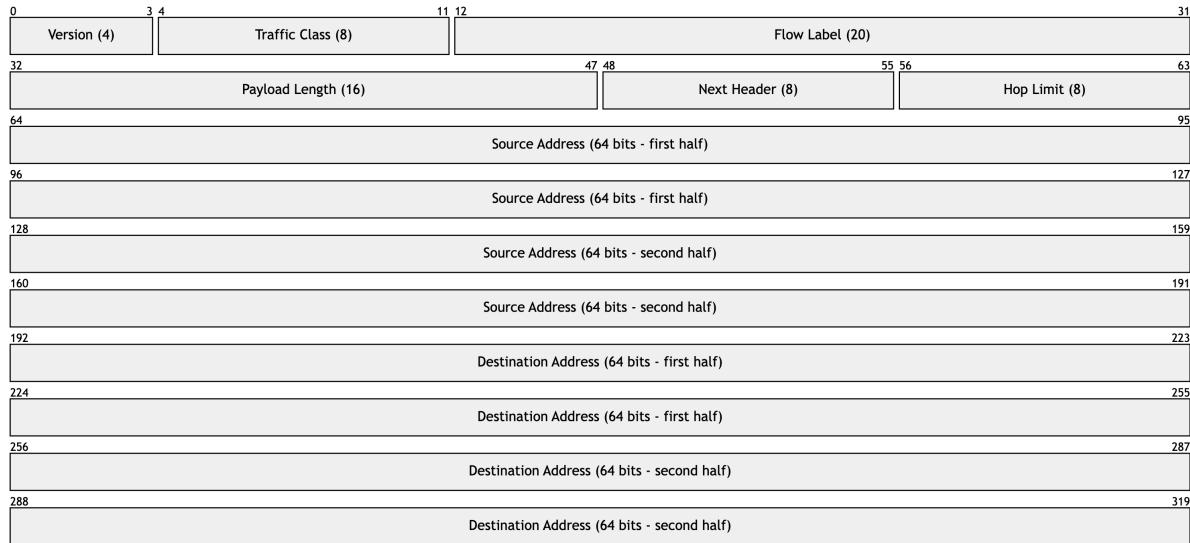


Figura 3.5: Formato de cabeceras de IPv6.

### 3.4.2. Protocolo ICMP (Internet Control Message Protocol)

ICMP es un protocolo complementario a IP que proporciona mecanismos de control, diagnóstico y reporte de errores en redes. Utiliza IP para su transporte (protocolo número 1) pero opera como herramienta de gestión de red. Es no orientado a conexión, no garantiza entrega, y está implementado obligatoriamente en todos los dispositivos IP. Su formato básico incluye campos Type, Code, Checksum y datos adicionales según el tipo de mensaje.

Los mensajes ICMP se clasifican en dos categorías principales: mensajes de error y mensajes de consulta. Los mensajes de error incluyen “Destination Unreachable” (Type 3) que indica problemas de alcance como red, host o puerto inaccesible; “Time Exceeded” (Type 11) usado cuando el TTL expira en tránsito; “Parameter Problem” (Type 12) para errores de configuración; o “Packet Too Big” en el mecanismo de MTU Discovery de IPv6. Los mensajes de consulta incluyen “Echo Request/Reply” (Type 8/0) utilizados por ping para verificar conectividad y medir latencia, y “Timestamp Request/Reply” (Type 13/14) para sincronización temporal.

#### Ping - Verificación de conectividad:

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=55 time=15.1 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=55 time=14.9 ms
```

Este ejemplo muestra ping enviando Echo Request (Type 8) al servidor DNS de Google y recibiendo Echo Reply (Type 0) exitosamente. Las respuestas muestran latencias de ~15ms, TTL=55, y confirman conectividad funcional.

#### Traceroute - Descubrimiento de ruta:

```
$ traceroute google.com
 1  192.168.1.1 (192.168.1.1)  3.414 ms  3.863 ms  1.752 ms
 2  100.70.0.1 (100.70.0.1)  5.245 ms  4.996 ms  4.405 ms
 3  10.14.0.53 (10.14.0.53)  7.091 ms  4.812 ms  4.892 ms
 4  10.14.246.6 (10.14.246.6)  4.209 ms  4.406 ms  4.230 ms
 5  * * *
 6  72.14.195.182 (72.14.195.182)  4.665 ms
    72.14.194.132 (72.14.194.132)  3.950 ms
    72.14.195.182 (72.14.195.182)  4.968 ms
 7  74.125.245.171 (74.125.245.171)  5.109 ms  5.751 ms  5.791 ms
 8  142.251.49.55 (142.251.49.55)  4.185 ms
    142.251.49.53 (142.251.49.53)  5.317 ms
    142.251.49.55 (142.251.49.55)  3.791 ms
 9  mad41s11-in-f14.1e100.net (142.250.185.14)  4.722 ms  6.253 ms  4.893 ms
```

Este ejemplo revela la ruta completa hacia google.com incrementando TTL progresivamente. Cada router responde “Time Exceeded” (Type 11, Code 0) mostrando su IP. El salto 5 muestra timeouts (\*), el salto 6 y 8 muestra平衡adores de carga, y finalmente alcanza el servidor de Google en el salto 9.

#### 3.4.3. NAT (Network Address Translation)

NAT surgió como una solución al problema del agotamiento de direcciones IPv4, permitiendo que múltiples dispositivos en una red privada comparten una sola dirección IP pública. Esta técnica se basa en el uso de direcciones privadas que pueden reutilizarse sin conflictos. El dispositivo NAT, típicamente integrado en routers de acceso doméstico o empresarial, actúa como intermediario entre la red interna y externa, traduciendo direcciones y puertos en tiempo real.

El funcionamiento de NAT se basa en mantener una tabla de traducción que mapea combinaciones de dirección IP privada y puerto interno con la dirección IP pública y un puerto externo único. Cuando un dispositivo interno inicia una conexión hacia Internet, el router NAT reemplaza la dirección IP de origen privada y el puerto por su dirección IP pública y un puerto disponible de su pool, registrando esta asociación en su tabla. Cuando llega la respuesta desde Internet, el router consulta su tabla de traducción para determinar a qué dispositivo interno debe entregar el paquete, revirtiendo la traducción antes de reenviarlo a la red local.

En la Figura 3.6 podemos ver dos ejemplos de NAT. El Host A envía un paquete desde 192.168.1.10:12345 hacia 8.8.8.8:80. El router NAT lo intercepta, reemplaza el origen por 203.0.113.100:5001 y crea una entrada en su tabla: 192.168.1.10:12345 → 5001. Cuando el servidor responde a 203.0.113.100:5001, el router consulta su tabla NAT, encuentra la correspondencia y reenvía el paquete a 192.168.1.10:12345. El proceso en el Host B sería idéntico, y gracias a NAT habríamos podido comunicarnos con dos dispositivos a través de una única IP.

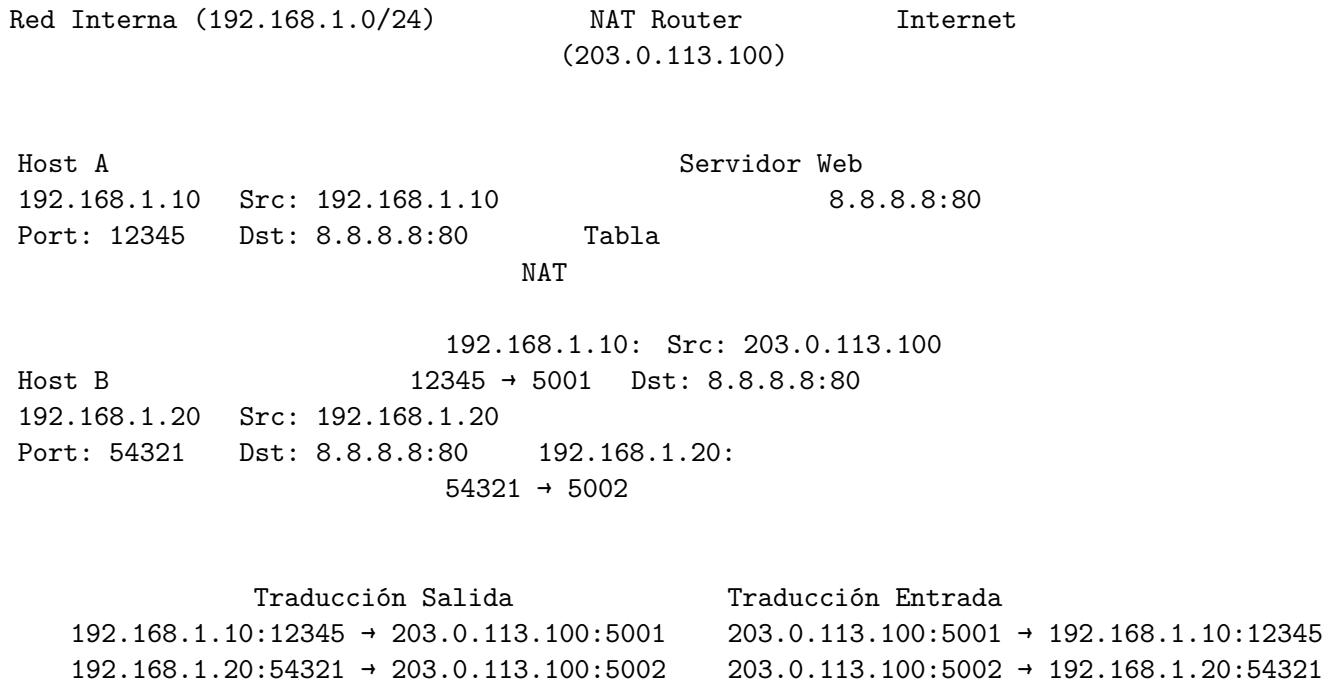


Figura 3.6: Ejemplo de NAT con dos Host que se comunican con un servidor web utilizando una única IP pública.

Sin embargo, NAT presenta limitaciones significativas como la imposibilidad de establecer conexiones entrantes sin configuración manual de port forwarding, complicaciones con aplicaciones que embebén direcciones IP en sus datos (como algunos protocolos VoIP), y la pérdida del principio end-to-end de Internet. A pesar de estas limitaciones, NAT se ha convertido en ubicuo en redes domésticas y empresariales, siendo una pieza fundamental que ha permitido que Internet continúe funcionando mientras se desarrolla la transición hacia IPv6.

La limitación de conexiones entrantes impide que otros dispositivos sean capaces de conectarse a nosotros directamente. Esto impide, por ejemplo, que dos personas puedan conectarse entre sí desde sus casas. Por otra parte, también hace más seguro estar conectado a la red. En determinados casos, conectarse entre sí puede mejorar la experiencia, mejorar la privacidad, o reducir la necesidad de servidores intermedios y sus consecuentes recursos. Para ello, se pueden utilizar diferentes técnicas que permiten saltarse las limitaciones del NAT:

- Hole punching: Técnica donde ambos dispositivos intentan conectarse simultáneamente al otro a través de sus respectivos NATs. El NAT crea temporalmente “agujeros” en su tabla de traducción cuando detecta tráfico saliente, permitiendo que la respuesta del otro extremo pase. Funciona peor (es más difícil) con NAT simétrico y requiere coordinación temporal precisa.
- STUN (Session Traversal Utilities for NAT): Protocolo que permite a un dispositivo descubrir su dirección IP pública y el tipo de NAT que tiene. Un servidor STUN externo ayuda al cliente a determinar cómo el NAT modifica sus paquetes, información crucial para establecer conexiones directas. Es especialmente útil para aplicaciones de tiempo real como VoIP.
- TURN (Traversal Using Relays around NAT): Cuando el hole punching falla, TURN proporciona un servidor relay que actúa como intermediario. Aunque no elimina completamente la necesidad de servidores, centraliza el tráfico en un punto controlado. Es más confiable pero consume más ancho de banda y recursos del servidor.
- UPnP (Universal Plug and Play): Permite que las aplicaciones configuren automáticamente el router para abrir puertos específicos. El dispositivo solicita al router que cree reglas de port forwarding temporales o permanentes. Es conveniente pero requiere que el router soporte UPnP y puede presentar riesgos de seguridad si no se gestiona adecuadamente.

# 4 Capa de transporte

La capa de transporte proporciona comunicación lógica entre procesos de aplicación que se ejecutan en diferentes hosts. Los protocolos de transporte se ejecutan en los hosts finales, no en el núcleo de la red. Los protocolos más comunes son UDP y TCP, que representan los dos lados del espectro en cuanto a funcionalidades. UDP contiene lo mínimo para ser un protocolo de comunicación en la capa de transporte y TCP es un protocolo mucho más complejo pero con más garantías. La elección entre uno y otro dependerá del dominio y la aplicación.

Primero, vamos a ver un ejemplo simplificado donde un Cliente A le manda 5 paquetes a un Servidor B. Entre medias, asumimos que hay una red, Internet, donde no profundizaremos por simplicidad, pero sería como en el Capítulo 3. Podéis ver un ejemplo del escenario en Figura 4.1. En este escenario, un proceso Cliente A le envía 5 paquetes a un proceso del Servidor B. El Servicio B está referenciado a través de la IP (8.8.8.8), y dentro de B podemos identificar el proceso a través del puerto, en este caso, 80. El Cliente A envía un total de 5 paquetes, llegando al servidor, en el siguiente orden: 1, 2, 4, 3. Aquí pasan varias cosas. Lo primero, en paquete 5 no ha llegado, se “perdió” en Internet. Aproximadamente el 1 % de los paquetes se pierden en condiciones normales. En UDP el paquete se perdería, y no nos enteraríamos. En TCP, el proceso se reintentaría. La segunda cosa que os puede llamar la atención es que el paquete 4 llega antes que el 3. Esto puede ocurrir también, ya que los paquetes pueden tomar diferentes caminos. En UDP no tenemos información para corregir el orden, así que se entregaría primero el 4 y después el 3. En cambio, TCP cuenta con mecanismos para corregir el orden.

En los siguientes apartados veremos las funcionalidades de la Capa de Transporte, y profundizaremos en los protocolos TCP y UDP, también veremos una pequeña comparativa de juegos utilizando TCP y UDP.

## 4.1. Funciones principales

Las funciones principales de la capa de transporte son dividir los mensajes en el emisor en segmentos y pasarlo a la capa de red, y posteriormente en el receptor recomponer los segmentos en mensajes y pasarlo a la capa de aplicación. La interfaz entre la capa de transporte y la capa de aplicación se llama sockets y la veremos en detalle en el capítulo de la Capa de Aplicación. Por ahora, sólo es necesario tener en cuenta que a través de los sockets podemos enviar y recibir información. Es la forma que tenemos que utilizar la Capa de Transporte desde la Capa de Aplicación.

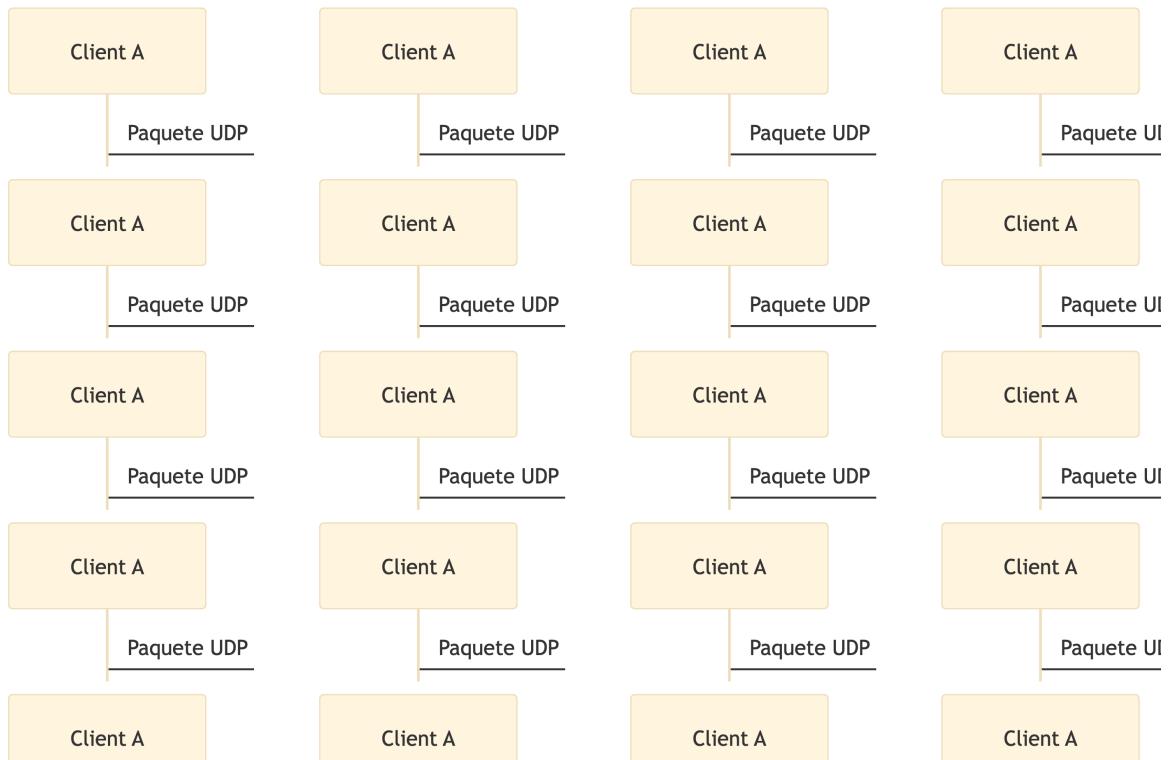


Figura 4.1: Ejemplo de envío de 5 paquetes a través de Internet con UDP. “Internet” en este diagrama representa todo el proceso de envío.

Los protocolos más comunes en la capa de transporte son TCP y UDP, que veremos a lo largo de este capítulo. Los dispositivos tienen generalmente en su kernel implementados estos protocolos y es un proceso del sistema. A través de los sockets nos podemos conectar a TCP o UDP. Este socket corre en un proceso, ya que en la Capa de Aplicación lo que se comunican son procesos entre sí. Para distinguir entre los diferentes sockets, se les otorga una identificación:

- En TCP los sockets se identifican por (IP origen, puerto origen, IP destino, puerto destino).
- En UDP los sockets se identifican por (IP origen, puerto origen).

Los puertos son identificadores numéricos desde 1 a 65535. Esta asignación puede ser manual o automática. Cuando creamos un socket en un servidor, la asignación generalmente es manual y siempre la misma, de tal forma que los procesos que se comunican lo pueden saber “de memoria”. Cuando abrimos un socket desde un cliente para conectarnos con un servidor, la asignación del puerto del cliente es aleatoria, ya que el puerto específico del cliente no es relevante.

Ahora que lo hemos visto de forma intuitiva vamos a definirlo un poco más formalmente. En la capa de transporte los protocolos tienen dos tareas comunes, la **multiplexación** y la **demultiplexación**. La multiplexación es el proceso por el cual recogemos información de diferentes sockets y lo enviamos por un único medio. Por el contrario, la demultiplexación es el proceso por el cual recibimos los segmentos por el medio único y lo enviamos a los sockets correspondientes. A modo de analogía se puede ver como un proceso de envío de cartas. La multiplexación sería el buzón de correos donde dejamos las cartas. La demultiplexación sería el personal de correos cogiendo las cartas y llevándolas a sus destinatarios. Posteriormente veremos alguna particularidad respecto a la multiplexación y demultiplexación entre TCP y UDP.

Otro concepto interesante es la **transferencia fiable**, que es básicamente aquella en la que la información llega tal cual se envió. Es decir, no se corrompe ningún bit, no se pierde información (paquetes) y la información se entrega en un orden correcto. Cuando queremos una transferencia fiable tenemos dos opciones, o bien utilizamos protocolos fiables que ya lo implementen nosotros, o implementamos nosotros esas características de tal forma que podamos tener una comunicación fiable sobre un medio no fiable.

En las siguientes secciones veremos los protocolos UDP y TCP con más detalle.

## 4.2. Protocolos

### 4.2.1. UDP (User Datagram protocol)

UDP (User Datagram protocol) es un protocolo minimalista dentro de la familia de protocolos de la capa de transporte. Implementa el mínimo que debe hacer un protocolo de transporte [RFC 768]. UDP sacrifica las garantías de entrega por algo más valioso en ciertos escenarios:

velocidad pura y simplicidad. Esto es especialmente útil en videojuegos interactivos, DNS o transmisión de videos. Las características principales de UDP son las siguientes:

- **Protocolo ligero y simple:** Es un protocolo basado en el principio best-effort. Esta aproximación significa que hace todo lo posible por entregar los datos al destinatario, pero no ofrece ninguna garantía sobre la entrega de los mismos, ni nos enteraremos sino se entregan debido a que se pierden o tienen errores.
- **No orientado a conexión:** Cuando vamos a enviar información no es necesario establecer una conexión previa entre receptor y emisor. Podríamos decir que cada paquete que se envía es autosuficiente, tiene toda la información necesaria para representar el “estado” de la conexión. Si se pierde, no hay mecanismo para recuperarlo. Esta independencia tiene grandes consecuencias. Primero, simplifica la implementación. Segundo, se elimina la necesidad del proceso de handshake típico de los protocolos orientados a conexión, reduciendo tanto la latencia inicial como la complejidad del protocolo. Tercero, reduce considerablemente los recursos necesarios en el servidor, ya que este no tiene que mantener ningún estado.
- **Entrega no fiable y sin orden:** UDP no ofrece ninguna garantía de entrega sobre la información que se envía. Esta información puede perderse, puede duplicarse, o pueden llegar desordenados. A veces se denomina UDP como protocolo “fire-and-forget”, es decir, que envías el paquete y te olvidas de que ha existido, independientemente de que llegue o no.
- **Integridad básica:** UDP tiene una comprobación de integridad a través de un checksum. Cuando el paquete llega a su receptor, UDP comprueba que el checksum es correcto, y en caso negativo, el paquete se descarta de forma silenciosa.
- **Multiplexación y demultiplexación:** La multiplexación y demultiplexación se realiza mediante el uso de números de puerto, que identifican de manera única los puntos finales de comunicación dentro de un host.

Respecto a las características no proporcionadas, tenemos el control de flujo, control de congestión, temporización, tasa de transferencias mínima y seguridad. Para implementar control de flujo, control de congestión y temporización necesitaríamos tener un estado en cliente y servidor, así como enviar mensajes de control, lo cual entra en conflicto con el principio de best-effort y no ser orientado a conexión. La tasa de transferencia mínima no es posible siendo agnósticos del medio de transporte y requeriría de estado en los routers, lo cual va en contra de la estructura actual de Internet y dificultaría su escalabilidad. Por último, la seguridad, dependiendo del tipo de algoritmo, probablemente requeriría compartir información previamente de forma segura (claves de cifrado) o autoridades centrales como en el caso de HTTPS. En ambos casos, se complicaría el protocolo.

La simplicidad de UDP nos ofrece sin embargo, otra opción. Implementar nosotros mismos a nivel de capa de aplicación las garantías que consideremos necesarias y no pagar el “precio” por las que no vamos a utilizar. Por ejemplo, supongamos que vamos a desarrollar un juego y enviamos las actualizaciones del jugador con estos requisitos:

- Se ignorarán los paquetes fuera de orden. Si enviamos (A B C), y UDP recibe (A C B), a nivel de aplicación descartaríamos B, resultando en (A C).
- Se ignorarán los duplicados. Si enviamos (A B C), UDP recibe (A A B), a nivel de aplicación descartamos la segunda A, resultando en (A B).

Para esta implementación nuestro protocolo podría añadir un número de paquete en los primeros bytes de UDP (antes de la actualización del juego), y el servidor tener un estado del último paquete que recibió. Cada vez que recibimos un paquete lo aceptamos si el número de paquete del servidor < número del paquete recibido, y lo rechazamos de otra forma. Con este pequeño y sencillo protocolo hemos conseguido ignorar paquetes fuera de orden y duplicado con una sobrecarga mínima en el protocolo, y sin la sobrecarga del resto de funcionalidades que no son necesarias.

La estructura de paquete de UDP es la siguiente:

0



S

32



64



Como podéis ver la estructura del paquete es realmente simple en comparación con el resto de protocolos. El puerto de destino le permite a UDP demultiplexar correctamente el paquete, mientras el de origen le permite al servidor contestar. La longitud indica el tamaño de los datos, que es un número de 16 bits, es decir, que podría ser hasta 65535 (no incluido), pero en la práctica está limitado por el MTU. Por último, el checksum que es una forma de verificar la integridad del paquete. Esta verificación es contra cambios accidentales, pero puede ser manipulada.

El cálculo del checksum UDP sigue un procedimiento sistemático que garantiza la verificación de integridad de todo el datagrama:

- Preparación de datos: Se concatena la pseudo-cabecera IP, la cabecera UDP (con checksum inicializado a cero) y los datos de aplicación, añadiendo un byte de padding si la longitud total es impar.
- Cálculo aritmético: Los datos se dividen en palabras de 16 bits que se suman usando aritmética de complemento a uno, incorporando cualquier carry al resultado final.
- Complemento final: Se calcula el complemento a uno del resultado y se inserta en el campo checksum de la cabecera UDP.

El proceso de verificación en el receptor utiliza el mismo algoritmo pero incluye el checksum recibido en el cálculo, esperando obtener 0xFFFF si no hay errores. Si el resultado difiere de 0xFFFF, el datagrama se descarta silenciosamente sin notificación al emisor. La pseudo-cabecera proporciona verificación adicional del direccionamiento correcto, validación del protocolo UDP, y consistencia entre las longitudes reportadas por IP y UDP. Es importante tener en cuenta, que el mecanismo solo detecta errores pero no los corrige, y UDP no implementa retransmisión automática de datagramas corruptos.

UDP es especialmente adecuado para aplicaciones donde la velocidad prima sobre la garantía de entrega, incluyendo multimedia streaming (tolerante a pérdidas menores pero sensible a interrupciones por control de congestión), consultas DNS que requieren respuestas rápidas, protocolos de administración de red como SNMP, sistemas de enrutamiento como RIP, gaming online donde la latencia baja es crítica, y como base para protocolos modernos de transporte como QUIC/HTTP3 que implementan sus propios mecanismos de confiabilidad optimizados.

#### 4.2.2. TCP (Transmission Control Protocol)

TCP (Transmission Control Protocol) es el protocolo de transporte más utilizado en Internet y representa el extremo opuesto a UDP en términos de garantías y complejidad [RFC 793]. TCP prioriza la confiabilidad y el orden de los datos sobre la velocidad pura, siendo fundamental para aplicaciones como navegadores web, correo electrónico, transferencia de archivos y cualquier servicio que requiera integridad absoluta de los datos. Las características principales de TCP son las siguientes:

- **Protocolo confiable y complejo:** TCP implementa múltiples mecanismos para garantizar que todos los datos enviados lleguen al destinatario en el orden correcto y sin errores. Esta confiabilidad viene al costo de mayor complejidad, latencia y overhead del protocolo.
- **Orientado a conexión:** Antes de enviar cualquier dato, TCP requiere establecer una conexión formal entre cliente y servidor mediante un proceso de handshaking de tres fases. Esta conexión mantiene estado en ambos extremos, permitiendo el seguimiento de cada byte enviado y recibido.
- **Entrega fiable y ordenada:** TCP garantiza que todos los datos enviados lleguen al destinatario exactamente una vez y en el mismo orden en que fueron enviados. Implementa mecanismos de detección de pérdidas, duplicados y reordenamiento automático.

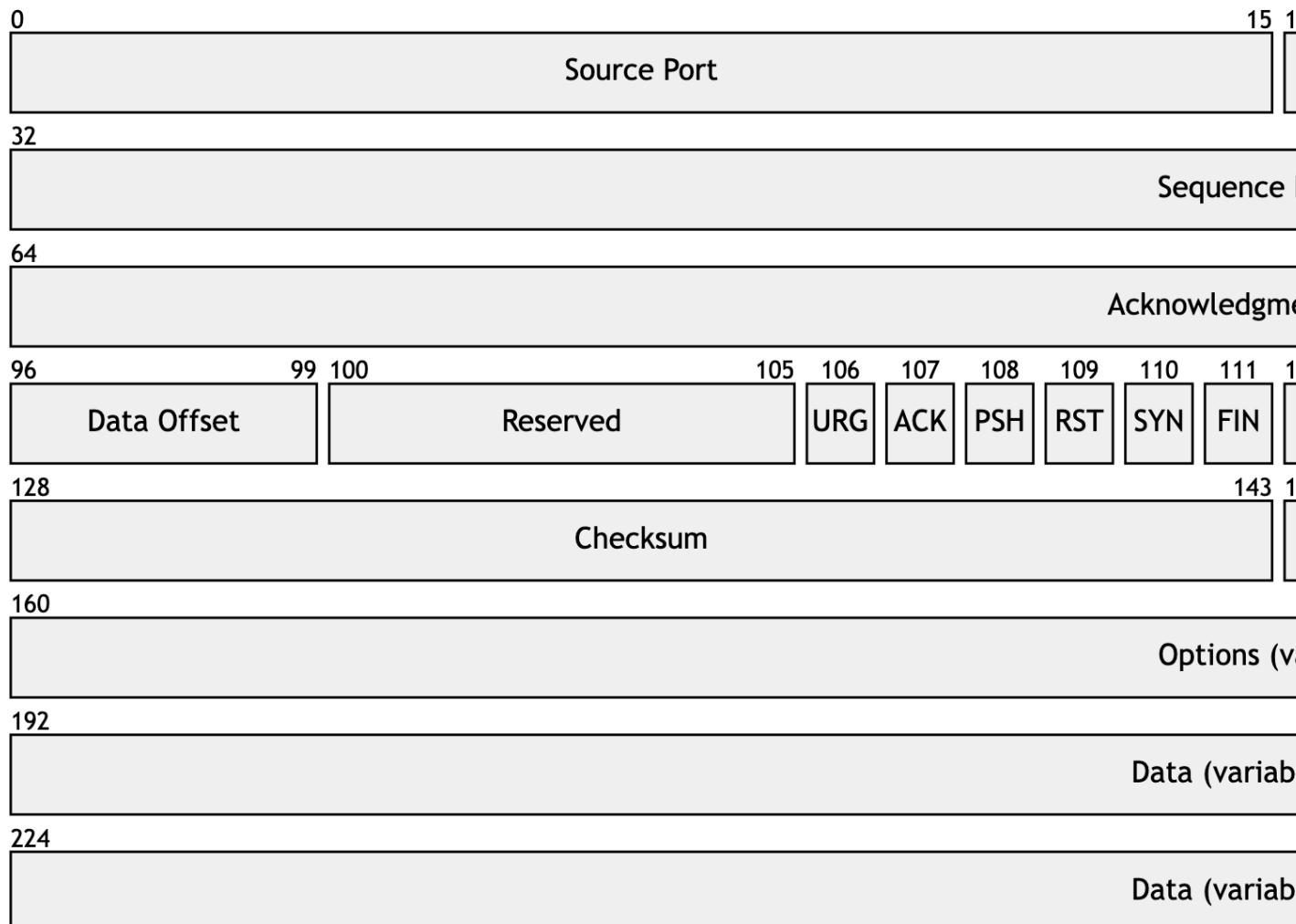
- **Control de flujo:** TCP implementa mecanismos para evitar que el emisor sature al receptor, ajustando automáticamente la velocidad de envío según la capacidad de procesamiento del destinatario a través del campo “Window”.
- **Control de congestión:** TCP detecta y responde a la congestión de la red, reduciendo automáticamente su tasa de transmisión cuando detecta pérdidas o aumentos en la latencia, contribuyendo así a la estabilidad general de Internet.
- **Multiplexación y demultiplexación:** Al igual que UDP, TCP utiliza números de puerto para identificar los diferentes servicios y aplicaciones en un mismo host.

Respecto a las características que TCP no proporciona, tenemos la temporización específica, tasa mínima garantizada de transferencia y seguridad nativa. TCP no puede garantizar una tasa mínima de transferencia porque debe adaptarse dinámicamente a las condiciones cambiantes de la red. La temporización específica no es posible debido a la naturaleza variable de Internet y los mecanismos de retransmisión que pueden introducir retrasos impredecibles. La seguridad debe implementarse en capas superiores (como TLS/SSL) ya que TCP se centra únicamente en la confiabilidad del transporte.

La robustez de TCP permite que las aplicaciones se enfoquen en su lógica de negocio sin preocuparse por los detalles de la transmisión de datos. Por ejemplo, cuando un navegador web solicita una página:

- TCP garantiza que todos los bytes del HTML, CSS, JavaScript e imágenes lleguen completos y en orden.
- Si algún paquete se pierde en la red, TCP lo detecta y retransmite automáticamente.
- Si la red se congestionada, TCP reduce su velocidad para no empeorar la situación.
- El navegador recibe los datos como si fuera un flujo continuo y confiable de bytes.

La estructura de paquete de TCP es considerablemente más compleja que UDP:



## Estructura del TCP

Como se puede observar, la cabecera TCP es mucho más rica en información que UDP. Los puertos de origen y destino funcionan igual que en UDP para la multiplexación. El número de secuencia identifica la posición del primer byte de datos en el flujo, mientras que el número de acknowledgment indica el siguiente byte que el receptor espera recibir, implementando así el mecanismo de confirmación acumulativa.

Los flags de control son cruciales para el funcionamiento de TCP:

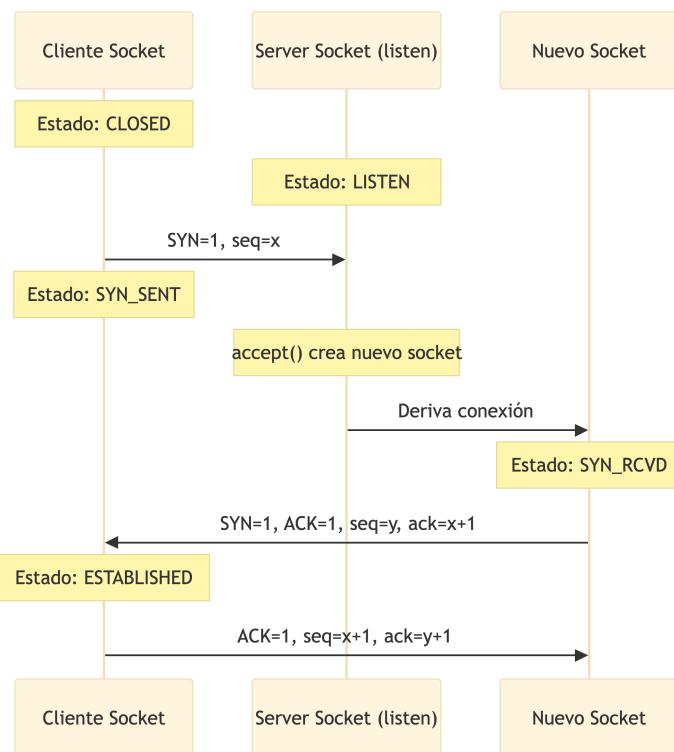
- **SYN**: Utilizado para sincronizar números de secuencia durante el establecimiento de conexión.
- **ACK**: Indica que el campo de acknowledgment es válido.

- **FIN**: Señala el fin de los datos del emisor.
- **RST**: Fuerza el reinicio de la conexión.
- **PSH**: Sigue directamente al proceso aplicación.
- **URG**: Indica datos urgentes.

El campo Window implementa el control de flujo, indicando cuántos bytes está dispuesto a recibir el destinatario. El checksum funciona de manera similar a UDP pero cubriendo todo el segmento TCP.

#### 4.2.2.1. Establecimiento de Conexión: Handshake de Tres Fases

El proceso de establecimiento de conexión TCP es un ejemplo de sincronización distribuida:



**Fase 1 - SYN:** El cliente envía un segmento con SYN=1 y un número de secuencia inicial aleatorio (x). Este número aleatorio es crucial para la seguridad, evitando ataques de predicción de secuencia.

**Fase 2 - SYN+ACK:** El servidor responde con SYN=1, ACK=1, su propio número de secuencia inicial (y) y confirma el número del cliente incrementado en uno (ack=x+1).

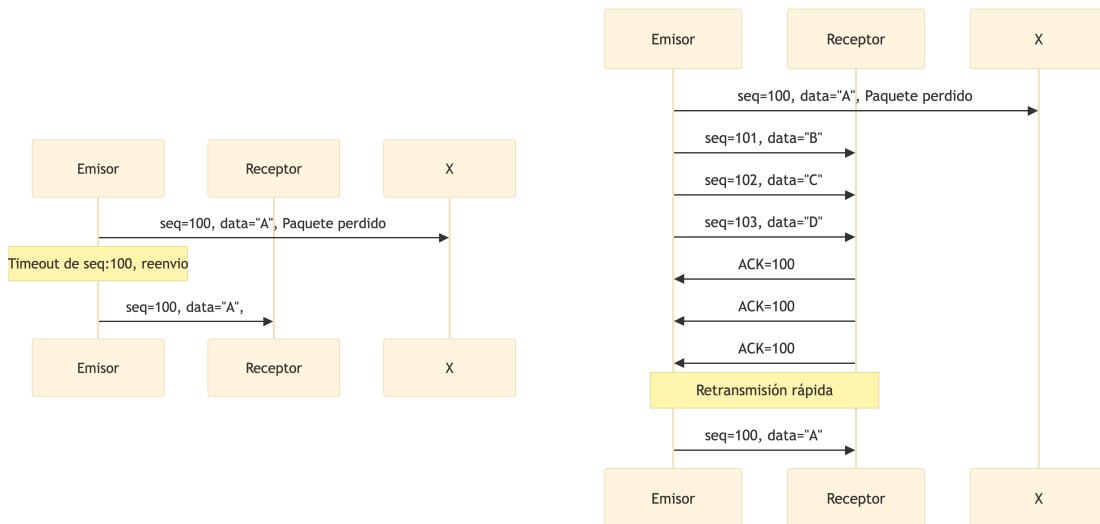
**Fase 3 - ACK:** El cliente confirma el número de secuencia del servidor (ack=y+1), estableciendo oficialmente la conexión bidireccional.

Durante este proceso se negocian parámetros importantes como el MSS (Maximum Segment Size), opciones de ventana deslizante y otras extensiones TCP.

#### 4.2.2.2. Mecanismos de Confiabilidad

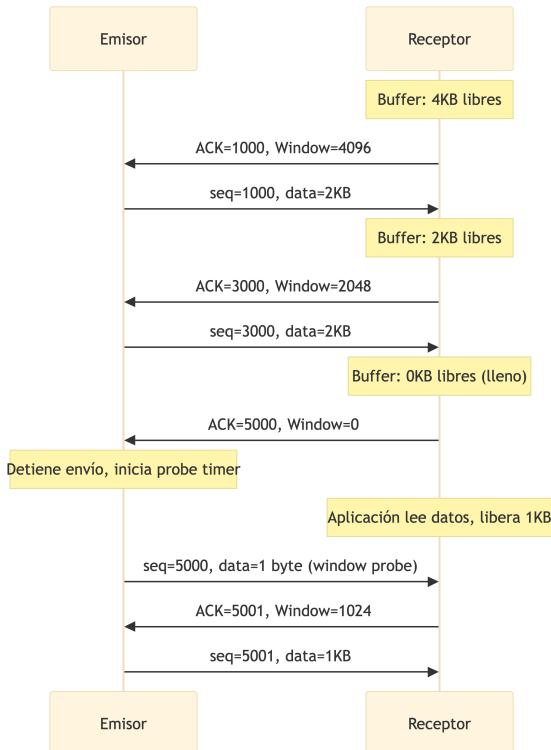
TCP implementa varios mecanismos para garantizar la entrega confiable. **Números de Secuencia y ACKs:** Cada byte en el flujo TCP tiene un número de secuencia único. Los ACKs son acumulativos, lo que significa que un ACK para el byte N confirma la recepción correcta de todos los bytes desde el inicio hasta N-1. **Detección de Pérdidas:** TCP utiliza dos métodos principales:

- **Timeout:** Si no recibe ACK en un tiempo determinado, asume pérdida y retransmite.
- **ACKs duplicados:** Si recibe tres ACKs duplicados para el mismo número de secuencia, asume pérdida del siguiente segmento y retransmite inmediatamente (Fast Retransmit).



#### 4.2.2.3. Control de Flujo

TCP mantiene buffers tanto en emisión como en recepción, permitiendo el manejo de segmentos fuera de orden y la optimización del flujo de datos. El control de flujo TCP es un mecanismo sofisticado que previene el desbordamiento del receptor:



La **VentanaRecepcion** se calcula como:

$$\text{VentanaRecepcion} = \text{BufferRecepcion} - (\text{UltimoByteRecibido} - \text{UltimoByteLeido})$$

El emisor debe asegurar que:

$$\text{UltimoByteEnviado} - \text{UltimoByteReconocido} \leq \text{VentanaRecepcion}$$

Cuando la ventana de recepción se reduce a cero, el emisor detiene el envío pero continúa sondeando periódicamente con segmentos de un byte para detectar cuándo hay espacio disponible nuevamente.

#### 4.2.2.4. Control de Congestión

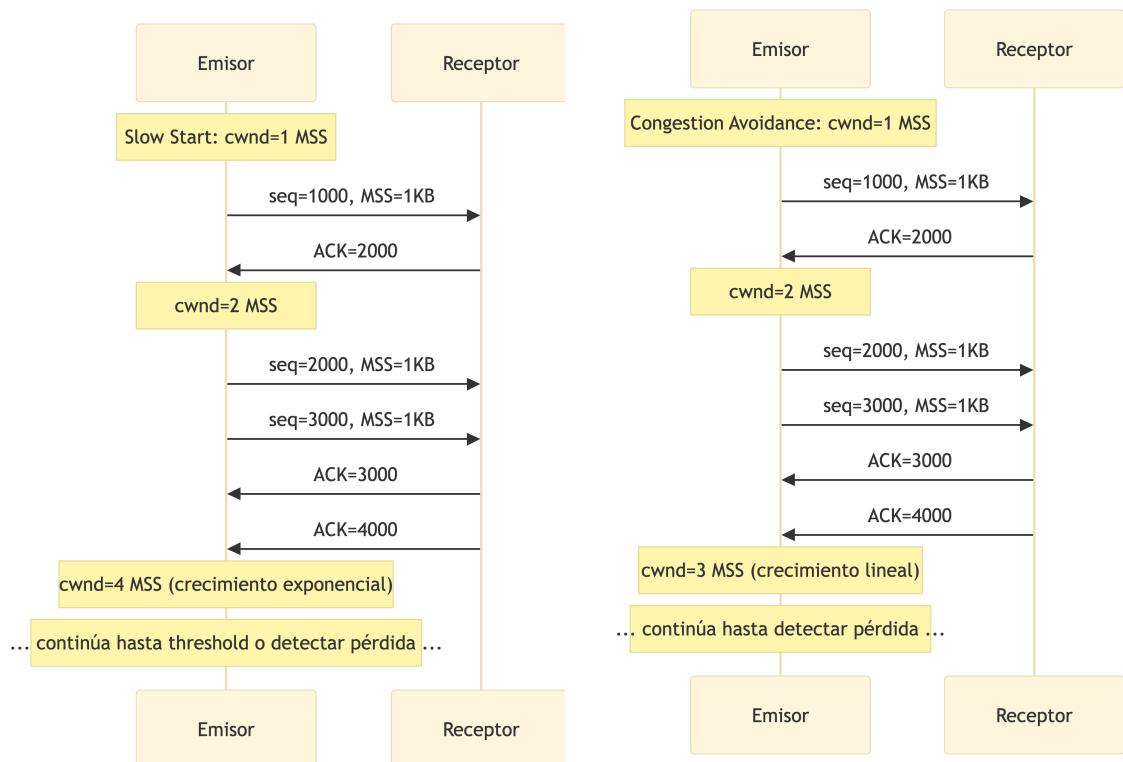
El control de congestión TCP es uno de los algoritmos más importantes de Internet. Utiliza la **ventana de congestión** para regular la velocidad de envío. La ventana de congestión es una variable del emisor que representa el número máximo de bytes que pueden estar en el “aire” en la red, es decir, el número máximo de bytes que pueden ser enviados sin que sean reconocidos (ack). Se combina con la ventana de recepción para determinar la tasa de envío

actual, siendo la tasa efectiva el mínimo de ambas. La ventana de congestión se regula en base a dos mecanismos:

- Slow start (arranque lento): Se inicia con una ventana de congestión igual a 1 MSS. Duplica la ventana cada RTT (ver ejemplo en ([slow-start?](#))) hasta detectar pérdida o alcanzar un umbral. Es decir, el tamaño crece exponencialmente y es ideal para descubrir el ancho de banda.
- Congestion avoidance (evitación de congestión): Tiene un comportamiento más conservador. Incrementa 1 MSS por cada RTT (ver el ejemplo en ([congestion-avoidance?](#))).

y dos eventos que regulan el paso entre los dos mecanismos:

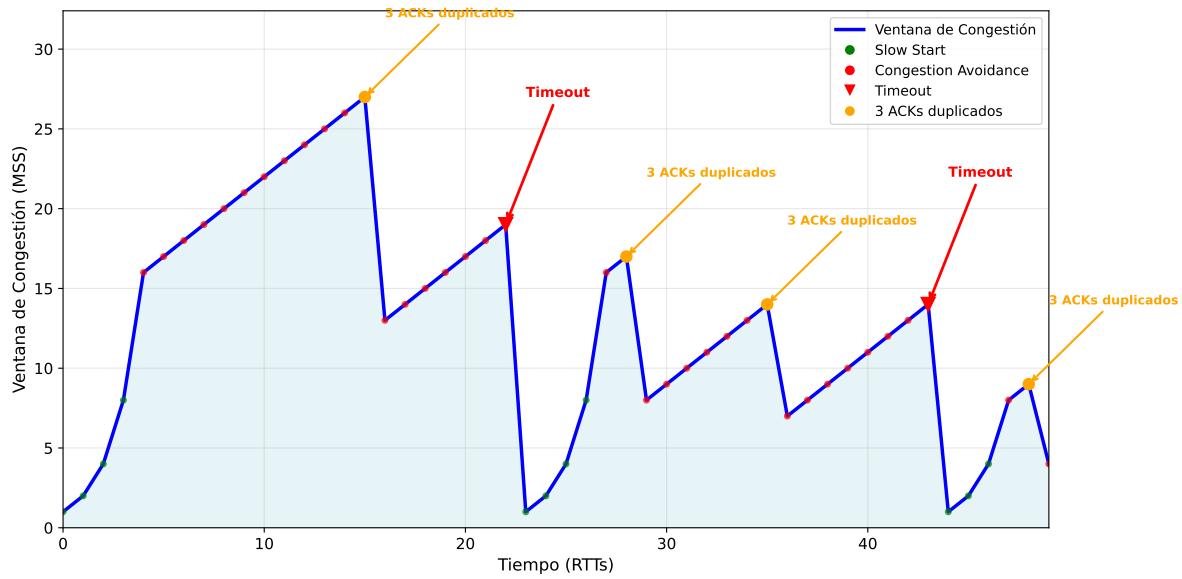
- Salta un temporizador: Pérdida severa. Se reduce la ventana de congestión a 1 MSS y pasamos a modo slow start.
- Tres ACK duplicados: Pérdida no tan severa, se reduce la ventana de congestión a la mitad.



Este mecanismo crea el característico patrón de “diente de sierra” en el throughput de TCP,

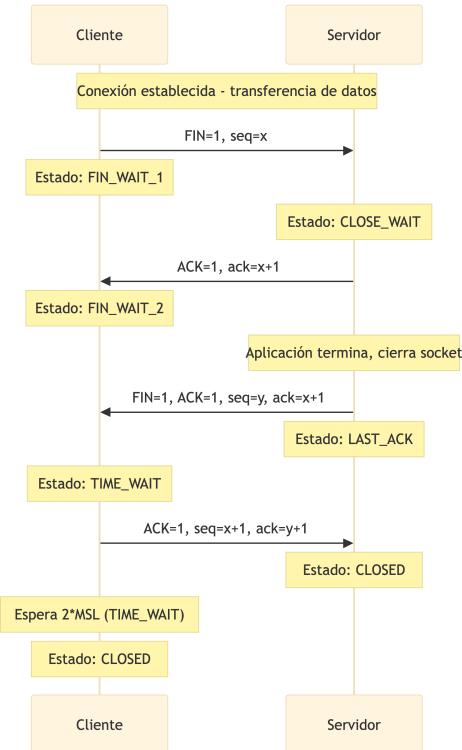
donde la ventana crece gradualmente hasta detectar congestión, se reduce drásticamente, y vuelve a crecer.

<Figure size 4200x2400 with 0 Axes>



#### 4.2.2.5. Terminación de Conexión

La terminación de conexión TCP requiere un proceso de cuatro fases debido a su naturaleza full-duplex:



1. **FIN del Cliente:** El cliente envía FIN indicando que terminó de enviar datos
2. **ACK del Servidor:** El servidor confirma la recepción del FIN
3. **FIN del Servidor:** El servidor envía su propio FIN cuando termina de enviar
4. **ACK del Cliente:** El cliente confirma y entra en estado TIME\_WAIT

El estado TIME\_WAIT es crucial para manejar ACKs retrasados y asegurar que la conexión se cierre completamente.

#### 4.2.2.6. Equidad y Coexistencia

TCP está diseñado para ser “fair” cuando múltiples conexiones comparten el mismo enlace. El algoritmo de control de congestión asegura que N conexiones TCP compartan equitativamente un enlace de capacidad R, obteniendo aproximadamente  $R/N$  cada una.

Sin embargo, esta equidad tiene limitaciones:

- **Aplicaciones UDP:** No implementan control de congestión, pueden monopolizar ancho de banda.
- **Conexiones paralelas:** Una aplicación puede abrir múltiples conexiones TCP para obtener mayor throughput.
- **RTT diferentes:** Conexiones con menor RTT pueden obtener ventaja.

### **4.3. Comparativa de TCP vs UDP para videojuegos**

La decisión de elegir entre TCP y UDP depende de la interactividad y la tolerancia a pérdidas de información del juego. Por ejemplo, si se requiere de latencias de menos de 50ms, con actualizaciones frecuentes y la información nueva es más valiosa que la vieja, UDP es el claro ganador. Esto se consigue gracias a unas cabeceras mucho más pequeñas y la ausencia de tráfico de control. Otro aspecto positivo es que el servidor necesita menos recursos, al no tener que gestionar la lógica de gestión ni mantener el estado. Algunos videojuegos donde UDP es mejor opción es en shooters o juegos de lucha debido a su alta interactividad.

Por contra, si la latencia es de 100ms a 200ms, se necesita una entrega ordenada garantizada y detección y corrección de errores, TCP es la mejor opción. Con tolerancias de latencia mayores, no tenemos que preocuparnos por bloqueo de cabeza de línea, es decir, que un paquete perdido impida el procesamiento de los posteriores que ya llegaron. También tendríamos que tener en cuenta las latencias variables, debido a retransmisiones, lo cual podría generar saltos de estado, incluso con tolerancias de 100ms. Por último, habría que considerar también que incurriremos en un mayor tráfico de red, tanto por el tráfico de control como por el mayor tamaño de los paquetes TCP. Los juegos de rol y por turnos son ejemplos de juegos que se adaptan muy bien a TCP.

Pasando a ejemplos concretos, World of Warcraft es un ejemplo de juego implementado sobre TCP. Los hechizos y ataques necesitan entrega garantizada para mantener la consistencia, así como la actualizaciones del inventario y estado de las misiones. Generalmente los MMORPGs pueden soportar latencias de 100s a 200ms.

En el caso de Counter Strike se utiliza UDP debido a que la retroalimentación inmediata es más importante que la entrega garantizada. Por ejemplo, las actualizaciones de posición y disparos necesitan una latencia muy baja. Es tan importante, que aún con UDP, es necesario utilizar técnicas de interpolación de estados en los clientes para conseguir transiciones suaves. Esta interpolación limita el efecto de paquetes perdidos.

## 5 Capa de aplicación

La capa de aplicación define los protocolos que utilizarán las aplicaciones para intercambiar datos. Las aplicaciones generalmente se representan con procesos, y por lo tanto, la capa de aplicación se centra en la comunicación entre procesos. Este nivel de ejecución nos va a quedar más claro si tenemos en cuenta que podemos crear nuestros propios protocolos que se ejecuten a nivel de capa de aplicación.

A continuación veremos un ejemplo de protocolo definido en la capa de aplicación, que realiza una función de “echo”, es decir, repite la información que recibe. Además, este pequeño ejemplo nos servirá para introducir los tipos de arquitecturas que pueden tener una aplicación de red. En concreto, este ejemplo utilizará una arquitectura cliente - servidor. En este tipo de arquitectura, tenemos un host (servidor) que está siempre activo con una dirección IP conocida y que ofrece servicio a otros hosts (clientes). Estos clientes podrán estar activos o no, y no se comunican entre ellos, sólo con el servidor. En este ejemplo tendremos un servidor, cuya funcionalidad será devolver la información recibida, con el formato “Echo: {message}”, donde {message} es el contenido recibido. El servidor continuará contestando la petición de los clientes hasta que reciba el mensaje “quit”, mediante el cual se cerrará la conexión entre ambos.

A continuación se muestra el servidor. Está programado en JavaScript, que veremos en la siguiente parte del libro. No os preocupéis si no entendéis todo, es simplemente a modo de ilustración.

```
const net = require('net');

function echoServer() {
    const server = net.createServer();

    server.on('connection', (socket) => {
        const clientAddress = `${socket.remoteAddress}:${socket.remotePort}`;
        console.log(`Client connected: ${clientAddress}`);

        handleClient(socket, clientAddress);
    });

    server.listen(8888, () => {
        console.log('Echo server listening on localhost:8888');
    });
}
```

```

}

function handleClient(socket, clientAddress) {
    socket.on('data', (data) => {
        const message = data.toString('utf-8').trim();
        console.log(`[${clientAddress}] ${message}`);

        if (message.toLowerCase() === 'quit') {
            socket.end();
            return;
        }

        // Echo back
        const echoResponse = `Echo: ${message}`;
        socket.write(echoResponse);
    });

    socket.on('close', () => {
        console.log(`[${clientAddress}] Disconnected`);
    });

    socket.on('error', (err) => {
        console.log(`[${clientAddress}] Error: ${err.message}`);
    });
}

```

Este servidor está formado por dos funciones, la función “handleClient” y la función “echoServer”. Empezando por “echoServer”, en las primeras líneas se crea un servidor TCP usando el módulo ‘net’ de Node.js. El servidor utiliza el modelo basado en eventos de JavaScript - cuando se conecta un cliente, se dispara automáticamente el evento ‘connection’, que delega el procesamiento del cliente a “handleClient”. La función “handleClient” define el “protocolo” mediante eventos: escucha el evento ‘data’ de forma indefinida hasta que se reciba un mensaje con la palabra “quit”, procesa los datos recibidos y los devuelve al cliente con el formato “Echo: {message}”. Esta ejecución también puede terminar cuando se disparan los eventos ‘close’ (cliente desconecta) o ‘error’ (error en la conexión), que son manejados automáticamente por el sistema de eventos de Node.js. Si os fijáis en esta función trabajamos con la variable “socket”, que es la interfaz entre la capa de aplicación y la capa de transporte. Dicho de otra forma, es la interfaz que tenemos de interactuar con la capa inferior, y la capa inferior con nosotros. El servidor queda escuchando en localhost:8888 y puede manejar múltiples clientes simultáneamente gracias al bucle de eventos asíncrono de Node.js.

Ahora pasaremos a la parte del cliente:

```

import socket

def echo_client():
    """Interactive echo client"""

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect(('localhost', 8888))

    while True:
        message = input("Enter message: ")

        if message.lower() == 'quit':
            client_socket.send(message.encode('utf-8'))
            break

        client_socket.send(message.encode('utf-8'))
        response = client_socket.recv(1024).decode('utf-8')
        print(f"Server response: {response}")

    client_socket.close()

```

En este caso el código está hecho con Python, no es un requisito necesario y podría estar en JavaScript, pero quería remarcar que la definición de protocolos en red permite la comunicación entre dos procesos que están en la misma u otra máquina, independientemente de su lenguaje de programación <sup>1</sup>. En este cliente de Python tenemos una única función que representa al cliente, “echo\_client”, donde en las primeras líneas establecemos una conexión con el servidor de JavaScript. Fijaros en el ('localhost', 8888), con esta combinación de identificador de máquina, “localhost”, podemos identificar el host donde está el servidor, y con el puerto, 8888, podemos identificar el proceso que corresponde al servidor. Como en el anterior ejemplo, tenemos un “socket” que permite una interacción bidireccional con la capa de transporte. No os preocupéis por estos detalles, los veremos en el siguiente capítulo.

Con este ejemplo hemos ilustrado los tres conceptos clave de este capítulo, los protocolos de la capa de aplicación, la arquitectura de las aplicaciones de red <sup>2</sup>, y los sockets que permiten la interacción entre la capa de aplicación y la capa de transporte. En los siguientes apartados profundizaremos en estos temas. Primero, veremos en detalle los sockets. Después, indagaremos en las arquitecturas de aplicaciones en red. Posteriormente veremos protocolos utilizados en la

---

<sup>1</sup>Dos máquinas pueden tener diferente **endianness** (orden de bytes): *big-endian* almacena el byte más significativo primero, mientras que *little-endian* lo guarda al final. Los protocolos de red usan **network byte order** (big-endian) para garantizar que ambas máquinas interpreten los datos correctamente, independientemente de su arquitectura interna.

<sup>2</sup>En realidad son para sistemas distribuidos. Pero las aplicaciones de red son inherentemente sistemas distribuidos.

actualidad como HTTP que utilizamos cuando navegamos por la web, SMTP, IMAP y POP que utilizamos en las aplicaciones de correo, entre otros.

## 5.1. Socket

Los sockets son la interfaz de programación que permite a las aplicaciones comunicarse con la capa de transporte. Actúan como un punto de conexión bidireccional entre la capa de aplicación y la capa de transporte, proporcionando una abstracción que oculta los detalles de bajo nivel de la comunicación en red. En esencia, un socket es un endpoint de comunicación que permite que los procesos intercambien datos, ya sea en la misma máquina o a través de una red. La API de sockets fue introducida en BSD4.1 UNIX en 1981. Fue explícitamente creada, usada y lanzada por las aplicaciones de red. Está basada en el paradigma cliente/servidor.

Cuando una aplicación necesita comunicarse a través de la red, crea un socket que especifica el protocolo de transporte a utilizar (TCP o UDP), la dirección IP del host de destino, y el número de puerto del proceso receptor. El socket encapsula toda la información necesaria para establecer y mantener una conexión de red, proporcionando una interfaz uniforme independientemente del protocolo de transporte subyacente. Los sockets se pueden clasificar según el protocolo de transporte que utilizan, siendo los más comunes los sockets TCP y UDP, cada uno con características y casos de uso específicos. Los detalles del funcionamiento interno de TCP y UDP los veremos en el capítulo de la capa de transporte.

Para identificar un proceso se necesita:

- **IP del host:** Dirección única de 32 bits (IPv4)
- **Número de puerto:** Asociado con el proceso en el host
- Ejemplos: HTTP (puerto 80), HTTPS (puerto 443), DNS (puerto 53)

### 5.1.1. Sockets TCP

Los sockets TCP (Transmission Control Protocol) proporcionan una comunicación confiable y orientada a conexión entre procesos. Antes de que los datos puedan ser intercambiados, se debe establecer una conexión explícita entre el cliente y el servidor, lo que garantiza que ambos extremos estén listos para la comunicación. Las características principales del socket TCP son las siguientes:

- **Orientado a conexión:** Requiere establecer una conexión antes del intercambio de datos.
- **Confiabilidad:** Garantiza que todos los datos enviados lleguen al destino sin errores y en orden.
- **Control de flujo:** Evita que el emisor saturé al receptor.
- **Control de congestión:** Adapta la velocidad de envío según las condiciones de la red.

- **Full-duplex:** Permite comunicación bidireccional simultánea.

Para crear un socket TCP de tipo servidor, es decir, que siempre está activo y está esperando las conexiones de los clientes (arquitectura cliente-servidor), utilizaremos el módulo net de Javascript. Dentro de este módulo, utilizaremos la función “createServer” para crear un socket de tipo servidor de TCP. Posteriormente, utilizaremos el método “listen” para escuchar en un puerto en concreto. En este caso, el 8888. El segundo parámetro, que en este caso es “localhost”, es opcional, y quiere decir que los clientes tienen que estar en esa red. Si obviamos el parámetro, los clientes podrán conectarse desde cualquier otra máquina. Finalmente, el último parámetro es un “callback” que se ejecutará una vez el servidor socket esté escuchando en el puerto correctamente.

```
const net = require('net');

// Crear servidor TCP
const server = net.createServer();

// Configurar el servidor para escuchar en puerto 8888
server.listen(8888, 'localhost', () => {
    console.log('Servidor TCP escuchando en localhost:8888');
});
```

Por ahora hemos bloqueado un puerto dentro de nuestra máquina y estamos esperando a que se conecten los clientes. Ahora, tenemos que gestionar los eventos de conexión. Para ello, utilizaremos el método “server.on”, especificándole que el evento que queremos escuchar es la conexión “connection” (primer parámetro). El segundo parámetro es un manejador de conexión (una función), que recibe un “socket”, y que será invocada por el servidor socket por cada cliente que se conecte. Recordemos que TCP está orientado a conexión. En nuestro código esa conexión con el cliente se realizará a través del “socket” que recibe el manejador.

```
// Manejar nuevas conexiones
server.on('connection', (socket) => {
    console.log('Cliente conectado:', socket.remoteAddress);
    // El socket está listo para intercambiar datos
});
```

Sobre este socket que hemos recibido en el manejador podemos escuchar diferentes eventos. El primer evento que veremos es “data”. Este evento se invocará cada vez que el socket reciba información desde el otro socket. Estos datos se procesan a través de un manejador que le pasaremos cuando escuchamos el evento “data”. El manejador recibirá un parámetro, que en el siguiente código se denomina “data”, y contendrá los datos enviados por el otro integrante de la conexión.

```
socket.on('data', (data) => {  
})
```

Por contextualizar, supongamos que tenemos un juego con dos jugadores que están en diferentes máquinas y estos se comunican con un servidor central. En este método recibiríamos por ejemplo las actualizaciones de estado de cada uno de los jugadores, y tendríamos que actualizar el estado del servidor y notificar al otro jugador.

El siguiente evento es “close”. Este evento se invocará cuando la conexión se haya cerrado. En el manejador que le pasamos como parámetro tendremos que realizar las operaciones oportunas en base al protocolo que estemos definiendo.

```
socket.on('close', () => {  
    console.log(`[${socket.remoteAddress}] Disconnected`);  
});
```

Siguiendo con el ejemplo, este evento podría invocarse si uno de los jugadores se desconecta. En ese caso, se invocaría ese método, el servidor debería actualizar a finalizado el estado del juego, y notificar al otro jugador de que la partida ha terminado.

Por último, tenemos el evento “error”. Este puede ocurrir cuando se cierra la conexión de forma inesperada, por ejemplo, te desconectas de la red. En este caso también se ejecutará el manejador de “close”, así que es recomendable poner la lógica de limpieza allí, ya que el “close” se ejecutará si la conexión se cierra tanto de forma natural como inesperada, mientras que el “error” solo cuando es de forma inesperada. Otro posible caso en el que se ejecuta el “error” es si estamos tratando de escribir en un socket que está cerrado. También puede ocurrir si salta un evento de “timeout” durante el envío de datos.

```
socket.on('error', (err) => {  
    console.log(`[${socket.remoteAddress}] Error: ${err.message}`);  
});
```

Ahora que sabemos como manejar los eventos, sólo nos falta ver como enviar información a través de un socket. Para ello, utilizaremos el método “write”. El segundo parámetro es un manejador que utilizaremos para capturar los errores durante el envío de información.

```
socket.write('Hello', (err) => {  
});
```

Este método lo utilizaríamos para enviar por ejemplo las actualizaciones de estado.

Una vez vista la parte del servidor veremos la del cliente. Para ello necesitaremos también el módulo “net” y crearemos un socket con “new net.Socket()”. Una vez creado el socket, lo conectaremos mediante la instrucción “socket.connect”. El primer parámetro es el puerto donde está escuchando el servidor socket en la máquina identificada por el segundo parámetro. En este caso, la conexión es a “localhost” y el puerto 8888. El tercer parámetro es un callback que se ejecutará una vez la conexión se haya establecido.

```
const net = require('net');

// Crear socket TCP
const socket = new net.Socket();

// Conectar al servidor (establece la conexión TCP)
socket.connect(8888, 'localhost', () => {
    console.log('Conectado al servidor TCP');
    // El socket está listo para intercambiar datos
});
```

Respecto a los métodos por la parte del cliente, son los mismos que explicamos con el socket del servidor (es decir, una vez establecida la conexión). Una vez se establece la conexión, no hay diferencia entre ambos. Como matiz, en el manejador de error del cliente tenemos algunos errores a mayores, como por ejemplo si no se puede establecer la conexión.

Ambos socket tienen que ser cerrados para liberar recursos una vez hayamos terminado. Para ello utilizaremos el método “close”:

```
socket.close()
```

En el caso del servidor también:

```
server.close()
```

Si no lo hacemos el bucle de eventos seguirá activo y la aplicación no terminará.

### 5.1.2. Sockets UDP

Los sockets UDP (User Datagram Protocol) proporcionan una comunicación sin conexión y de mejor esfuerzo. El mejor esfuerzo se refiere a que va a intentar lo mejor que pueda enviar la información al destinatario, pero en caso de que falle, no va a volver a intentarlo ni te notificará. Esto contrasta con TCP que si lo reintenta y en caso de no poder te notifica. Sus características principales son las siguientes:

- **Sin conexión:** No requiere establecer conexión previa
- **Mejor esfuerzo:** No garantiza entrega, orden ni integridad de datos
- **Baja latencia:** Menor overhead que TCP
- **Simplicidad:** Protocolo más simple y directo
- **Broadcast/Multicast:** Soporte nativo para envío a múltiples destinatarios

Para recibir paquetes de UDP, crearemos un servidor de UDP utilizando el paquete “dgram”. El socket se crea mediante la expresión “dgram.createSocket(‘udp4’). En este caso se utiliza “udp4” ya que utilizamos IPv4, pero si queremos utilizar IPv6 sería “udp6”. Veremos las diferencias en el capítulo de capa de red. Una vez creado el socket, nos mantenemos a la escucha con la instrucción “bind”. En este caso, el puerto 8888. El segundo parámetro, en este caso “localhost”, indica que solo aceptaremos peticiones de la red “localhost”. Como en TCP, si lo dejamos vacío será cualquier red. También podremos especificar otras redes. Finalmente tenemos un manejador que se invocará si el socket empieza a escuchar en el puerto 8888 correctamente.

```
const dgram = require('dgram');

// Crear socket UDP
const server = dgram.createSocket('udp4');

// Vincular el socket al puerto 8888
server.bind(8888, 'localhost', () => {
  console.log('Servidor UDP escuchando en localhost:8888');
});
```

Para recibir mensajes, añadimos un manejador al evento “message”. Este manejador recibe dos parámetros. El mensaje, que es lo que nos han enviado desde el socket UDP cliente y el parámetro rinfo, que contiene la información necesaria para identificar el socket que nos envía información.

```
// Escuchar mensajes entrantes
server.on('message', (msg, rinfo) => {
  console.log(`Mensaje recibido de ${rinfo.address}:${rinfo.port}`);
  // No hay conexión establecida, cada mensaje es independiente
});
```

También podremos añadir un manejador de errores con el evento “error”. Los errores podrían ser que no se puede hacer el bind al puerto. Esto puede ocurrir si el puerto ya está en uso o es un puerto reservado y no tenemos los permisos necesarios.

```

socket.on('error', (err) => {
  console.error('Socket error:', err.message);
});

}

```

Para enviar los mensajes, tendremos que crear un socket con el módulo “dgram”. Posteriormente, utilizaremos “createSocket” para crear el socket cliente que nos permitirá enviar información.

```

const dgram = require('dgram');

// Crear socket UDP
const client = dgram.createSocket('udp4');

```

Para enviar la información utilizaremos el método send. Como no tenemos una conexión como en TCP, cada vez que enviamos información tenemos que indicarle cuál es el puerto de destino (8888) y la IP de destino (localhost). El manejador se invocará indicándonos si ha habido un error durante el envío o no. Algunos errores pueden ser que el destino no se pueda alcanzar, que el buffer de UDP esté lleno, entre otros. Como hemos comentado, que el mensaje se haya enviado no quiere decir que el destinatario lo reciba.

```

client.send('Hola servidor UDP', 8888, 'localhost', (err) => {
  if (err) throw err;
  console.log('Mensaje enviado al servidor UDP');
});

```

Una pregunta que os puede surgir con UDP es, ¿Cómo le escribe de vuelta el actual “servidor” al “cliente”? La respuesta es simple, invirtiendo los roles. Cuando creamos nuestro “socket cliente” sin decirle que haga un bind a un puerto determinado, cuando enviamos un mensaje se hace un bind a un puerto aleatorio que esté libre. A través del “rinfo” anterior tenemos tanto “rinfo.address” como “rinfo.port” que son la IP y el puerto. Por lo tanto, podemos escribir al cliente utilizando esa información.

Para receptionar ese mensaje, en el cliente tendríamos que escuchar el evento de “message”:

```

client.on('message', (msg, rinfo) => {
  console.log(`Mensaje recibido de ${rinfo.address}:${rinfo.port}`);
  // No hay conexión establecida, cada mensaje es independiente
});

```

y también podríamos como hicimos antes capturar el evento de errores. Con esto podemos llegar a una interesante conclusión. Tanto el socket del cliente como el del servidor son iguales.

La única diferencia es que en el servidor, le indicamos específicamente en qué puerto queremos escuchar. Esto lo hacemos para facilitar que los demás sepan dónde está ubicado. En el caso del cliente no tenemos esa necesidad. Podemos escoger un puerto aleatorio. Cuando enviamos un mensaje al servidor el sabrá el puerto del cliente y podrá escribirle también.

Finalmente, es necesario cerrar tanto el cliente:

```
client.close()
```

como el servidor

```
server.close()
```

Si no el bucle de eventos seguirá activo y la ejecución no terminará. También se liberarán los recursos.

### 5.1.3. Servicios Requeridos y elección de capa de transporte

Las aplicaciones de red tienen diferentes requisitos en cuanto a los servicios que necesitan de la capa de transporte. Estos requisitos determinan qué protocolo de transporte es más apropiado para cada aplicación específica.

**Transferencia Confiable:** Algunas aplicaciones requieren que todos los datos enviados lleguen al destino sin errores ni pérdidas. Esta característica es fundamental para aplicaciones donde la integridad de la información es crítica. Ejemplos de aplicaciones que requieren una confiabilidad total son la transferencia de archivos, correo electrónico, navegación web, banca online, comercio electrónico. En estos casos, la pérdida de datos podría resultar en archivos corruptos, mensajes incompletos o transacciones fallidas. Por otra parte, algunas aplicaciones son tolerantes a la pérdida de información, como el streaming de audio/vídeo, videoconferencias, juegos en tiempo real. Estas aplicaciones pueden funcionar adecuadamente incluso si se pierden algunos paquetes ocasionalmente, ya que el contenido perdido puede ser interpolado o simplemente ignorado sin afectar significativamente la experiencia del usuario.

**Temporización (Timing):** El tiempo de respuesta es crucial para aplicaciones interactivas y en tiempo real. Algunas aplicaciones son sensibles a la latencia, como los juegos multijugador online, trading de alta frecuencia, aplicaciones de realidad virtual, control remoto de dispositivos. Estas aplicaciones requieren tiempos de respuesta muy bajos (típicamente menos de 50-100ms) para proporcionar una experiencia fluida. En otros casos no es tan importante, como en el correo electrónico, transferencia de archivos en segundo plano, respaldos automáticos. Estas aplicaciones pueden funcionar correctamente con latencias más altas sin afectar significativamente la experiencia del usuario.

**Ancho de Banda:** Las necesidades de ancho de banda varían enormemente entre aplicaciones. Ejemplos de aplicaciones sensibles al ancho de banda son el streaming de vídeo 4K/8K,

videoconferencias de alta calidad, transferencia de archivos grandes, respaldos de bases de datos. Estas aplicaciones requieren una tasa mínima garantizada de transferencia para funcionar correctamente. Cuando una aplicación no es sensible, a veces se denominan elásticas, es decir, estas aplicaciones pueden adaptarse al ancho de banda disponible, funcionando más lento con conexiones limitadas pero manteniéndose operativas. Algunos ejemplos son: Navegación web, correo electrónico, mensajería instantánea.

**Seguridad:** Los requisitos de seguridad incluyen varios aspectos:

- **Confidencialidad:** Garantizar que solo los destinatarios autorizados puedan leer los datos (mediante cifrado).
- **Integridad:** Asegurar que los datos no han sido modificados durante la transmisión.
- **Autenticación:** Verificar la identidad de las partes que se comunican.
- **No repudio:** Garantizar que el emisor no pueda negar haber enviado los datos.

Aplicaciones como banca online, comercio electrónico, mensajería privada y transferencia de documentos confidenciales requieren múltiples aspectos de seguridad, mientras que aplicaciones como streaming público o noticias pueden tener requisitos de seguridad más relajados.

La elección entre sockets TCP y UDP depende de los requisitos específicos de la aplicación:

**Usar TCP cuando:**

- La integridad de datos es crítica
- Se necesita garantizar el orden de los mensajes
- La aplicación puede tolerar mayor latencia
- Se transfieren archivos o datos importantes

**Usar UDP cuando:**

- La velocidad y baja latencia son prioritarias
- La aplicación puede manejar pérdida ocasional de datos
- Se implementan aplicaciones en tiempo real
- Se necesita comunicación multicast o broadcast

Algunos ejemplos de elección son los siguientes:

Aplicación	Confiabilidad	Temporización	Ancho de Banda	Seguridad	Protocolo Típico
Transferencia de archivos	Sí	No crítica	Elástica	Según contenido	TCP
Correo electrónico	Sí	No crítica	Elástica	Sí	TCP

Aplicación	Confiabilidad	Temporización	Ancho de Banda	Seguridad	Protocolo Típico
Navegación web	Sí	Moderada	Elástica	Sí (HTTPS)	UDP (HTTP/3) / TCP (HTTP/1.1-2)
Streaming de vídeo	Tolerante	Crítica	Mínima garantizada	Según contenido	UDP/TCP
Juegos en tiempo real	Tolerante	Muy crítica	Moderada	Sí	UDP
Videoconferencia	Tolerante	Crítica	Mínima garantizada	Sí	UDP/TCP
DNS	Tolerante	Crítica	Elástica	Creciente	UDP/TCP (DoH/DoT)

En esta tabla igual hay un detalle que os llama la atención. Hemos dicho que UDP no es confiable. Se puede perder información o incluso llegar en distinto orden. Sin embargo, en la navegación web que requiere de confiabilidad, se indica que se utiliza UDP cuando el protocolo es HTTP/3. ¿Cómo es esto posible? La respuesta es QUIC, que veremos posteriormente en este capítulo. Lo interesante en este caso es darnos cuenta de que podemos tener una comunicación confiable (QUIC) a través de un medio no confiable (UDP). Para ello, el protocolo QUIC añade una nueva capa (encapsular) con la información y lógica necesaria para garantizar la confiabilidad en ambos extremos. Otra forma de verlo es que a veces podemos movernos entre TCP y UDP añadiendo los requisitos que necesitemos a UDP, que es el protocolo más básico, y evitar algunas de las desventajas de TCP.

## 5.2. Arquitecturas de Aplicaciones Distribuidas

Las arquitecturas en las aplicaciones distribuidas, es decir, con más de un nodo, indican cómo se conectan entre sí los nodos y cuál será el rol de cada uno de los nodos. A grandes rasgos, distinguimos tres tipos de arquitecturas: cliente - servidor, peer-to-peer e híbrida. La arquitectura cliente - servidor la mencionamos en el ejemplo anterior. En el caso de peer-to-peer, tenemos un conjunto de nodos que se conectan entre sí. La topología de las conexiones no tiene por qué ser un grafo completo, y puede variar a lo largo del tiempo. En este caso la funcionalidad está distribuida por los nodos. Un ejemplo de peer-to-peer es BitTorrent. Finalmente, las arquitecturas híbridas son una mezcla entre ambas, teniendo generalmente autoridades centrales que permiten mantener la red en funcionamiento, o determinadas funcionalidades. Las arquitecturas híbridas son más comunes que las puramente peer-to-peer.

En los siguientes apartados exploraremos estas tres arquitecturas, así como las aplicaciones populares y juegos para cada una de ellas.

### **5.2.1. Arquitectura Cliente/Servidor**

La arquitectura cliente-servidor es un modelo fundamental de computación distribuida donde múltiples clientes solicitan servicios, recursos o datos de un servidor centralizado. En este paradigma, el servidor actúa como el punto central de control y coordinación, mientras que los clientes consumen los servicios proporcionados. Esta arquitectura se caracteriza por tener un host siempre activo (el servidor) que atiende las peticiones de numerosos hosts clientes, los cuales pueden conectarse y desconectarse dinámicamente sin afectar el funcionamiento del sistema. Los clientes poseen direcciones IP dinámicas y no se comunican directamente entre sí, sino que toda la comunicación se canaliza a través del servidor.

En el funcionamiento típico de esta arquitectura, el cliente inicia la comunicación enviando una solicitud al servidor, especificando qué servicio o recurso necesita. El servidor procesa esta petición, accede a los datos o recursos necesarios, y envía una respuesta de vuelta al cliente. Este modelo permite la centralización de recursos, datos y lógica de negocio, facilitando el mantenimiento, la seguridad y la consistencia del sistema. El servidor debe tener una dirección IP fija y conocida para que los clientes puedan localizarlo, y típicamente opera de forma continua para estar disponible cuando los clientes lo necesiten.

Los requerimientos de infraestructura para sistemas cliente-servidor populares son considerables. Los servidores deben ser capaces de manejar múltiples conexiones simultáneas, procesar grandes volúmenes de datos y mantener alta disponibilidad. Esto frecuentemente requiere centros de datos con clusters de servidores, sistemas de balanceamiento de carga, redundancia y respaldo, así como conexiones de red de alto ancho de banda. Para aplicaciones con millones de usuarios, como las redes sociales o servicios de streaming, la infraestructura puede incluir múltiples centros de datos distribuidos geográficamente para optimizar la latencia y garantizar la disponibilidad del servicio.

Ejemplos cotidianos de arquitectura cliente-servidor incluyen aplicaciones web como Netflix, donde el cliente (navegador web o aplicación móvil) solicita contenido de vídeo al servidor, que almacena y transmite las películas y series. Spotify funciona de manera similar, donde los clientes solicitan canciones y playlists que están almacenadas en los servidores de la plataforma. Instagram representa otro caso típico donde los clientes suben fotos y vídeos a los servidores, y otros usuarios pueden solicitar y visualizar este contenido. Los servicios de correo electrónico como Gmail operan bajo este modelo, donde los servidores almacenan y gestionan los mensajes mientras los clientes acceden a ellos a través de aplicaciones web o móviles.

En el contexto de los videojuegos, la arquitectura cliente-servidor se ha convertido en el estándar para juegos multijugador masivos y competitivos. El servidor mantiene el estado autoritativo del juego, procesando todas las acciones de los jugadores y distribuyendo las actualizaciones correspondientes. Los clientes se encargan principalmente de la presentación visual, la captura de entrada del usuario y la comunicación con el servidor. Esta separación permite que el servidor tenga control total sobre la lógica del juego, previniendo trampas y garantizando la coherencia del estado del juego entre todos los participantes.

Ejemplos típicos de esta arquitectura incluyen juegos como World of Warcraft, donde miles de jugadores se conectan a servidores dedicados que mantienen mundos persistentes. Counter-Strike: Global Offensive utiliza servidores dedicados para partidas competitivas, asegurando que todas las acciones sean validadas centralmente. League of Legends emplea esta arquitectura para sus partidas clasificatorias, donde el servidor procesa todos los movimientos, ataques y habilidades de los campeones. Fortnite Battle Royale también implementa servidores dedicados para mantener la sincronización entre los 100 jugadores en cada partida.

Los juegos de estrategia en tiempo real como StarCraft II y Age of Empires IV también adoptan esta arquitectura para sus modos multijugador competitivos. En estos casos, el servidor procesa todas las órdenes de construcción, movimiento de unidades y combates, garantizando que ambos jugadores vean exactamente el mismo estado del juego. Los MMORPGs como Final Fantasy XIV y Guild Wars 2 son ejemplos perfectos donde el servidor no solo mantiene el estado del juego sino también la persistencia de los personajes, inventarios y progreso de los jugadores.

Uno de los principales problemas en juegos cliente-servidor es la latencia o “lag”, que se refiere al tiempo que tarda una acción del jugador en ser procesada por el servidor y reflejada de vuelta al cliente. Esta latencia puede causar una experiencia de juego frustrante, especialmente en juegos de acción rápida como shooters en primera persona. Para mitigar este problema, muchos juegos implementan técnicas como la predicción del lado del cliente, donde el cliente asume temporalmente el resultado de una acción antes de recibir la confirmación del servidor.

El problema de la sincronización es otro desafío crítico en los juegos cliente-servidor. Cuando múltiples jugadores interactúan simultáneamente, el servidor debe procesar las acciones en un orden específico y comunicar los resultados a todos los clientes de manera coherente. Los juegos como Rocket League han tenido que implementar sistemas sofisticados de interpolación y extrapolación para mantener la fluidez del juego mientras se sincronizan las posiciones de la pelota y los vehículos entre todos los jugadores.

Los servidores sobrecargados representan un problema significativo, especialmente durante los lanzamientos de juegos populares o eventos especiales. Diablo III experimentó problemas masivos en su lanzamiento debido a que sus servidores no podían manejar la cantidad de jugadores conectados simultáneamente. World of Warcraft ha enfrentado desafíos similares durante las expansiones, donde millones de jugadores intentan conectarse al mismo tiempo, causando colas de conexión y caídas del servidor.

La pérdida de conexión con el servidor es otro problema común que puede arruinar la experiencia de juego. En juegos competitivos como Dota 2 o Overwatch, una desconexión del servidor puede resultar en penalizaciones para el jugador, incluso si la falta no fue suya. Los desarrolladores han implementado sistemas de reconexión automática y buffers de tolerancia para minimizar el impacto de desconexiones temporales, pero el problema persiste como una limitación inherente del modelo cliente-servidor.

Los costes de infraestructura representan un desafío económico significativo para los desarrolladores de juegos que adoptan esta arquitectura. Mantener granjas de servidores, centros de datos distribuidos globalmente y el ancho de banda necesario para soportar millones de jugadores

concurrentes requiere inversiones masivas. Epic Games, por ejemplo, ha invertido cientos de millones de dólares en infraestructura para soportar Fortnite, incluyendo partnerships con proveedores de servicios en la nube como Amazon Web Services para escalar dinámicamente según la demanda.

A pesar de estos desafíos, la arquitectura cliente-servidor sigue siendo la opción preferida para juegos multijugador serios debido a sus ventajas en términos de seguridad, control y escalabilidad. Los avances en tecnologías de red, computación en la nube y técnicas de optimización continúan mejorando la viabilidad de esta arquitectura. Los desarrolladores modernos implementan soluciones híbridas que combinan servidores dedicados con técnicas de peer-to-peer para diferentes aspectos del juego, optimizando tanto la experiencia del jugador como los costes operativos.

### **5.2.2. Arquitectura Peer-to-Peer (P2P)**

La arquitectura peer-to-peer (P2P) es un modelo de computación distribuida donde los participantes (pares o peers) comparten recursos directamente entre sí sin depender de servidores centralizados. A diferencia del modelo cliente-servidor, en P2P no existe una entidad central que controle o coordine las comunicaciones; en su lugar, cada participante actúa simultáneamente como cliente y servidor, compartiendo y consumiendo recursos de manera equitativa. Esta arquitectura se caracteriza por la ausencia de dependencia de servidores siempre activos, permitiendo que los pares se conecten de forma intermitente y estableciendo comunicación directa entre ellos.

El funcionamiento de las redes P2P se basa en la colaboración voluntaria de los participantes, donde cada peer contribuye con recursos computacionales, de almacenamiento o ancho de banda al conjunto de la red. Los peers pueden unirse o abandonar la red libremente sin comprometer significativamente su funcionamiento, ya que la arquitectura es inherentemente autoescalable: cuantos más participantes se unen, más recursos totales están disponibles. Esta característica contrasta marcadamente con los sistemas centralizados, donde el servidor puede convertirse en un cuello de botella cuando aumenta el número de usuarios.

Existen diferentes clasificaciones de arquitecturas P2P según su nivel de pureza y estructura:

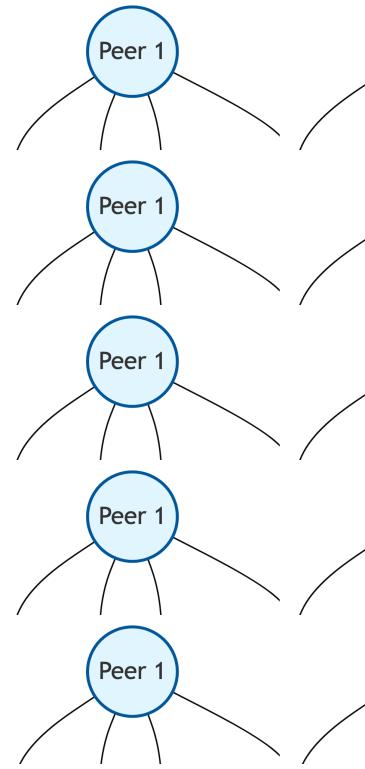
- Por pureza, encontramos sistemas centralizados como Napster (que dependía de un servidor central para indexar archivos) y BitTorrent (que utiliza trackers centrales para coordinar descargas), versus sistemas completamente descentralizados como Freenet y Gnutella, que no dependen de ningún equipo específico para su funcionamiento.
- Por paridad, las redes pueden ser estructuradas, donde existen categorías específicas de nodos con control sobre la estructura de la red, o desestructuradas, donde las conexiones y la topología emergen de manera arbitraria según las decisiones individuales de cada peer.

También existen diferentes tipos de topologías:

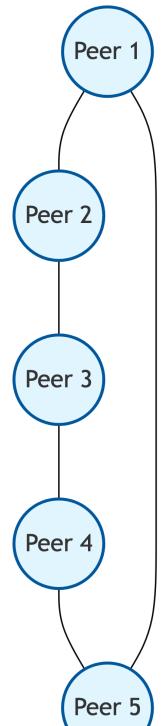
- La topología *Full Mesh* es la más robusta pero también la más demandante en términos de recursos, ya que cada peer se conecta directamente con todos los demás participantes de la red. Esta configuración ofrece la máxima redundancia y la latencia más baja posible entre cualquier par de nodos, pero el número de conexiones crece exponencialmente con cada nuevo participante, haciendo que sea práctica solo para grupos muy pequeños de peers.
- La topología *Ring* organiza los peers en una estructura circular donde cada nodo se conecta únicamente con sus vecinos inmediatos, formando un anillo cerrado. Los datos viajan alrededor del anillo hasta llegar a su destino, lo que puede introducir latencia variable dependiendo de la distancia entre peers en la estructura circular. Esta topología es más eficiente en términos de conexiones que el full mesh, pero presenta vulnerabilidades ya que la falla de un solo peer puede interrumpir la comunicación en todo el anillo, aunque existen implementaciones bidireccionales que mitigan este riesgo.
- La topología *Star* representa un enfoque pseudo-P2P donde un peer central actúa como hub para todos los demás participantes. Aunque técnicamente sigue siendo P2P porque no requiere un servidor dedicado, esta configuración introduce un punto único de falla en el peer central. Sin embargo, es la más eficiente en términos de gestión de conexiones y sincronización, ya que reduce significativamente la complejidad de coordinación. Es común en juegos cooperativos donde el host del juego actúa como el nodo central, gestionando el estado del juego y redistribuyendo información a los otros jugadores.
- Las topologías *Híbridas* combinan elementos de diferentes enfoques según los requerimientos específicos del juego o aplicación. Por ejemplo, un juego podría usar una topología de star para la lógica principal del juego mientras implementa conexiones mesh directas para comunicación de voz entre jugadores. Estas implementaciones permiten optimizar diferentes aspectos del rendimiento, balanceando latencia, confiabilidad y eficiencia de recursos según las necesidades particulares de cada función dentro del sistema P2P.

Las aplicaciones cotidianas de P2P incluyen sistemas de compartición de archivos como BitTorrent, donde los usuarios descargan fragmentos de archivos desde múltiples peers simultáneamente, distribuyendo la carga y mejorando la velocidad de descarga. Skype utilizó originalmente arquitectura P2P (similar a la arquitectura híbrida en (**hybrid-topology?**)) para enrutar llamadas de voz a través de la red de usuarios, aprovechando el ancho de banda y poder computacional distribuido. Las criptomonedas como Bitcoin operan sobre redes P2P completamente descentralizadas, donde cada nodo mantiene una copia del blockchain y participa en la validación de transacciones. Los sistemas de mensajería como Tox y Briar implementan comunicación P2P directa para garantizar privacidad y resistencia a la censura.

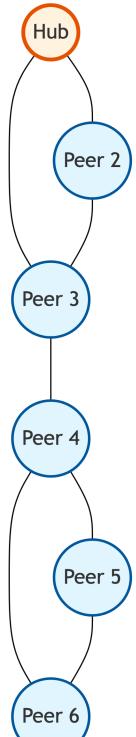
Las redes de distribución de contenido P2P como IPFS (InterPlanetary File System) permiten almacenar y distribuir información de manera descentralizada, donde cada participante contribuye espacio de almacenamiento y ancho de banda. Los juegos masivos como algunos



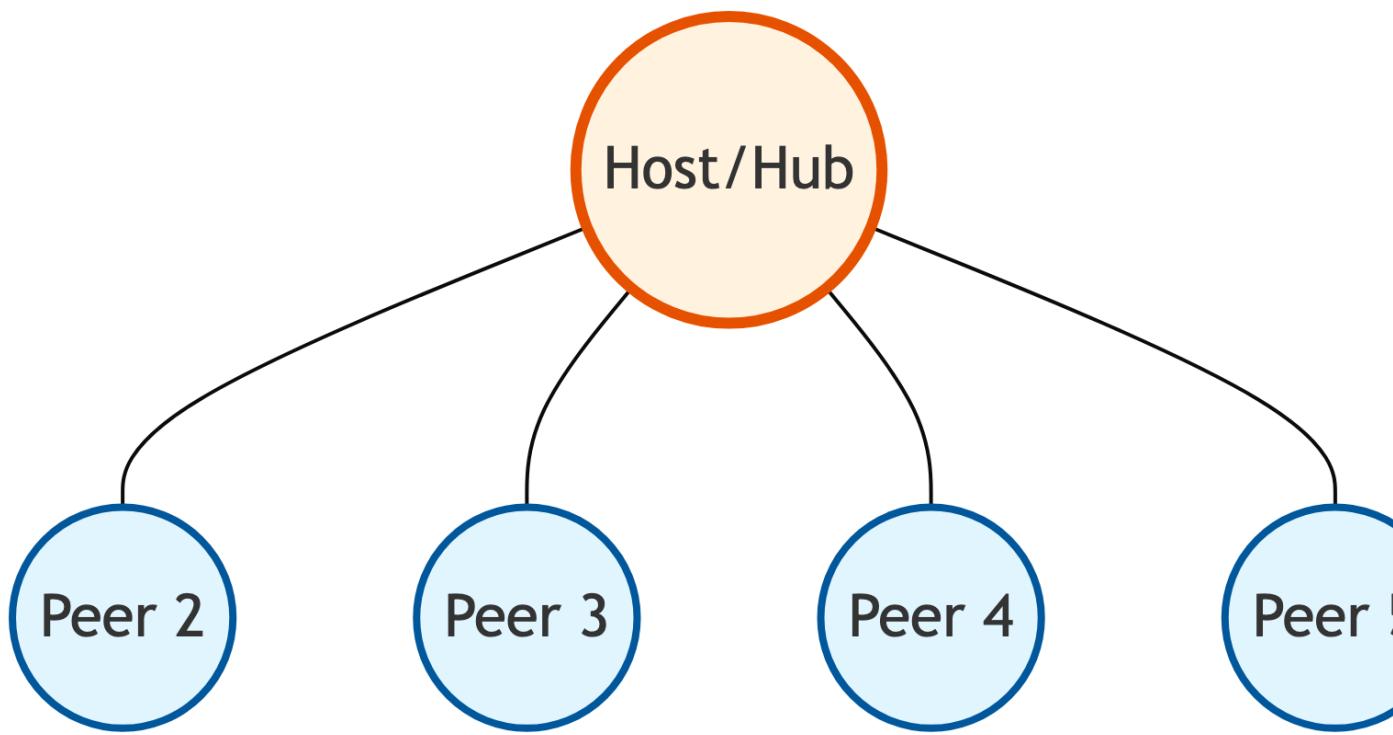
Full Mesh Topology



Ring Topology



Hybrid Topology



Star Topology (Pseudo-P2P)

servidores privados de World of Warcraft han experimentado con arquitecturas P2P para distribuir actualizaciones y contenido. Las aplicaciones de videoconferencia como Jitsi Meet pueden operar en modo P2P para llamadas pequeñas, estableciendo conexiones directas entre participantes para reducir latencia y eliminar la dependencia de servidores centrales.

En el contexto de los videojuegos, la arquitectura P2P ofrece ventajas únicas pero también presenta desafíos específicos. Los juegos P2P eliminan la necesidad de servidores dedicados, reduciendo costos operativos y permitiendo que los jugadores continúen partidas incluso si los servidores oficiales están fuera de línea. Esta arquitectura es especialmente efectiva en juegos con pocos participantes donde la latencia directa entre jugadores puede ser menor que la latencia a un servidor centralizado. Cada peer mantiene su propia copia del estado del juego y sincroniza cambios con otros participantes.

Los juegos de lucha como Street Fighter 6, Tekken 8 y Guilty Gear Strive utilizan arquitecturas P2P sofisticadas con tecnología de rollback netcode. En estos juegos, ambos jugadores mantienen una simulación completa del combate y sincronizan entradas periódicamente. Cuando hay discrepancias debido a latencia, el sistema “retrocede” el estado del juego y lo recalcula con la información correcta, creando una experiencia fluida incluso con conexiones imperfectas. Esta implementación es ideal para juegos 1v1 donde la latencia directa entre jugadores suele ser menor que la latencia a un servidor dedicado.

Los juegos cooperativos como Portal 2, It Takes Two y A Way Out aprovechan las ventajas de P2P para ofrecer experiencias de baja latencia entre un pequeño grupo de jugadores. En estos casos, uno de los peers actúa como “host” manteniendo el estado autoritativo del juego mientras otros se conectan directamente. Esta configuración elimina la necesidad de servidores dedicados para experiencias cooperativas, permitiendo que los desarrolladores ofrezcan funcionalidad multijugador sin costos adicionales de infraestructura. Los juegos de estrategia en tiempo real como Age of Empires II y StarCraft: Brood War originalmente utilizaban P2P, donde todos los jugadores ejecutaban la misma simulación y compartían comandos.

Sin embargo, los juegos P2P enfrentan desafíos significativos en términos de seguridad y prevención de trampas. Dado que cada peer tiene acceso completo al estado del juego, es relativamente fácil para usuarios malintencionados modificar datos o implementar cheats. Los juegos como Dark Souls han experimentado problemas con hackers que pueden modificar estadísticas de personajes o comportamientos del juego. La validación distribuida es compleja y requiere que múltiples peers acuerden sobre la validez de las acciones, lo que puede ser problemático cuando uno de los participantes está haciendo trampa.

La sincronización representa otro desafío mayor en juegos P2P, especialmente cuando el número de participantes aumenta. En juegos con muchos jugadores, cada peer debe comunicarse con todos los demás, creando un crecimiento cuadrático en el tráfico de red. Minecraft multijugador en modo LAN ejemplifica este problema: funciona bien para grupos pequeños pero se vuelve inmanejable con muchos jugadores. Los problemas de conectividad NAT también complican las conexiones P2P, ya que muchos jugadores están detrás de routers y firewalls que impiden

conexiones directas, requiriendo técnicas como hole punching o servidores de relay para establecer comunicación entre peers.

## 5.3. Protocolos

### 5.3.1. HTTP

HTTP (HyperText Transfer Protocol) es un protocolo público definido en un RFC que sirve para la transferencia de información en la World Wide Web. Es un protocolo de comunicación que permite la transferencia de recursos (como páginas web, imágenes, documentos, etc.) entre clientes (navegadores web) y servidores web a través de Internet. El protocolo utiliza texto legible tanto para los comandos como para las respuestas. El protocolo opera típicamente sobre TCP/IP, utilizando el puerto 80 para conexiones HTTP estándar y el puerto 443 para conexiones HTTPS seguras.

HTTP opera bajo el modelo **cliente-servidor**, donde los navegadores web (u otros programas) actúan como clientes que solicitan recursos, y los servidores web responden proporcionando el contenido solicitado. Esta arquitectura descentralizada permite que la web sea escalable y resiliente, distribuyendo la carga de trabajo entre diferentes servidores. Además, al ser un protocolo **sin estado**, se facilita su escalabilidad. Que no tenga estado implica que cada vez que se realiza una petición es completamente independiente de las anteriores.

Cada recurso en el servidor se identifica a través de una URL (Uniform Resource Locator), que especifica no solo la ubicación del recurso sino también el protocolo necesario para acceder a él. Una URL típica como “<https://www.ejemplo.com/pagina.html>” contiene el protocolo (https), el nombre del host (www.ejemplo.com), y la ruta específica del recurso (/pagina.html). Esta estructura jerárquica permite organizar y localizar millones de recursos de manera eficiente. Las URL pueden referenciar archivos HTML, hojas de estilo CSS, código, binarios, etc.

Las acciones en HTTP están asociadas a un verbo que indica el objeto de las mismas. Los principales son los siguientes:

- GET: Pedir el objeto de la URL al servidor. Es una operación idempotente, si la repetimos varias veces el resultado debería de ser siempre el mismo. No cambia el estado del servidor. El cuerpo del mensaje está vacío. Cuando descargamos imágenes en Instagram o similares, los comentarios, etc lo hacemos a través de GET.
- POST: Se utiliza para pedir/enviar un objeto asociado a una URL cuando este depende de los datos de un formulario. Puede cambiar el estado del servidor. Por ejemplo, cuando nos registramos en una página estaríamos haciendo un POST.
- HEAD: Es igual que el GET pero no devuelve nada. Se utiliza para debuguear.
- PUT: Nos permite cargar un objeto en la URL. Es una operación idempotente, si la repetimos varias veces el resultado será siempre el mismo.
- DELETE: Borra el recurso asociado a la URL.

Cabe destacar que este uso esperado de los verbos lo tenemos que implementar nosotros. Nada nos quita de hacer que un GET borre cosas, o se utilice para acciones para las que no estaba diseñado. Sin embargo, seguir la especificación nos va a permitir que otros usuarios de nuestra API la puedan utilizar correctamente de una forma más sencilla.

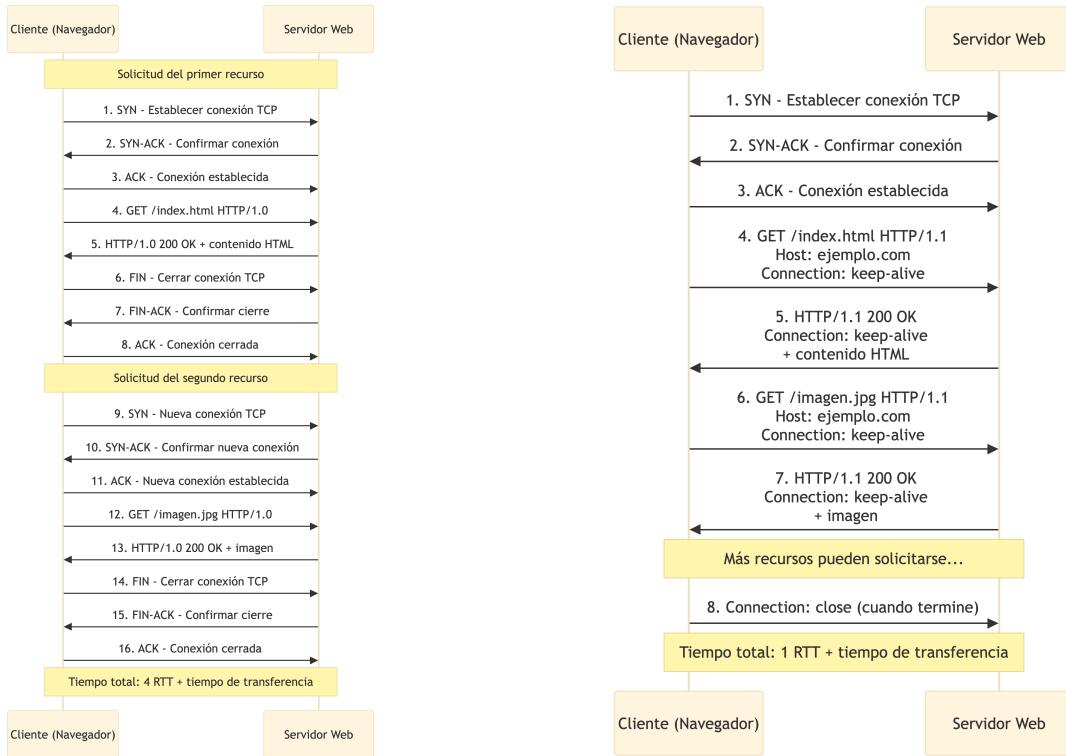
Todas estas acciones, que en la jerga de HTML se llaman peticiones, tienen asociada una respuesta. Esta respuesta está formada por un **código de respuesta**, el **cuerpo**, y **cookies**. Los códigos de respuesta son un identificador numérico de 3 cifras que indica el resultado de la petición y están asociados a un identificador textual. Se dividen en 5 grupos:

- **1XX**: Respuesta informativa, señalan que la solicitud está siendo procesada.
- **2XX**: Respuesta satisfactoria, la solicitud se recibió, entendió y se completó con éxito. Por ejemplo, 200 OK.
- **3XX**: Redirecciones, informan que se necesita tomar una acción adicional para completar la solicitud. Por ejemplo, 301 Moved Permanently: Indica que el recurso se ha movido de forma permanente a una nueva URL.
- **4XX**: Error en los clientes, indican un error en la solicitud del cliente, como solicitar un recurso inexistente. Por ejemplo, 400 Bad Request, 403 Forbidden o 404 Not Found.
- **5XX**: Error en los servidores, señalan que el servidor no pudo completar una solicitud debido a un error interno. Por ejemplo, 500 Internal Server Error o 503 Service Unavailable.

Dependiendo de la versión de HTTP se utilizarán diferentes tipos de conexión para enviar las peticiones. En HTTP/1.0, se utilizaban conexiones **no persistentes**, y para cada recurso se creaba una nueva conexión, incurriendo en un retardo de 2 RTT por objeto y la sobrecarga de abrir y cerrar conexiones. A partir de HTTP/1.1, se utilizan conexiones persistentes, donde varios objetos pueden ser enviados en la misma conexión, y por lo tanto, teniendo un retardo de 1 RTT por objeto. La limitación que tenía HTTP/1.1, es que si uno de los recursos tardaba mucho, ralentizaba a los que iban detrás. Para solucionar este problema se utilizan múltiples streams independientes sobre una conexión, solucionando el problema de que un recurso bloquee a los posteriores.

Por último, tiene un mecanismo adicional, las cookies que permiten guardar información en forma de pares de clave valor en el cliente. Las cookies se pueden configurar utilizando el campo de respuesta de la petición. En general se utilizan para mantener sesiones, personalización, análisis o con fines publicitarios. Estas cookies pueden ser propias, cuando es de la web que estamos navegando, o de terceros, cuando es un servicio que utiliza la web. Además del par de clave valor, también incluyen una fecha de expiración y del dominio del servidor. Las cookies expiran cuando pasa la fecha de expiración, aunque también pueden ser permanentes. El dominio es por seguridad, ya que determinadas cookies sólo queremos que sean accedidas por su dominio, con el fin de evitar suplantaciones de identidad.

Existe una variante de HTTP denominada HTTPS (Secure HyperText Transfer Protocol) en la cual las peticiones y sus respuestas no van en texto plano y se ha convertido en el estándar de la Web. De hecho algunos navegadores ya no dejan acceder a sitios a través de HTTP.



El protocolo opera generalmente sobre TCP, pero a partir de HTTP/3 opera sobre **QUIC**, que es un protocolo que implementa mecanismos de comunicación fiables sobre **UDP**. HTTP/3 está soportado por la gran mayoría de los navegadores actuales, y el soporte en los servidores está creciendo.

En determinadas situaciones para disminuir el tiempo de las peticiones se utilizan **servidores proxy**. Los servidores proxy son unos intermediarios, que analizan las peticiones, si pueden resolverlas ellos contestan directamente, y si no contestan a través de la petición al servidor. Las ventajas es que se obtiene una navegación más rápida, se reduce el tráfico, y además ganamos seguridad y anonimato. Suelen estar localizados en los navegadores (caché local), ISP o CDNs. En concreto, los servidores proxy cachearán las peticiones GET, ya que es una operación idempotente, y utilizan la herramienta del GET condicional donde en caso de que no haya actualización no devuelva nada, ahorrando el tiempo de envío del recurso.

### 5.3.2. DNS

DNS (Domain Name System) es uno de los protocolos más importantes de Internet. El objetivo de DNS es simple, traducir identificadores textuales que sean fácil de recordar por humanos a direcciones IP. Por ejemplo, traducir “www.google.es” a 142.250.200.67. El sistema de DNS

está diseñado como un sistema distribuido sin servidores centrales, lo que le permite distribuir la carga entre diferentes nodos y ser tolerante a fallos.

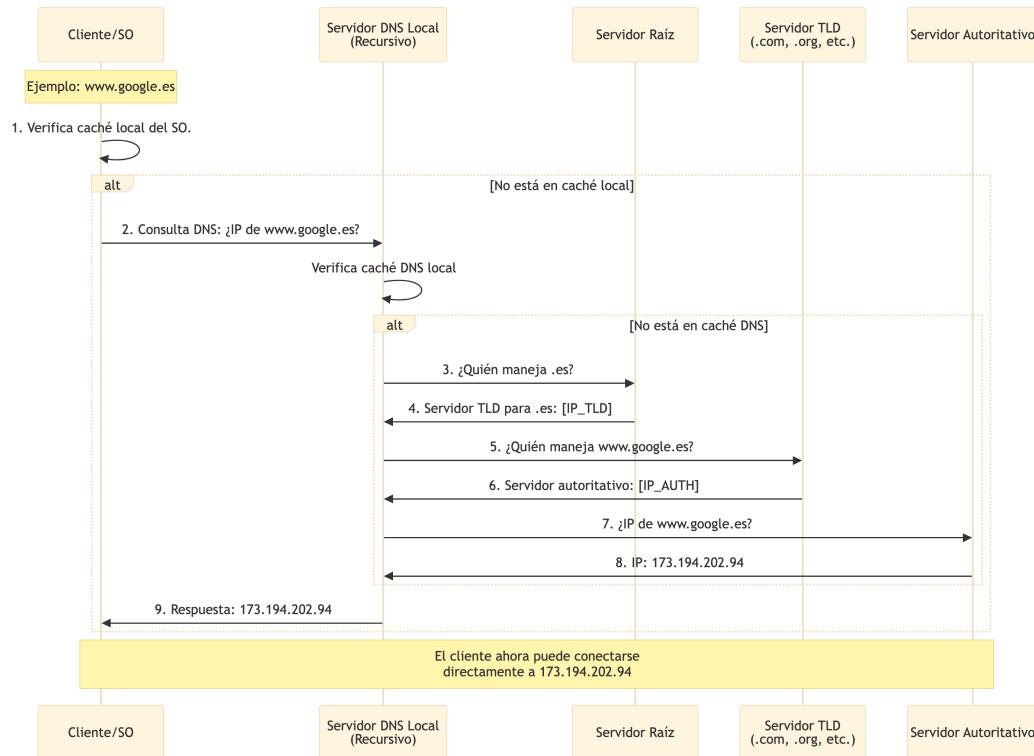
El sistema distribuido de DNS está formado por una estructura jerárquica de 4 tipos de nodos:

- **Servidores raíz:** Son las raíces de la jerarquía DNS y representan el nivel más alto del sistema. Existen 13 servidores raíz lógicos identificados con letras de la A a la M (a.root-servers.net hasta m.root-servers.net), aunque físicamente hay cientos de servidores distribuidos. Estos servidores conocen la ubicación de todos los servidores TLD y responden a consultas sobre dónde encontrar información de dominios de nivel superior.
- **Servidores TLD (Top Domain Level):** Son responsables de los dominios de nivel superior como .com, .org, .net, .edu, y los dominios de país como .es, .mx, .ar. Mantienen información sobre qué servidores autoritativos son responsables de cada dominio específico dentro de su TLD.
- **Servidores autoritativos:** Contienen la información definitiva y oficial sobre un dominio específico. Son los que tienen la autoridad final sobre las zonas DNS que administran y proporcionan las respuestas definitivas sobre las direcciones IP de los hosts dentro de su dominio.
- **Servidores locales:** También llamados servidores recursivos o resolvers, son los que reciben las consultas directamente de los clientes (como tu computadora). Se encargan de realizar todo el proceso de resolución consultando a los diferentes niveles de la jerarquía DNS hasta obtener la respuesta final, que luego envían de vuelta al cliente. Suelen mantener una caché para mejorar la eficiencia.

Para entender el proceso vamos a realizar un ejemplo de cómo sería la consulta para resolver la URL www.google.es a una IP con DNS. El diagrama de secuencia lo podéis ver en la (**DNS-GOOGLE?**). Los pasos para la resolución del DNS son los siguientes:

1. **Verificación de caché local del sistema operativo:** Cuando escribes una URL en tu navegador, el sistema operativo primero verifica su caché local para ver si ya tiene almacenada la dirección IP correspondiente. Si la encuentra y no ha expirado, la utiliza inmediatamente sin necesidad de hacer consultas externas.
2. **Consulta al servidor DNS local:** Si la información no está en caché o ha expirado, el cliente envía una consulta al servidor DNS configurado (generalmente proporcionado por tu ISP o servicios como 8.8.8.8 de Google). Esta consulta es recursiva, lo que significa que el cliente espera una respuesta completa.
3. **El servidor DNS local consulta al servidor raíz:** El servidor DNS local, al no tener la información solicitada, inicia el proceso de resolución consultando a uno de los 13 servidores raíz. Le pregunta: “¿Quién maneja el dominio de nivel superior de este nombre?”
4. **Respuesta del servidor raíz:** El servidor raíz no conoce la dirección IP específica, pero sí sabe qué servidor TLD maneja ese tipo de dominio (.com, .org, .es, etc.). Responde con la dirección del servidor TLD apropiado.

5. **Consulta al servidor TLD:** El servidor DNS local ahora consulta al servidor TLD correspondiente preguntando: "¿Qué servidor autoritativo maneja este dominio específico?"
6. **Respuesta del servidor TLD:** El servidor TLD responde con la información del servidor autoritativo responsable del dominio consultado. Por ejemplo, si buscas www.ejemplo.com, te dirá cuál es el servidor autoritativo para ejemplo.com.
7. **Consulta al servidor autoritativo:** Finalmente, el servidor DNS local consulta al servidor autoritativo del dominio, que tiene la información definitiva sobre todos los registros de ese dominio.
8. **Respuesta del servidor autoritativo:** El servidor autoritativo responde con la dirección IP correspondiente al nombre solicitado (registro A) o la información solicitada según el tipo de consulta.
9. **Respuesta final al cliente:** El servidor DNS local almacena la respuesta en su caché (con un tiempo de vida o TTL específico) y envía la dirección IP al cliente que originó la consulta.



### 5.3.3. SMTP, IMAP y POP

Los protocolos SMTP, IMAP y POP son protocolos que definen el funcionamiento del correo electrónico tal y como lo conocemos hoy en día. Cada uno tiene un propósito específico en el proceso de envío, almacenamiento y recuperación de mensajes.

**SMTP** es el protocolo estándar para el **envío** de correos electrónicos a través de Internet. Funciona como un servicio de entrega que transporta mensajes desde el cliente de correo del remitente hasta el servidor de correo del destinatario. Es un protocolo “push”, empuja los mensajes desde el origen hacia el destino, y no maneja la recepción de los correos.

**POP**, especialmente POP3 (la versión más actual), es un protocolo para **descargar** correos electrónicos desde el servidor al dispositivo local. POP descarga los mensajes completos al dispositivo local, y por defecto, los elimina los mensajes del servidor tras la descarga. Es ideal para usuarios que acceden al correo desde un único dispositivo, pero presenta limitaciones para sincronización entre múltiples dispositivos

**IMAP** es un protocolo más moderno que permite **acceder** a los correos electrónicos manteniendo la sincronización entre el servidor y múltiples clientes. Los mensajes permanecen en el servidor y permite sincronización en tiempo real entre dispositivos. Soporta carpetas, etiquetas y búsquedas en el servidor. Es ideal para usuarios que acceden al correo desde múltiples dispositivos

#### 5.3.4. QUIC

QUIC representa una evolución revolucionaria en los protocolos de transporte de Internet, desarrollado inicialmente por Google en 2012 y estandarizado por la IETF en 2021 como RFC 9000. Este protocolo moderno construido sobre UDP combina las mejores características de TCP con la seguridad integrada de TLS 1.3, eliminando muchas de las limitaciones históricas de los protocolos tradicionales. Sus principales ventajas incluyen el multiplexado nativo de streams sin el problema de head-of-line blocking que afecta a HTTP/2 sobre TCP, el establecimiento de conexiones con latencia cero (0-RTT) para reconexiones, y la capacidad única de migración de conexión que permite a los dispositivos cambiar transparentemente entre redes WiFi y móviles sin interrumpir las sesiones activas. Además, QUIC incorpora algoritmos de control de congestión más sofisticados y mecanismos de corrección de errores (Forward Error Correction) que mejoran significativamente el rendimiento en condiciones de red inestables o con alta pérdida de paquetes.

Los casos de uso de QUIC son especialmente relevantes en aplicaciones que requieren baja latencia y alta confiabilidad, siendo adoptado masivamente por servicios de streaming, aplicaciones de videoconferencia, juegos en línea, y plataformas de contenido como YouTube, donde Google reporta reducciones de hasta 30 % en tiempo de carga. Su adopción en 2025 ha alcanzado cifras impresionantes: el 8.2 % de todos los sitios web globalmente utilizan QUIC, mientras que HTTP/3 (que funciona exclusivamente sobre QUIC) es empleado por el 31.1 % de los sitios web.

## **5.4. Servicios**

### **5.4.1. CDNs**

Las CDN funcionan mediante una red distribuida de servidores edge ubicados estratégicamente en diferentes regiones geográficas, que almacenan copias del contenido desde los servidores origen para reducir la distancia física que deben recorrer los datos hasta llegar al usuario final. El sistema utiliza enrutamiento inteligente que automáticamente dirige cada solicitud al servidor más cercano disponible, típicamente reduciendo la latencia de carga de 200-500ms a menos de 50ms. La estrategia de caché varía según el tipo de contenido: archivos estáticos como imágenes, videos y assets de aplicaciones se almacenan por períodos prolongados (días o semanas), mientras que contenido dinámico como respuestas de APIs se cachea por minutos u horas con validación frecuente. Para contenido personalizado, las CDN implementan técnicas de caché parcial donde elementos comunes se reutilizan entre usuarios, y para streaming en tiempo real dividen el contenido en pequeños segmentos que pueden cachearse individualmente.

Más allá de la simple entrega de contenido, las CDN modernas actúan como una capa de protección y optimización que incluye compresión automática de archivos, conversión de formatos de imagen según el dispositivo del usuario, y balanceado de carga inteligente que redistribuye el tráfico cuando algún servidor se sobrecarga. En aplicaciones como videojuegos, las CDN aceleran la descarga de actualizaciones y assets mediante técnicas de pre-carga predictiva, mientras que para aplicaciones web ejecutan código simple directamente en los servidores edge para personalización básica sin necesidad de consultar el servidor origen. La arquitectura distribuida proporciona resistencia natural contra caídas de servicio y ataques DDoS, ya que el tráfico malicioso se dispersa automáticamente entre múltiples ubicaciones, y sistemas de monitoreo en tiempo real pueden redirigir usuarios desde servidores con problemas hacia alternativas saludables, manteniendo la disponibilidad del servicio incluso durante fallas regionales o ataques coordinados

## **Parte II**

# **Desarrollo en el cliente**

# 6 JavaScript para Desarrollo de Videojuegos

## 6.1. Introducción

JavaScript es un lenguaje de programación que permite incorporar interactividad en las páginas web, lo que lo convierte en una herramienta fundamental para el desarrollo de videojuegos web. Con JavaScript se puede modificar la página y ejecutar código cuando se interactúa con ella a través del modelo de objetos del documento (DOM). También se pueden hacer peticiones al servidor web en segundo plano y actualizar el contenido de la web con los resultados (AJAX).

### 6.1.1. Características de JavaScript

JavaScript es un lenguaje de programación basado en el estándar ECMAScript de ECMA (una organización diferente al W3C). Aunque en el pasado existían diferencias significativas en la implementación de JavaScript entre navegadores, actualmente todos son bastante compatibles entre sí.

### 6.1.2. Versiones de ECMAScript

**ES5 (2011):** La versión del estándar que popularizó el lenguaje ECMAScript fue la 5.1 (2011), aunque generalmente se la conoce como ES5. Prácticamente todos los navegadores modernos soportan la mayoría de las características definidas en el estándar 5.1.

**ES2015 (ES6):** En junio 2015 finalizó el desarrollo de ES6 con una evolución importante del lenguaje. A última hora decidieron llamarle oficialmente ES2015. Está soportada casi al completo por casi todos los navegadores modernos. Introdujo características fundamentales como clases, módulos, arrow functions y promises.

**Versiones actuales:** Desde ES2015, ECMAScript sigue un ciclo de actualizaciones anuales con compatibilidad hacia atrás: - ECMAScript 2016-2024 (versiones 7-15) - Cada versión añade nuevas características manteniendo compatibilidad - Los navegadores modernos soportan características hasta ES2023

### **6.1.3. JavaScript vs Java**

Aunque algunos elementos de la sintaxis recuerden a Java, son lenguajes completamente diferentes. El nombre JavaScript se eligió al publicar el lenguaje en una época en la que Java estaba en auge y fue principalmente por marketing (inicialmente se llamó LiveScript).

### **6.1.4. Características principales de JavaScript**

**Scripting:** No necesita compilador. Inicialmente era un lenguaje interpretado, pero actualmente se ejecuta en máquinas virtuales en los navegadores, proporcionando mayor velocidad de ejecución y eficiencia de memoria.

**Tipado dinámico:** Habitual en los lenguajes de script. Las variables no requieren declaración de tipo.

**Funcional:** Las funciones son elementos de primer orden, pueden asignarse a variables y pasarse como parámetros.

**Orientado a objetos:** Basado en prototipos, no en clases como Java, C++, Ruby, aunque ES2015 introdujo una sintaxis de clases más familiar.

### **6.1.5. DOM y BOM**

**DOM (Document Object Model):** Biblioteca (API) para manipular el documento HTML cargado en el navegador. Permite la gestión de eventos, insertar y eliminar elementos, etc.

**BOM (Browser Object Model):** Acceso a otros elementos del browser: historial, peticiones de red AJAX, etc. El BOM incluye al DOM como uno de sus elementos.

### **6.1.6. Librerías y Frameworks JavaScript**

Existen multitud de bibliotecas JavaScript para el desarrollo de aplicaciones de videojuegos:

**Bibliotecas de propósito general:** - **Lodash:** Biblioteca moderna que reemplaza a underscore.js para trabajar con estructuras de datos con un enfoque funcional - **Axios:** Cliente HTTP moderno que reemplaza las peticiones AJAX tradicionales

**Frameworks para videojuegos:** - **Phaser:** Framework completo para desarrollo de juegos 2D - **Three.js:** Biblioteca para gráficos 3D y WebGL - **Babylon.js:** Motor 3D completo para juegos web - **PixiJS:** Renderizador 2D de alta performance

**Frameworks de aplicación moderna:** - **React:** Biblioteca para interfaces de usuario componentizadas - **Vue.js:** Framework progresivo para aplicaciones web - **Angular:** Framework completo para aplicaciones de gran escala

## 6.2. Configuración del Entorno de Desarrollo con Node.js

### 6.2.1. Introducción a Node.js

Node.js es un entorno de ejecución para JavaScript construido sobre el motor V8 de Chrome. Permite ejecutar JavaScript fuera del navegador y se ha convertido en el estándar para el desarrollo de aplicaciones JavaScript modernas.

### 6.2.2. Instalación de Node.js

1. **Descargar Node.js:** Visita [nodejs.org](https://nodejs.org) y descarga la versión LTS (Long Term Support)
2. **Verificar instalación:** Abre una terminal y ejecuta:

```
node --version  
npm --version
```

### 6.2.3. Gestión de Paquetes con npm

**npm** (Node Package Manager) es el gestor de paquetes oficial de Node.js que permite instalar y gestionar dependencias.

```
# Verificar versión de npm  
npm --version  
  
# Actualizar npm  
npm install -g npm@latest  
  
# Obtener ayuda  
npm help
```

### 6.2.4. Creación de un Proyecto de Videojuego

#### 6.2.4.1. Inicializar un proyecto

```
# Crear directorio del proyecto  
mkdir mi-juego-web  
cd mi-juego-web
```

```
# Inicializar proyecto npm
npm init -y
```

Esto crea un archivo package.json que describe el proyecto:

```
{
  "name": "mi-juego-web",
  "version": "1.0.0",
  "description": "Un videojuego web desarrollado en JavaScript",
  "main": "src/index.js",
  "scripts": {
    "start": "node server.js",
    "dev": "webpack serve --mode development",
    "build": "webpack --mode production"
  },
  "keywords": ["juego", "javascript", "web"],
  "author": "Tu Nombre",
  "license": "MIT"
}
```

#### 6.2.4.2. Estructura de carpetas recomendada

```
mi-juego-web/
  package.json
  webpack.config.js
  src/
    index.js
    game/
      Game.js
      Player.js
      Enemy.js
    assets/
      images/
      sounds/
      fonts/
    styles/
      main.css
  dist/
  public/
    index.html
```

### **6.2.5. Instalación de Dependencias**

El sistema de gestión de paquetes de npm nos permite instalar bibliotecas y herramientas desarrolladas por la comunidad, evitando tener que escribir todo el código desde cero. En el desarrollo de videojuegos, esto es especialmente valioso porque podemos aprovechar engines de juegos, bibliotecas de física, sistemas de audio y muchas otras funcionalidades ya probadas y optimizadas.

Las dependencias se dividen en dos categorías principales: las de producción (que forman parte del juego final) y las de desarrollo (que solo usamos durante el proceso de creación).

#### **6.2.5.1. Dependencias de producción**

Estas son las bibliotecas que formarán parte de nuestro juego final y que los jugadores descargaráan:

```
# Framework de juegos
npm install Phaser

# Utilidades
npm install lodash

# Cliente HTTP
npm install axios
```

#### **6.2.5.2. Dependencias de desarrollo**

Las herramientas de desarrollo nos ayudan durante el proceso de creación del juego, pero no se incluyen en la versión final que descargan los jugadores. Estas incluyen herramientas para optimizar código, servir archivos durante el desarrollo, y verificar la calidad del código:

```
# Bundler y servidor de desarrollo
npm install --save-dev webpack webpack-cli webpack-dev-server

# Loaders para recursos
npm install --save-dev html-webpack-plugin css-loader style-loader file-loader

# Herramientas de desarrollo
npm install --save-dev eslint prettier
```

### 6.2.6. Configuración de Webpack

Webpack es una herramienta fundamental en el desarrollo moderno de JavaScript que actúa como empaquetador de módulos. Su función principal es tomar todos los archivos JavaScript, CSS, imágenes y otros recursos de nuestro proyecto y crear uno o varios archivos optimizados para el navegador.

En el contexto del desarrollo de videojuegos, Webpack nos permite organizar nuestro código en múltiples archivos (clases para jugadores, enemigos, sistemas de audio, etc.) y luego combinarlos en un solo paquete eficiente. También puede optimizar imágenes, procesar archivos de audio y gestionar otros recursos del juego.

Para configurar Webpack en nuestro proyecto de videojuego, creamos un archivo `webpack.config.js`:

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
    clean: true
  },
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader']
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif|ogg|mp3|wav)$/i,
        type: 'asset/resource'
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html'
    })
  ],
  devServer: {
```

```
        static: './dist',
        hot: true
    }
};
```

### 6.2.7. Scripts de Desarrollo

Una vez configurado Webpack, necesitamos definir comandos que automaticen las tareas más comunes del desarrollo. Los scripts de npm nos permiten crear atajos para ejecutar procesos complejos con comandos simples. Esto es especialmente útil cuando trabajamos en equipo, ya que todos los desarrolladores pueden usar los mismos comandos estándar.

Actualizamos el `package.json` con scripts útiles para el desarrollo de videojuegos:

```
{
  "scripts": {
    "dev": "webpack serve --mode development --open",
    "build": "webpack --mode production",
    "lint": "eslint src/",
    "format": "prettier --write src/"
  }
}
```

## 6.3. El Lenguaje JavaScript

### 6.3.1. Características del Lenguaje

La mayoría de las características que se van a repasar están disponibles desde JavaScript ES5, aunque las versiones más recientes ofrecen mejoras significativas. Las versiones más recientes pueden presentar problemas de compatibilidad con navegadores muy antiguos, pero esto no es una preocupación para el desarrollo moderno.

### 6.3.2. Imperativo y Estructurado

JavaScript mantiene las características imperativas familiares para programadores de Java y C:  
- Se declaran variables - Se ejecutan las sentencias en orden - Dispone de sentencias de control de flujo (if, while, for...) - La sintaxis imperativa/estructurada es muy parecida a Java y C

### **6.3.3. Lenguaje de Script**

- No existe compilador (aunque se usan transpiladores como Babel para compatibilidad)
- El navegador carga el código, lo analiza y lo ejecuta
- El navegador indica tanto errores de sintaxis como errores de ejecución
- Los motores modernos (V8, SpiderMonkey) incluyen compilación JIT para optimización

### **6.3.4. Tipado Dinámico**

- Al declarar una variable no se indica su tipo
- A lo largo de la ejecución del programa una misma variable puede tener valores de diferentes tipos
- Esto proporciona flexibilidad pero requiere más cuidado en el desarrollo

### **6.3.5. Orientado a Objetos**

- Todos los valores son objetos (no como en Java, que existen tipos primitivos)
- Existe recolector de basura para liberar la memoria automáticamente
- La orientación a objetos está basada en prototipos, aunque ES2015+ añadió sintaxis de clases
- En tiempo de ejecución se pueden crear objetos, cambiar atributos e invocar métodos
- En tiempo de ejecución se pueden añadir y borrar atributos y métodos dinámicamente

### **6.3.6. Funciones como Ciudadanos de Primera Clase**

- Aunque sea orientado a objetos, también permite declarar funciones independientes
- Las funciones se pueden declarar en cualquier sitio, asignarse a variables y pasarse como parámetro
- Existen funciones anónimas y arrow functions
- En JavaScript se puede implementar código siguiendo el paradigma funcional

### **6.3.7. Modo Estricto**

Las primeras versiones de JavaScript permitían escribir código que posteriormente se consideró propenso a errores. ES5 definió un modo estricto que genera errores para código problemático.

Para activar el modo estricto (recomendable) basta poner al principio del código:

```
"use strict";
```

En módulos ES2015+, el modo estricto está activado por defecto.

## 6.4. Integración con HTML

### 6.4.1. Inclusión de JavaScript

El código JavaScript se puede incluir directamente en el documento HTML en etiquetas `<script>`, pero es recomendable que el código esté en ficheros independientes:

```
<html>
<head>
    <!-- Para código que debe ejecutarse antes que el DOM -->
    <script src="js/config.js"></script>
</head>
<body>
    <!-- Contenido HTML -->

    <!-- Scripts al final para mejor rendimiento -->
    <script src="js/game.js"></script>
</body>
</html>
```

### 6.4.2. Optimización de Carga

**Ubicación de scripts:** Cuando se carga el JavaScript, el navegador bloquea el procesamiento del HTML. Por ello, se recomienda poner los elementos `<script>` como último elemento de la página.

Atributos modernos:

```
<!-- Carga asíncrona, no bloquea el HTML -->
<script src="js/game.js" async></script>

<!-- Carga diferida, ejecuta después del HTML -->
<script src="js/game.js" defer></script>
```

## 6.5. Sintaxis Básica

### 6.5.1. Mostrar Información

```
// Escribir en el documento HTML (poco usado en desarrollo moderno)
document.write('Texto');

// Escribir en la consola del navegador (debugging)
console.log('Información de debug');
console.error('Error crítico');
console.warn('Advertencia');
```

### 6.5.2. Comentarios

```
// Comentario de una línea

/*
 * Comentario
 * multilínea
 */
```

### 6.5.3. Delimitadores

- Bloques: {}
- Sentencias: ; (opcionales pero recomendados)

### 6.5.4. Tipos de Datos

JavaScript maneja diferentes tipos de datos que nos permiten representar toda la información necesaria para un videojuego. A diferencia de lenguajes como Java o C++, JavaScript es de tipado dinámico, lo que significa que una variable puede cambiar de tipo durante la ejecución del programa.

**Primitivos** (todos son objetos en JavaScript):

```
// Number - enteros y reales
let score = 1000;
let health = 75.5;

// String - cadenas de caracteres
let playerName = "Jugador1";
let message = 'Game Over';
```

```
// Boolean
let isGameRunning = true;
let isPaused = false;

// Tipos especiales
let powerUp = null;           // Ausencia intencional de valor
let specialAbility;          // undefined - no inicializada
```

#### Template literals (ES2015+):

Los template literals son una característica moderna de JavaScript que nos permite crear strings de manera más expresiva y legible. Son especialmente útiles en videojuegos para construir mensajes dinámicos, interfaces de usuario y contenido HTML generado dinámicamente:

```
let level = 5;
let experience = 1250;

// Interpolación de strings - insertar valores directamente
let status = `Nivel ${level} - EXP: ${experience}`;

// Strings multilínea - útil para templates HTML
let gameInfo = `
Jugador: ${playerName}
Nivel: ${level}
Puntuación: ${score}
`;
```

#### 6.5.5. Variables

En JavaScript moderno, tenemos tres formas de declarar variables, cada una con características específicas que las hacen adecuadas para diferentes situaciones. Esta evolución del lenguaje ha mejorado significativamente la robustez y mantenibilidad del código.

#### Declaraciones modernas (ES2015+):

```
// const - valor inmutable, block scope
const MAX_LIVES = 3;
const GAME_CONFIG = {
    width: 800,
    height: 600
};
```

```
// let - valor mutable, block scope
let currentLives = MAX_LIVES;
let playerPosition = { x: 0, y: 0 };

// var - evitar en código nuevo (function scope)
var oldStyle = "no recomendado";
```

### Ámbito de variables:

El concepto de ámbito (scope) es fundamental para entender cómo JavaScript maneja las variables. El ámbito determina desde qué partes del código podemos acceder a una variable:

- `let` y `const` tienen ámbito de bloque - solo existen dentro del bloque `{}` donde se declaran
- `var` tiene ámbito de función - existe en toda la función donde se declara (puede causar problemas)
- Variables no declaradas se convierten en globales - esto es un error en modo estricto

### 6.5.6. Operadores

JavaScript incluye la mayoría de operadores familiares para programadores de otros lenguajes, pero también añade algunos específicos que son importantes conocer para evitar errores comunes.

#### Similares a Java:

Los operadores aritméticos y lógicos funcionan de manera similar a otros lenguajes de programación:

```
// Aritméticos
let damage = baseDamage + bonus;
let remaining = total - used;
let area = width * height;
let average = sum / count;
let remainder = value % modulo;

// Comparación
if (health > 0 && mana >= spellCost) {
    castSpell();
}

// Lógicos
let canAct = isAlive && !isStunned;
let shouldRespawn = isDead || health <= 0;
```

### Específicos de JavaScript:

JavaScript tiene operadores de comparación únicos que es crucial entender para evitar errores sutiles pero problemáticos:

```
// Igualdad estricta (recomendado)
if (playerID === targetID) {
    // Compara valor y tipo
}

// Desigualdad estricta
if (level !== previousLevel) {
    // Diferentes valor o tipo
}

// Igualdad débil (evitar)
if (score == "100") {
    // true - hace conversión de tipos
}
```

### Operadores modernos (ES2020+):

Las versiones recientes de JavaScript han introducido operadores que simplifican patrones comunes de programación y hacen el código más legible y menos propenso a errores:

```
// Nullish coalescing - valor por defecto solo para null/undefined
let playerName = savedName ?? "Jugador Anónimo";

// Optional chaining - acceso seguro a propiedades
let weapon = player.inventory?.equipment?.weapon;

// Logical assignment
playerName ||= "Jugador por defecto"; // solo si falsy
playerName ??= "Valor por defecto"; // solo si null/undefined
```

## 6.6. Arrays

Los arrays en JavaScript son estructuras de datos fundamentales para el desarrollo de videojuegos, donde frecuentemente necesitamos manejar listas de enemigos, ítems del inventario, puntuaciones, coordenadas y muchos otros elementos. Aunque comparten similitudes con los arrays de Java, tienen características únicas que los hacen más flexibles pero también requieren mayor cuidado en su uso.

Los arrays de JavaScript son dinámicos por naturaleza, pueden contener elementos de diferentes tipos y crecen automáticamente cuando se necesita. Esta flexibilidad es especialmente útil en videojuegos donde las listas de elementos pueden cambiar constantemente durante el juego.

```
// Creación de arrays
let empty = [];
let numbers = [1, 2, 3, 4, 5];
let mixed = ["texto", 42, true, null];
let inventory = new Array(10); // Array de 10 elementos undefined

// Acceso y modificación
console.log(numbers[0]);           // 1
numbers[2] = 999;                  // Modifica el elemento
console.log(numbers.length);       // 5

// Arrays pueden crecer dinámicamente
numbers[10] = 100;                // Crea huecos con undefined
console.log(numbers.length);       // 11
```

### 6.6.1. Métodos de Array Modernos

Los métodos modernos de arrays en JavaScript son herramientas poderosas que permiten manipular datos de manera más expresiva y funcional. Estos métodos son especialmente útiles en videojuegos donde constantemente necesitamos filtrar enemigos, transformar datos, buscar elementos específicos o procesar listas de objetos del juego.

Estos métodos siguen un paradigma funcional: no modifican el array original (excepto algunos como `push` y `pop`), sino que retornan nuevos arrays o valores. Esto hace el código más predecible y menos propenso a errores:

```
let enemies = [
  { id: 1, health: 100, type: "orc" },
  { id: 2, health: 50, type: "goblin" },
  { id: 3, health: 0, type: "orc" }
];

// Filtrar elementos
let aliveEnemies = enemies.filter(enemy => enemy.health > 0);
let orcs = enemies.filter(enemy => enemy.type === "orc");

// Transformar elementos
let healthValues = enemies.map(enemy => enemy.health);
```

```

let enemyIds = enemies.map(enemy => enemy.id);

// Encontrar elementos
let firstOrc = enemies.find(enemy => enemy.type === "orc");
let orcIndex = enemies.findIndex(enemy => enemy.type === "orc");

// Verificar condiciones
let allDead = enemies.every(enemy => enemy.health === 0);
let someDead = enemies.some(enemy => enemy.health === 0);

// Reducir a un valor
let totalHealth = enemies.reduce((sum, enemy) => sum + enemy.health, 0);

// Modificar array
enemies.push({ id: 4, health: 75, type: "troll" }); // Añadir al final
let firstEnemy = enemies.shift(); // Quitar del inicio
let lastEnemy = enemies.pop(); // Quitar del final

// Eliminar/insertar elementos
enemies.splice(1, 2); // Eliminar 2 elementos desde índice 1
enemies.splice(1, 0, newEnemy); // Insertar en índice 1

```

### 6.6.2. Destructuring de Arrays (ES2015+)

El destructuring es una característica moderna que nos permite extraer valores de arrays de manera más concisa y legible. Es especialmente útil cuando trabajamos con coordenadas, vectores o cualquier conjunto de valores relacionados que se almacenan en arrays:

```

let coordinates = [100, 200];
let [x, y] = coordinates; // x = 100, y = 200

let [first, second, ...rest] = inventory; // Rest operator

```

## 6.7. Sentencias de Control de Flujo

Las sentencias de control de flujo en JavaScript son fundamentales para implementar la lógica de nuestros videojuegos. Nos permiten tomar decisiones, repetir acciones y controlar el comportamiento del juego basándose en diferentes condiciones como el estado del jugador, la fase del juego o las acciones del usuario.

### 6.7.1. Sentencias Básicas

JavaScript utiliza una sintaxis muy similar a Java y C para las estructuras de control, lo que facilita la transición desde otros lenguajes de programación:

```
// if - else
if (health > 50) {
    statusColor = "green";
} else if (health > 20) {
    statusColor = "yellow";
} else {
    statusColor = "red";
}
```

```
// switch
switch (gameState) {
    case "menu":
        showMenu();
        break;
    case "playing":
        updateGame();
        break;
    case "paused":
        showPauseScreen();
        break;
    default:
        handleUnknownState();
}
```

```
// Loops
for (let i = 0; i < enemies.length; i++) {
    updateEnemy(enemies[i]);
}

// for...of - iterar valores (ES2015+)
for (let enemy of enemies) {
    updateEnemy(enemy);
}

// for...in - iterar propiedades
for (let key in gameConfig) {
    console.log(key, gameConfig[key]);
```

```
}

// while
while (isGameRunning && playerLives > 0) {
    processGameFrame();
}
```

### 6.7.2. Valores Falsy

Un concepto importante en JavaScript es el de valores “falsy”. JavaScript es más permisivo que otros lenguajes al evaluar condiciones booleanas, y automáticamente convierte ciertos valores a `false` en contextos booleanos. Esto puede ser tanto útil como fuente de errores si no se comprende bien.

JavaScript considera falso: `false`, `null`, `undefined`, `""` (cadena vacía), `0`, `NaN`

```
// Verificación de existencia
if (player.weapon) {
    // weapon existe y no es falsy
    player.attack();
}

// Valores por defecto
let playerName = inputName || "Jugador Anónimo";
```

## 6.8. Funciones

Las funciones son uno de los conceptos más importantes en JavaScript, especialmente para el desarrollo de videojuegos donde necesitamos organizar el código en bloques reutilizables y manejables. JavaScript trata las funciones como “ciudadanos de primera clase”, lo que significa que pueden almacenarse en variables, pasarse como argumentos a otras funciones y retornarse como valores.

Esta flexibilidad es especialmente valiosa en videojuegos, donde frecuentemente necesitamos sistemas de callbacks para eventos, funciones que generen comportamientos aleatorios, o sistemas de actualización que procesen diferentes tipos de objetos del juego.

### 6.8.1. Declaración de Funciones

JavaScript ofrece varias formas de declarar funciones, cada una con sus propias características y casos de uso apropiados:

```
// Declaración tradicional
function calculateDamage(baseDamage, criticalHit) {
    if (criticalHit) {
        return baseDamage * 2;
    }
    return baseDamage;
}

// Expresión de función
let heal = function(amount) {
    player.health += amount;
    if (player.health > player.maxHealth) {
        player.health = player.maxHealth;
    }
};

// Arrow functions (ES2015+)
let movePlayer = (deltaX, deltaY) => {
    player.x += deltaX;
    player.y += deltaY;
};

// Arrow function con una expresión
let isAlive = (entity) => entity.health > 0;

// Arrow function sin parámetros
let generateRandomID = () => Math.random().toString(36);
```

### 6.8.2. Parámetros de Función

JavaScript es muy flexible en el manejo de parámetros de función, ofreciendo características modernas que simplifican el código y lo hacen más robusto. Esta flexibilidad es especialmente útil en videojuegos donde las funciones pueden necesitar comportamientos adaptativos según diferentes contextos de juego:

```

// Parámetros por defecto (ES2015+)
function createEnemy(health = 100, damage = 10) {
    return { health, damage };
}

// Rest parameters (ES2015+)
function logMessage(level, ...messages) {
    console.log(`[${level}]`, ...messages);
}

// Destructuring de parámetros
function updatePosition({ x, y }, { deltaX, deltaY }) {
    return { x: x + deltaX, y: y + deltaY };
}

```

### 6.8.3. Closures y Scope

Los closures (clausuras) son un concepto avanzado pero fundamental en JavaScript que permite a las funciones “recordar” el entorno en el que fueron creadas. En el contexto de videojuegos, los closures son especialmente útiles para crear sistemas como contadores de puntuación, generadores de identificadores únicos, o sistemas de estado que mantienen información privada.

Un closure se forma cuando una función interna hace referencia a variables de su función externa, y esa función interna se utiliza fuera de su contexto original. La función interna “cierra” sobre las variables del ámbito externo, manteniéndolas vivas incluso después de que la función externa haya terminado de ejecutarse:

```

function createCounter(initialValue = 0) {
    let count = initialValue;

    return {
        increment: () => ++count,
        decrement: () => --count,
        getValue: () => count
    };
}

let scoreCounter = createCounter(0);
scoreCounter.increment(); // 1
scoreCounter.increment(); // 2
console.log(scoreCounter.getValue()); // 2

```

## 6.9. Manejo de Excepciones

El manejo de excepciones en JavaScript funciona de manera muy similar a Java, utilizando los bloques `try`, `catch` y `finally`. En el desarrollo de videojuegos, el manejo adecuado de errores es crucial para crear experiencias robustas que no se rompan cuando ocurren situaciones inesperadas, como fallos al cargar recursos, errores de red, o datos corruptos en las partidas guardadas.

JavaScript permite lanzar cualquier tipo de objeto como excepción, aunque la práctica recomendada es usar objetos `Error` o crear errores personalizados con propiedades `name` y `message` descriptivas:

```
try {
    // Código que puede fallar
    let gameData = JSON.parse(savedGameString);
    loadGame(gameData);
} catch (error) {
    // Manejo del error
    console.error('Error loading game:', error.message);
    showErrorDialog('No se pudo cargar la partida guardada');
} finally {
    // Código que siempre se ejecuta
    hideLoadingSpinner();
}

// Lanzar excepciones personalizadas
function validatePlayerInput(input) {
    if (!input || input.trim() === '') {
        throw new Error('El nombre del jugador no puede estar vacío');
    }

    if (input.length > 20) {
        throw new Error('El nombre del jugador es demasiado largo');
    }
}
```

## 6.10. Programación Orientada a Objetos

### 6.10.1. Clases ES2015+

```
class GameObject {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.active = true;
    }

    update(deltaTime) {
        // Lógica base de actualización
    }

    render(context) {
        // Lógica base de renderizado
    }

    destroy() {
        this.active = false;
    }
}

class Player extends GameObject {
    constructor(x, y, name) {
        super(x, y);
        this.name = name;
        this.health = 100;
        this.maxHealth = 100;
        this.inventory = [];
    }

    // Getters y setters
    get isAlive() {
        return this.health > 0;
    }

    set health(value) {
        this._health = Math.max(0, Math.min(this.maxHealth, value));
    }
}
```

```

get health() {
    return this._health;
}

// Métodos
takeDamage(amount) {
    this.health -= amount;
    if (this.health <= 0) {
        this.destroy();
    }
}

heal(amount) {
    this.health += amount;
}

addItem(item) {
    this.inventory.push(item);
}

// Método estático
static createDefaultPlayer() {
    return new Player(0, 0, "Jugador");
}
}

```

### 6.10.2. Módulos ES2015+

```

// game-object.js
export class GameObject {
    // ... implementación
}

export const GAME_CONSTANTS = {
    GRAVITY: 9.8,
    MAX_SPEED: 200
};

// player.js
import { GameObject, GAME_CONSTANTS } from './game-object.js';

```

```

export class Player extends GameObject {
    // ... implementación
}

// main.js
import { Player } from './player.js';
import { Game } from './game.js';

const game = new Game();
const player = new Player(100, 100, "Jugador1");
game.addPlayer(player);

```

## 6.11. Almacenamiento de Datos en el Navegador

En el desarrollo de videojuegos web, frecuentemente necesitamos almacenar información que persista entre sesiones de juego. Esto incluye datos como puntuaciones máximas, configuraciones del jugador, progreso en niveles, preferencias de audio/video, y estados de partidas guardadas. JavaScript proporciona varios mecanismos para almacenar datos en el navegador del usuario, cada uno con características específicas que los hacen apropiados para diferentes situaciones.

### 6.11.1. Local Storage

Local Storage es una API moderna del navegador que permite almacenar datos de forma persistente en el dispositivo del usuario. A diferencia de las cookies, los datos del Local Storage no se envían automáticamente al servidor con cada petición HTTP, lo que los hace ideales para almacenar información que solo necesita el cliente.

Las características principales del Local Storage son:

- Persistencia:** Los datos permanecen hasta que el usuario los elimine manualmente o la aplicación los borre
- Capacidad:** Típicamente 5-10MB por dominio (mucho más que las cookies)
- Sincronía:** Las operaciones son síncronas y bloquean el hilo principal
- Ámbito:** Los datos son específicos del dominio y protocolo

#### 6.11.1.1. Guardar datos

Local Storage solo acepta strings, por lo que para guardar números u objetos debemos convertirlos primero:

```
// Guardar strings y números
localStorage.setItem('playerName', 'Jugador1');
localStorage.setItem('highScore', '15000');
```

Para objetos más complejos, utilizamos JSON:

```
const gameConfig = {
    volume: 0.8,
    difficulty: 'normal'
};
localStorage.setItem('gameConfig', JSON.stringify(gameConfig));
```

#### 6.11.1.2. Leer datos

Para recuperar los datos utilizamos `getItem()`:

```
const playerName = localStorage.getItem('playerName');
const highScore = localStorage.getItem('highScore');
```

Para objetos, debemos parsear el JSON de vuelta:

```
const configString = localStorage.getItem('gameConfig');
const config = configString ? JSON.parse(configString) : null;
```

#### 6.11.1.3. Verificar existencia

Podemos comprobar si existe un dato antes de intentar leerlo:

```
if (localStorage.getItem('playerName') !== null) {
    console.log('El jugador ya tiene un nombre guardado');
}
```

#### 6.11.1.4. Eliminar datos

Para eliminar datos específicos:

```
localStorage.removeItem('playerName');
```

Para eliminar todos los datos del dominio:

```
localStorage.clear();
```

#### 6.11.1.5. Información adicional

Podemos obtener información sobre el storage:

```
// Número de elementos almacenados
const cantidadItems = localStorage.length;

// Iterar sobre todas las claves
for (let i = 0; i < localStorage.length; i++) {
    const clave = localStorage.key(i);
    const valor = localStorage.getItem(clave);
    console.log(clave, valor);
}
```

#### 6.11.1.6. Manejo de errores

```
// Siempre usar try-catch con localStorage
function guardarDatos(clave, valor) {
    try {
        localStorage.setItem(clave, valor);
        return true;
    } catch (error) {
        console.error('Error al guardar:', error);
        // Puede fallar si el storage está lleno o en modo privado
        return false;
    }
}

function cargarDatos(clave) {
    try {
        return localStorage.getItem(clave);
    } catch (error) {
        console.error('Error al cargar:', error);
        return null;
    }
}
```

## 6.11.2. Session Storage

Session Storage funciona de manera idéntica a Local Storage, pero los datos solo persisten durante la sesión actual del navegador. Cuando el usuario cierra la pestaña o ventana, los datos se eliminan automáticamente.

La API es exactamente la misma que localStorage:

```
// Guardar datos temporales  
sessionStorage.setItem('tempData', 'valor temporal');
```

```
// Leer datos temporales  
const tempData = sessionStorage.getItem('tempData');
```

```
// Eliminar datos temporales  
sessionStorage.removeItem('tempData');  
sessionStorage.clear();
```

Es especialmente útil para datos que no deben persistir entre sesiones:

```
// Estado actual del juego que se pierde al cerrar  
sessionStorage.setItem('currentGameState', JSON.stringify(gameState));
```

## 6.11.3. Cookies

Las cookies son un mecanismo más antiguo pero siguen siendo útiles cuando el servidor necesita acceso a los datos o para configuraciones que deben enviarse automáticamente con las peticiones HTTP.

Características de las cookies:  
- Tamaño máximo de 4KB por cookie  
- Se envían automáticamente con cada petición HTTP al dominio  
- Tienen fecha de expiración configurable  
- Pueden configurarse como seguras (solo HTTPS) o HttpOnly

### 6.11.3.1. Escribir cookies

La sintaxis básica para crear una cookie:

```
document.cookie = "playerName=Jugador1";
```

Para añadir una fecha de expiración (por ejemplo, 30 días):

```
const fecha = new Date();
fecha.setTime(fecha.getTime() + (30 * 24 * 60 * 60 * 1000));
document.cookie = `highScore=15000; expires=${fecha.toUTCString()}; path=/`;
```

#### 6.11.3.2. Leer cookies

Leer cookies requiere parsear la cadena que devuelve `document.cookie`:

```
function getCookie(nombre) {
  const value = `; ${document.cookie}`;
  const parts = value.split(`; ${nombre}=`);
  if (parts.length === 2) {
    return parts.pop().split(';').shift();
  }
  return null;
}
```

```
const playerName = getCookie('playerName');
```

#### 6.11.3.3. Eliminar cookies

Para eliminar una cookie, establecemos una fecha de expiración en el pasado:

```
document.cookie = "playerName=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/";
```

#### 6.11.3.4. Opciones de seguridad

Las cookies pueden incluir opciones adicionales de seguridad:

```
document.cookie = "sessionId=abc123; Secure; SameSite=Strict; path=/";
```

#### 6.11.3.5. Utilidad para simplificar cookies

Dado que las cookies tienen una sintaxis más compleja, es útil crear funciones auxiliares:

```

const CookieUtil = {
  set(nombre, valor, dias = 7) {
    const fecha = new Date();
    fecha.setTime(fecha.getTime() + (dias * 24 * 60 * 60 * 1000));
    document.cookie = `${nombre}=${valor}; expires=${fecha.toUTCString()}; path=/`;
  }
};

get(nombre) {
  const value = `; ${document.cookie}`;
  const parts = value.split(`; ${nombre}=`);
  return parts.length === 2 ? parts.pop().split(';').shift() : null;
}

delete(nombre) {
  document.cookie = `${nombre}=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/`;
}

```

Uso de la utilidad:

```

CookieUtil.set('playerLevel', '5', 30); // 30 días
const level = CookieUtil.get('playerLevel');
CookieUtil.delete('playerLevel');

```

#### 6.11.4. Comparación y Recomendaciones de Uso

Característica	localStorage	sessionStorage	Cookies
<b>Capacidad</b>	~5-10MB	~5-10MB	4KB
<b>Persistencia</b>	Hasta eliminar manualmente	Solo la sesión	Configurable
<b>Envío al servidor</b>	No	No	Automático
<b>API</b>	Síncrona	Síncrona	Manual

Recomendaciones para videojuegos:

- **localStorage**: Configuraciones, puntuaciones, progreso del juego
- **sessionStorage**: Estado temporal, datos de sesión
- **cookies**: Autenticación, preferencias que el servidor necesita

### 6.11.5. Ejemplo Práctico: Sistema de Guardado

```
// Sistema simple de guardado para un juego
const GameSave = {
    save(gameData) {
        try {
            localStorage.setItem('gameSave', JSON.stringify(gameData));
            console.log('Juego guardado');
        } catch (error) {
            console.error('Error al guardar:', error);
        }
    },
    load() {
        try {
            const data = localStorage.getItem('gameSave');
            return data ? JSON.parse(data) : null;
        } catch (error) {
            console.error('Error al cargar:', error);
            return null;
        }
    },
    delete() {
        localStorage.removeItem('gameSave');
        console.log('Partida eliminada');
    },
    exists() {
        return localStorage.getItem('gameSave') !== null;
    }
};

// Uso
const gameData = {
    level: 5,
    score: 12500,
    lives: 3,
    powerUps: ['speed', 'jump']
};

GameSave.save(gameData);
```

```
const loadedData = GameSave.load();
if (GameSave.exists()) {
    console.log('Hay una partida guardada');
}
```

# 7 JavaScript Orientado a Objetos para Videojuegos

## 7.1. Introducción

JavaScript es el lenguaje fundamental para el desarrollo de videojuegos web modernos. A diferencia de otros lenguajes que habéis estudiado, JavaScript utiliza un sistema de orientación a objetos basado en **prototipos** en lugar de clases tradicionales, aunque las versiones modernas incluyen sintaxis de clases que simplifican el desarrollo.

En este capítulo profundizaremos en las características únicas de JavaScript para videojuegos: la orientación a objetos basada en prototipos, las clases modernas ES2015+, y cómo trabajar con objetos dinámicos.

## 7.2. Orientación a Objetos en JavaScript

### 7.2.1. Diferencias con Java: Clases vs Prototipos

Mientras que Java utiliza un sistema basado en clases donde los objetos son instancias de una clase predefinida, JavaScript tradicionalmente utiliza **prototipos**. En JavaScript, cualquier objeto puede servir como prototipo para otros objetos, creando una cadena de herencia más flexible.

#### 7.2.1.1. Creación Básica de Objetos

```
// Objeto literal simple
const enemigo = {
    vida: 100,
    damage: 15,
    atacar() {
        return `Enemigo ataca causando ${this.damage} puntos de daño`;
    }
};
```

```
// Crear otro enemigo basado en el prototipo
const goblin = Object.create(enemigo);
goblin.vida = 50;
goblin.damage = 8;
goblin.tipo = "Goblin";

console.log(goblin.atacar()); // "Enemigo ataca causando 8 puntos de daño"
console.log(goblin.vida);    // 50 (propio)
console.log(goblin.toString()); // function (heredado de Object)
```

En este ejemplo, `goblin` hereda el método `atacar()` del objeto `enemigo`. Cuando JavaScript busca una propiedad en `goblin` y no la encuentra, automáticamente busca en su prototipo (`enemigo`), y si no la encuentra ahí, continúa hacia arriba en la cadena hasta llegar a `Object.prototype`.

### 7.2.1.2. Herencia con Prototipos

Para crear una jerarquía de herencia más compleja con prototipos, necesitamos establecer la cadena de prototipos manualmente:

```
// Prototipo base
const personajeBase = {
  mover(x, y) {
    this.x += x;
    this.y += y;
    console.log(`"${this.nombre}" se mueve a (${this.x}, ${this.y})`);
  },
  toString() {
    return `${this.nombre}: vida=${this.vida}, pos=(${this.x},${this.y})`;
  }
};

// Prototipo específico para jugadores
const protottipoJugador = Object.create(personajeBase);
protottipoJugador.atacar = function(objetivo) {
  return `${this.nombre} ataca a ${objetivo.nombre} por ${this.fuerza} puntos`;
};

protottipoJugador.levelUp = function() {
  this.nivel++;
```

```

    this.fuerza += 5;
    console.log(`"${this.nombre} sube a nivel ${this.nivel}!`);
};

// Crear instancia de jugador
const jugador = Object.create(prototipoJugador);
jugador.nombre = "Aragorn";
jugador.vida = 100;
jugador.x = 0;
jugador.y = 0;
jugador.fuerza = 20;
jugador.nivel = 1;

jugador.mover(5, 3); // Método heredado de personajeBase
jugador.levelUp(); // Método del prototipoJugador

```

Este ejemplo muestra una cadena de herencia: `jugador` → `prototipoJugador` → `personajeBase` → `Object.prototype`. Cada nivel puede añadir o sobrescribir métodos.

#### 7.2.1.3. Acceso a Propiedades y Métodos de Objetos

JavaScript permite acceder a las propiedades de un objeto tanto con notación punto como con corchetes:

```

const config = {
  sonido: true,
  volumen: 0.8,
  idioma: "es"
};

// Acceso con notación punto (recomendado)
console.log(config.sonido); // true
config.volumen = 0.5;

// Acceso con corchetes (útil para propiedades dinámicas)
console.log(config["idioma"]); // "es"
const propiedad = "volumen";
console.log(config[propiedad]); // 0.5

// Añadir nuevas propiedades dinámicamente

```

```
config.dificultad = "normal";
config["maxJugadores"] = 4;
```

La notación con corchetes es especialmente útil cuando el nombre de la propiedad está en una variable o cuando queremos iterar sobre las propiedades del objeto.

#### 7.2.1.4. Iteración sobre Propiedades

```
const inventario = {
    espada: 1,
    pocion: 5,
    oro: 150
};

// Iterar sobre todas las propiedades propias
for (let item in inventario) {
    console.log(`"${item}": ${inventario[item]}`);
}

// Verificar si una propiedad existe
if ("oro" in inventario) {
    console.log("El jugador tiene oro");
}

// Obtener todas las claves como array
const items = Object.keys(inventario);
console.log(items); // ["espada", "potion", "oro"]

// Obtener todos los valores como array
const cantidades = Object.values(inventario);
console.log(cantidades); // [1, 5, 150]
```

El bucle `for...in` itera sobre todas las propiedades enumerables del objeto, incluyendo las heredadas del prototipo. Si solo queremos las propiedades propias del objeto, podemos usar `Object.keys()` o verificar con `hasOwnProperty()`.

#### 7.2.1.5. Diferencias entre null y undefined

JavaScript tiene dos valores especiales para representar “nada”:

```

let jugador = null;           // Ausencia intencional de valor
let powerUp;                 // undefined - no inicializada

// Verificación segura antes de usar objetos
if (jugador) {
    jugador.mover(5, 0);    // Solo se ejecuta si jugador no es null/undefined
}

// Verificación más específica
if (jugador !== null && jugador !== undefined) {
    jugador.atacar();
}

// Operador de optional chaining (ES2020+)
jugador?.mover?(5, 0);     // Solo ejecuta si jugador y mover existen

```

Esta diferencia es importante en videojuegos donde objetos pueden existir o no (enemigos destruidos, power-ups recogidos, etc.).

### 7.2.2. Función Constructor (Patrón Tradicional)

Antes de ES2015, el patrón más común para crear “clases” era usar funciones constructor:

```

function Jugador(nombre, x, y) {
    this.nombre = nombre;
    this.x = x;
    this.y = y;
    this.vida = 100;
    this.velocidad = 5;
}

// Métodos compartidos en el prototipo
Jugador.prototype.mover = function(deltaX, deltaY) {
    this.x += deltaX * this.velocidad;
    this.y += deltaY * this.velocidad;
    console.log(`${this.nombre} se mueve a (${this.x}, ${this.y})`);
};

Jugador.prototype.recibirDanio = function(cantidad) {
    this.vida -= cantidad;
    if (this.vida <= 0) {

```

```

        console.log(`"${this.nombre}" ha sido derrotado!`);
        return false;
    }
    return true;
};

// Uso del constructor
const player1 = new Jugador("Aragorn", 10, 20);
player1.mover(2, 3); // "Aragorn se mueve a (20, 35)"

// IMPORTANTE: Siempre usar 'new'
const player2 = Jugador("Legolas", 0, 0); // ¡ERROR! Sin 'new'
// Sin 'new', 'this' apuntaría al objeto global, causando problemas

```

La función constructor se ejecuta cuando usamos el operador `new`. Este operador crea un nuevo objeto, establece `this` para que apunte a ese objeto, y al final devuelve el objeto automáticamente.

Los métodos se definen en `Jugador.prototype` para que todos los objetos creados con este constructor compartan las mismas funciones en memoria, siendo más eficiente que definir los métodos dentro del constructor.

#### 7.2.2.1. Herencia con Funciones Constructor

```

// Constructor padre
function Personaje(nombre, vida) {
    this.nombre = nombre;
    this.vida = vida;
}

Personaje.prototype.saludar = function() {
    return `Hola, soy ${this.nombre}`;
};

// Constructor hijo
function Guerrero(nombre, vida, fuerza) {
    Personaje.call(this, nombre, vida); // Llamar al constructor padre
    this.fuerza = fuerza;
}

```

```

// Establecer herencia de prototipos
Guerrero.prototype = Object.create(Personaje.prototype);
Guerrero.prototype.constructor = Guerrero;

// Añadir métodos específicos del guerrero
Guerrero.prototype.atacar = function() {
    return `${this.nombre} ataca con fuerza ${this.fuerza}`;
};

const conan = new Guerrero("Conan", 150, 25);
console.log(conan.saludar()); // "Hola, soy Conan" (heredado)
console.log(conan.atacar()); // "Conan ataca con fuerza 25" (propio)

```

Este patrón requiere tres pasos: llamar al constructor padre con `call()`, establecer la cadena de prototipos con `Object.create()`, y restaurar la propiedad `constructor`.

### 7.2.3. Clases ES2015+ (Sintaxis Moderna)

Las clases modernas ofrecen una sintaxis más familiar para desarrolladores de Java, pero internamente siguen usando prototipos. La palabra clave `class` es “azúcar sintáctico” sobre el sistema de prototipos existente.

```

class GameObject {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.activo = true;
    }

    actualizar(deltaTime) {
        // Lógica base de actualización que pueden sobrescribir las subclases
        if (!this.activo) return;
    }

    destruir() {
        this.activo = false;
        console.log("Objeto destruido");
    }
}

class Enemigo extends GameObject {

```

```

constructor(x, y, tipo) {
    super(x, y); // Llamada obligatoria al constructor padre
    this.tipo = tipo;
    this.vida = 50;
    this.velocidad = 2;
    this._direccion = { x: 1, y: 0 };
}

// Getter: se accede como una propiedad, pero ejecuta código
get estaVivo() {
    return this.vida > 0;
}

// Setter: permite validación cuando se asigna un valor
set vida(valor) {
    this._vida = Math.max(0, valor); // No puede ser negativa
    if (this._vida === 0) {
        this.destruir();
    }
}

get vida() {
    return this._vida;
}

actualizar(deltaTime) {
    super.actualizar(deltaTime); // Llamar al método padre
    if (this.estaVivo) {
        this.x += this._direccion.x * this.velocidad;
        this.y += this._direccion.y * this.velocidad;
    }
}

// Método estático: pertenece a la clase, no a las instancias
static crearOrc() {
    const orc = new Enemigo(0, 0, "Orc");
    orc.vida = 80;
    orc.velocidad = 1.5;
    return orc;
}
}

```

### Explicación de elementos clave:

- **constructor**: Método especial que se ejecuta al crear una instancia. Es equivalente a la función constructor del patrón tradicional.
- **super()**: En el constructor, llama al constructor de la clase padre. Debe ser la primera línea antes de usar **this**.
- **extends**: Establece herencia entre clases. `Enemigo extends GameObject` significa que `Enemigo` hereda de `GameObject`.
- **get estaVivo()**: Es un **getter**, una propiedad calculada que se accede como `enemigo.estaVivo` pero ejecuta código. No necesita paréntesis al accederla.
- **set vida(valor)**: Es un **setter**, permite interceptar y validar cuando se asigna un valor a la propiedad. Se usa como `enemigo.vida = 100`.
- **static crearOrc()**: Método estático que pertenece a la clase, no a las instancias. Se llama como `Enemigo.crearOrc()`.
- **super.actualizar(deltaTime)**: En un método, llama al método del mismo nombre en la clase padre.

#### 7.2.3.1. Ejemplo de Uso

```
// Crear enemigos usando el constructor normal
const goblin = new Enemigo(10, 20, "Goblin");
goblin.vida = 30; // Usa el setter, valida que no sea negativo

// Usar el getter
if (goblin.estaVivo) { // No necesita paréntesis
    console.log("El goblin sigue vivo");
}

// Crear enemigo usando método estático
const orc = Enemigo.crearOrc();

// Polimorfismo: ambos objetos tienen el mismo interfaz
const enemigos = [goblin, orc];
enemigos.forEach(enemigo => {
    enemigo.actualizar(16); // Cada uno usa su propia implementación
});
```

### 7.2.3.2. Ventajas de las Clases ES2015+

1. **Sintaxis más clara:** Más familiar para desarrolladores de otros lenguajes orientados a objetos.
2. **Herencia simplificada:** `extends` y `super()` son más directos que manipular prototipos manualmente.
3. **Getters y setters integrados:** Permiten crear propiedades con lógica de validación o cálculo.
4. **Métodos estáticos:** Para funciones que pertenecen a la clase conceptualmente pero no necesitan una instancia.
5. **Mejor soporte de herramientas:** Los IDEs y linters comprenden mejor las clases modernas.

La sintaxis de clases es la recomendada para proyectos nuevos de videojuegos, especialmente cuando trabajáis en equipo o cuando el proyecto va a crecer en complejidad. Sin embargo, entender el sistema de prototipos subyacente os ayudará a comprender mejor cómo funciona JavaScript internamente y a debuggear problemas más efectivamente.

## 7.3. Ejercicios Prácticos

### 7.3.1. Ejercicio 1: Separación de Arrays

Crear una función que reciba un array como parámetro y devuelva un array de dos elementos: el primer elemento contendrá los números pares y el segundo los números impares.

```
function separarParesImpares(numeros) {  
    const pares = [];  
    const impares = [];  
  
    for (let numero of numeros) {  
        if (numero % 2 === 0) {  
            pares.push(numero);  
        } else {  
            impares.push(numero);  
        }  
    }  
  
    return [pares, impares];  
}
```

```
// Versión con métodos modernos
const separarParesImparesModerno = (numeros) => [
    numeros.filter(n => n % 2 === 0),
    numeros.filter(n => n % 2 !== 0)
];

// Pruebas
console.log(separarParesImpares([1, 2, 3, 4, 5, 6]));
// [[2, 4, 6], [1, 3, 5]]
```

### 7.3.2. Ejercicio 2: Procesamiento de Array Bidimensional

Crear una función que reciba un array bidimensional, elimine los ceros y ordene las filas por longitud.

```
function procesarArrayBidimensional(matriz) {
    // Eliminar ceros de cada fila
    const sinCeros = matriz.map(fila =>
        fila.filter(elemento => elemento !== 0)
    );

    // Ordenar por longitud de fila
    return sinCeros.sort((a, b) => a.length - b.length);
}

// Prueba
const matriz = [
    [7, 0, 2, 1, 0, 1],
    [3, 0, 0, 2],
    [7, 9, 0],
    [6, 5, 0, 1, 0, 2, 0]
];

console.log(procesarArrayBidimensional(matriz));
// [[3, 2], [7, 9], [7, 2, 1, 1], [6, 5, 1, 2]]
```

Este capítulo proporciona una base sólida de JavaScript moderno para el desarrollo de videojuegos, cubriendo desde la configuración del entorno de desarrollo hasta las características avanzadas del lenguaje que son esenciales para crear juegos web interactivos y eficientes.

# 8 Introducción a Phaser 3

Phaser 3 es un framework open source de HTML5 diseñado para la creación de videojuegos que se ejecutan directamente en navegadores web. Liberado en 2018, representa una evolución significativa respecto a versiones anteriores, ofreciendo un sistema más modular y potente para el desarrollo de juegos 2D.

## 8.1. Características principales

El framework se caracteriza por su versatilidad y accesibilidad. Al estar basado en tecnologías web estándar (HTML5, JavaScript y Canvas/WebGL), permite crear juegos que no requieren instalación adicional por parte del usuario final. Esto facilita enormemente la distribución y accesibilidad de los juegos desarrollados.

Phaser 3 soporta dos sistemas de renderizado: **Canvas**, que es el modo por defecto y el más compatible entre navegadores y dispositivos, y **WebGL**, que permite gráficos más avanzados pero requiere mayor soporte del hardware. La elección entre uno u otro dependerá de las necesidades específicas del proyecto y del público objetivo.

Está orientado tanto a escritorio como a dispositivos móviles, lo que lo convierte en una opción ideal para el desarrollo de juegos multiplataforma sin necesidad de reescribir código para diferentes sistemas operativos.

## 8.2. Requisitos para empezar

Para comenzar a desarrollar con Phaser 3 necesitamos:

### 8.2.1. Navegador web

Cualquier navegador moderno funcionará, aunque se recomienda por orden de preferencia: Chrome, Firefox, Safari, Edge u Opera. Chrome suele ofrecer las mejores herramientas de desarrollo integradas.

### 8.2.2. Framework Phaser 3

Podemos obtenerlo de tres formas:

**Descarga directa:** Desde <https://phaser.io/download/stable>

**Repositorio GitHub:** <https://github.com/photonstorm/phaser>

**CDN** (la más rápida para empezar):

```
<script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
```

## 8.3. Estructura básica de un juego en Phaser 3

### 8.3.1. El elemento Canvas

Phaser utiliza el elemento HTML5 `<canvas>` para renderizar los gráficos del juego. Este elemento es una etiqueta de HTML5 que permite dibujar gráficos mediante JavaScript y CSS.

El canvas funciona como un lienzo en blanco donde podemos dibujar formas, imágenes y animaciones. A diferencia de un elemento `<img>`, el canvas no tiene atributos `src` ni `alt`, pero sí tiene `width` y `height` que definen el espacio de coordenadas (por defecto 300x300 píxeles).

El sistema de coordenadas del canvas coloca el origen (0,0) en la esquina superior izquierda, con el eje X positivo hacia la derecha y el eje Y positivo hacia abajo.

### 8.3.2. Configuración inicial del juego

La estructura mínima de un archivo HTML con Phaser tiene este aspecto:

```
<!DOCTYPE html>
<html>
<head>
    <script src="phaser.js"></script>
</head>
<body>
    <script>
        var config = {
            type: Phaser.AUTO,
            width: 800,
            height: 600,
            physics: {
```

```

        default: 'arcade',
        arcade: {
            gravity: { y: 300 }
        }
    },
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);

function preload() {
    // Cargar recursos
}

function create() {
    // Crear objetos del juego
}

function update() {
    // Actualizar cada frame
}
</script>
</body>
</html>

```

### 8.3.2.1. Objeto de configuración

El objeto `config` define los parámetros fundamentales del juego:

- **type:** Especifica el sistema de renderizado (`Phaser.AUTO`, `Phaser.CANVAS` o `Phaser.WEBGL`)
- **width/height:** Dimensiones del canvas en píxeles
- **physics:** Configuración del motor de físicas
- **scene:** Funciones que componen la escena del juego

### 8.3.2.2. Las tres funciones principales

**preload()**: Se ejecuta automáticamente al inicio para cargar todos los recursos necesarios (imágenes, sonidos, sprites). Phaser espera a que todos los recursos se carguen antes de continuar.

```
function preload() {
    this.load.image('cielo', 'assets/cielo.png');
    this.load.image('suelo', 'assets/plataforma.png');
    this.load.image('estrella', 'assets/estrella.png');
}
```

En este ejemplo cargamos tres recursos que luego podremos utilizar en el juego mediante sus identificadores ('cielo', 'suelo', 'estrella').

**create()**: Se ejecuta una vez después de preload(), se utiliza para crear y posicionar los objetos del juego. Los elementos se muestran en el orden en que se añaden.

```
function create() {
    this.add.image(400, 300, 'cielo');
    var plataforma = this.physics.add.image(400, 500, 'suelo');
    plataforma.setCollideWorldBounds(true);
}
```

Aquí añadimos una imagen de fondo centrada en el canvas y creamos una plataforma con físicas. El método `setCollideWorldBounds(true)` hace que la plataforma no pueda salirse de los límites del juego.

**update(time, delta)**: Es un bucle infinito que se ejecuta constantemente (aproximadamente 60 veces por segundo). Aquí se maneja la lógica del juego que debe actualizarse continuamente. Los parámetros `time` y `delta` ayudan a garantizar que el movimiento sea consistente en diferentes dispositivos.

```
function update(time, delta) {
    if (this.cursors.left.isDown) {
        this.jugador.x -= 5;
    }
    if (this.cursors.right.isDown) {
        this.jugador.x += 5;
    }
}
```

En este ejemplo comprobamos continuamente si las teclas de flecha están pulsadas y movemos al jugador 5 píxeles en la dirección correspondiente.

## 8.4. Gestión de escenas

### 8.4.1. Concepto de escena

Una escena en Phaser 3 puede interpretarse como una pantalla independiente del juego con su propio flujo de ejecución. Por ejemplo, un juego típico podría tener escenas para el menú principal, el gameplay, la pantalla de pausa, y la pantalla de game over.

Cada escena mantiene su propio conjunto de recursos, objetos y lógica, lo que permite organizar el código de manera modular y mantener separadas las diferentes fases del juego.

### 8.4.2. SceneManager

Phaser 3 incluye un **SceneManager** que se encarga de gestionar todas las escenas del juego. Podemos definir las escenas iniciales en la configuración:

```
var config = {
    scene: [MenuPrincipal, Juego, GameOver]
};
```

Este array define el orden en que se cargarán las escenas. La primera del array (**MenuPrincipal**) será la que se inicie automáticamente.

#### 8.4.2.1. Añadir y eliminar escenas dinámicamente

```
// Añadir una nueva escena en tiempo de ejecución
var nuevaEscena = this.scene.add('clave', ConfigEscena, autoStart, datos);

// Eliminar una escena
this.scene.remove('clave');
```

El parámetro **autoStart** determina si la escena se inicia inmediatamente, mientras que **datos** permite pasar información a la nueva escena.

#### 8.4.2.2. Gestionar escenas

Phaser ofrece múltiples formas de controlar el estado de las escenas:

```

// Iniciar una escena (apaga la actual)
this.scene.start('clave', datos);

// Lanzar en paralelo (mantiene la actual)
this.scene.launch('clave', datos);

// Pausar (detiene update pero sigue renderizando)
this.scene.pause('clave');

// Reanudar una escena pausada
this.scene.resume('clave');

// Dormir (detiene update y renderizado)
this.scene.sleep('clave');

// Despertar una escena dormida
this.scene.wake('clave');

// Detener completamente (limpia recursos)
this.scene.stop('clave');

```

La diferencia clave entre `start` y `launch` es que `start` detiene la escena actual, mientras que `launch` permite ejecutar múltiples escenas simultáneamente. Esto es útil para mostrar interfaces como menús de pausa sobre el juego activo.

#### 8.4.2.3. Transiciones entre escenas

Podemos crear transiciones visuales suaves entre escenas:

```

this.scene.transition({
  target: 'SiguienteEscena',
  duration: 1000, // Duración en milisegundos
  moveBelow: true,
  onUpdate: (progress) => {
    // Código durante la transición
  }
});

```

La función `onUpdate` se ejecuta durante toda la transición, recibiendo un valor `progress` que va de 0 a 1, permitiendo crear efectos visuales personalizados como fundidos o desvanecimientos.

#### 8.4.2.4. Reordenar escenas

Las escenas tienen un orden de renderizado que puede modificarse:

```
// Mover arriba/abajo en la pila
this.scene.moveUp('clave');
this.scene.moveDown('clave');

// Intercambiar posición de dos escenas
this.scene.swapPosition('claveA', 'claveB');

// Mover encima/debajo de otra escena
this.scene.moveAbove('clave', 'claveReferencia');
this.scene.moveBelow('clave', 'claveReferencia');

// Traer al frente o enviar al fondo
this.scene.bringToFront('clave');
this.scene.sendToBack('clave');
```

El orden de renderizado determina qué escenas se dibujan sobre otras. Esto es especialmente útil cuando se ejecutan múltiples escenas en paralelo, como un HUD sobre la pantalla de juego.

**Ejemplo práctico:** Sistema de menú con pausa

```
class Juego extends Phaser.Scene {
    constructor() {
        super('Juego');
    }

    create() {
        // Configurar el juego
        this.input.keyboard.on('keydown-ESC', () => {
            this.scene.pause();
            this.scene.launch('MenuPausa');
        });
    }
}

class MenuPausa extends Phaser.Scene {
    constructor() {
        super('MenuPausa');
    }
}
```

```

create() {
    let botonContinuar = this.add.text(400, 300, 'Continuar');
    botonContinuar.setInteractive();

    botonContinuar.on('pointerdown', () => {
        this.scene.stop();
        this.scene.resume('Juego');
    });
}

```

Este ejemplo muestra cómo implementar un sistema de pausa: cuando el jugador pulsa ESC durante el juego, se pausa la escena principal y se lanza el menú de pausa sobre ella. Al hacer clic en “Continuar”, el menú se cierra y el juego se reanuda desde donde estaba.

## 8.5. Trabajo con imágenes

### 8.5.1. Cargar y mostrar imágenes

Las imágenes son elementos fundamentales en cualquier juego. En Phaser, primero debemos cargarlas en `preload()` y luego añadirlas al juego en `create()`:

```

function preload() {
    // Cargar la imagen con un identificador único
    this.load.image('personaje', 'assets/personaje.png');
    this.load.image('fondo', 'assets/fondo.jpg');
}

function create() {
    // Añadir la imagen al canvas en posición (x, y)
    this.add.image(400, 300, 'fondo');
    let sprite = this.add.image(200, 150, 'personaje');
}

```

Es importante cargar las imágenes en `preload()` para garantizar que estén disponibles antes de intentar usarlas. El identificador que asignamos ('personaje', 'fondo') es la clave que utilizaremos después para referenciar estas imágenes.

### 8.5.2. Sistema de coordenadas y origen

Por defecto, todas las imágenes se posicionan usando su centro como punto de referencia. Podemos cambiar este comportamiento:

```
// Establecer el origen en la esquina superior izquierda
let imagen = this.add.image(100, 100, 'personaje').setOrigin(0, 0);

// Origen en el centro inferior
let imagen2 = this.add.image(200, 200, 'personaje').setOrigin(0.5, 1);
```

Los valores de origen van de 0 a 1, donde (0,0) es la esquina superior izquierda y (1,1) es la inferior derecha de la imagen. Cambiar el origen es útil para alinear objetos o hacer que rotén alrededor de puntos específicos.

### 8.5.3. Transformaciones básicas

```
function create() {
    let jugador = this.add.image(400, 300, 'personaje');

    // Escalar la imagen (0.5 = mitad del tamaño, 2 = doble)
    jugador.setScale(1.5);

    // Escalar de forma independiente en X e Y
    jugador.setScale(2, 0.5);

    // Voltar horizontalmente
    jugador.flipX = true;

    // Voltar verticalmente
    jugador.flipY = true;

    // Rotar (en radianes)
    jugador.rotation = Math.PI / 4; // 45 grados

    // Cambiar profundidad (controla qué se dibuja encima)
    jugador.depth = 10;
}
```

El escalado independiente en X e Y permite crear efectos de estiramiento. La propiedad `depth` funciona como las capas en programas de diseño: valores mayores se dibujan sobre valores menores. Para convertir grados a radianes, recordemos que `radianes = 180 grados.`

**Ejemplo práctico:** Sprite que sigue al ratón

```
let jugador;

function create() {
    jugador = this.add.image(400, 300, 'nave');
}

function update() {
    // Obtener posición del ratón
    let pointer = this.input.activePointer;

    // Mover suavemente hacia el ratón
    jugador.x += (pointer.x - jugador.x) * 0.1;
    jugador.y += (pointer.y - jugador.y) * 0.1;

    // Rotar hacia la dirección del movimiento
    let angulo = Phaser.Math.Angle.Between(
        jugador.x, jugador.y,
        pointer.x, pointer.y
    );
    jugador.rotation = angulo;
}
```

Este ejemplo crea un efecto de seguimiento suave: en lugar de mover el sprite directamente a la posición del ratón, solo se desplaza el 10% de la distancia en cada frame (multiplicador 0.1). Esto produce un movimiento más natural y fluido. La función `Angle.Between` calcula automáticamente el ángulo necesario para que el sprite “mire” hacia el cursor.

## 8.6. Introducción a Phaser 3

Phaser 3 es un framework open source de HTML5 diseñado para la creación de videojuegos que se ejecutan directamente en navegadores web. Liberado en 2018, representa una evolución significativa respecto a versiones anteriores, ofreciendo un sistema más modular y potente para el desarrollo de juegos 2D.

### **8.6.1. Características principales**

El framework se caracteriza por su versatilidad y accesibilidad. Al estar basado en tecnologías web estándar (HTML5, JavaScript y Canvas/WebGL), permite crear juegos que no requieren instalación adicional por parte del usuario final. Esto facilita enormemente la distribución y accesibilidad de los juegos desarrollados.

Phaser 3 soporta dos sistemas de renderizado: **Canvas**, que es el modo por defecto y el más compatible entre navegadores y dispositivos, y **WebGL**, que permite gráficos más avanzados pero requiere mayor soporte del hardware. La elección entre uno u otro dependerá de las necesidades específicas del proyecto y del público objetivo.

Está orientado tanto a escritorio como a dispositivos móviles, lo que lo convierte en una opción ideal para el desarrollo de juegos multiplataforma sin necesidad de reescribir código para diferentes sistemas operativos.

### **8.6.2. Requisitos para empezar**

Para comenzar a desarrollar con Phaser 3 necesitamos:

#### **8.6.2.1. Navegador web**

Cualquier navegador moderno funcionará, aunque se recomienda por orden de preferencia: Chrome, Firefox, Safari, Edge u Opera. Chrome suele ofrecer las mejores herramientas de desarrollo integradas.

#### **8.6.2.2. Framework Phaser 3**

Podemos obtenerlo de tres formas:

**Descarga directa:** Desde <https://phaser.io/download/stable>

**Repositorio GitHub:** <https://github.com/photonstorm/phaser>

**CDN** (la más rápida para empezar):

```
<script src="//cdn.jsdelivr.net/npm/phaser@3.55.2/dist/phaser.js"></script>
```

## 8.7. Estructura básica de un juego en Phaser 3

### 8.7.1. El elemento Canvas

Phaser utiliza el elemento HTML5 <canvas> para renderizar los gráficos del juego. Este elemento es una etiqueta de HTML5 que permite dibujar gráficos mediante JavaScript y CSS.

El canvas funciona como un lienzo en blanco donde podemos dibujar formas, imágenes y animaciones. A diferencia de un elemento <img>, el canvas no tiene atributos `src` ni `alt`, pero sí tiene `width` y `height` que definen el espacio de coordenadas (por defecto 300x300 píxeles).

El sistema de coordenadas del canvas coloca el origen (0,0) en la esquina superior izquierda, con el eje X positivo hacia la derecha y el eje Y positivo hacia abajo.

### 8.7.2. Configuración inicial del juego

La estructura mínima de un archivo HTML con Phaser tiene este aspecto:

```
<!DOCTYPE html>
<html>
<head>
    <script src="phaser.js"></script>
</head>
<body>
    <script>
        var config = {
            type: Phaser.AUTO,
            width: 800,
            height: 600,
            physics: {
                default: 'arcade',
                arcade: {
                    gravity: { y: 300 }
                }
            },
            scene: {
                preload: preload,
                create: create,
                update: update
            }
        };
    </script>
</body>
</html>
```

```

var game = new Phaser.Game(config);

function preload() {
    // Cargar recursos
}

function create() {
    // Crear objetos del juego
}

function update() {
    // Actualizar cada frame
}
</script>
</body>
</html>

```

#### 8.7.2.1. Objeto de configuración

El objeto `config` define los parámetros fundamentales del juego:

- **type**: Especifica el sistema de renderizado (`Phaser.AUTO`, `Phaser.CANVAS` o `Phaser.WEBGL`)
- **width/height**: Dimensiones del canvas en píxeles
- **physics**: Configuración del motor de físicas
- **scene**: Funciones que componen la escena del juego

#### 8.7.2.2. Las tres funciones principales

**preload()**: Se ejecuta automáticamente al inicio para cargar todos los recursos necesarios (imágenes, sonidos, sprites). Phaser espera a que todos los recursos se carguen antes de continuar.

```

function preload() {
    this.load.image('cielo', 'assets/cielo.png');
    this.load.image('suelo', 'assets/plataforma.png');
    this.load.image('estrella', 'assets/estrella.png');
}

```

En este ejemplo cargamos tres recursos que luego podremos utilizar en el juego mediante sus identificadores ('cielo', 'suelo', 'estrella').

**create()**: Se ejecuta una vez después de preload(), se utiliza para crear y posicionar los objetos del juego. Los elementos se muestran en el orden en que se añaden.

```
function create() {
    this.add.image(400, 300, 'cielo');
    var plataforma = this.physics.add.image(400, 500, 'suelo');
    plataforma.setCollideWorldBounds(true);
}
```

Aquí añadimos una imagen de fondo centrada en el canvas y creamos una plataforma con físicas. El método `setCollideWorldBounds(true)` hace que la plataforma no pueda salirse de los límites del juego.

**update(time, delta)**: Es un bucle infinito que se ejecuta constantemente (aproximadamente 60 veces por segundo). Aquí se maneja la lógica del juego que debe actualizarse continuamente. Los parámetros `time` y `delta` ayudan a garantizar que el movimiento sea consistente en diferentes dispositivos.

```
function update(time, delta) {
    if (this.cursors.left.isDown) {
        this.jugador.x -= 5;
    }
    if (this.cursors.right.isDown) {
        this.jugador.x += 5;
    }
}
```

En este ejemplo comprobamos continuamente si las teclas de flecha están pulsadas y movemos al jugador 5 píxeles en la dirección correspondiente.

## 8.8. Gestión de escenas

### 8.8.1. Concepto de escena

Una escena en Phaser 3 puede interpretarse como una pantalla independiente del juego con su propio flujo de ejecución. Por ejemplo, un juego típico podría tener escenas para el menú principal, el gameplay, la pantalla de pausa, y la pantalla de game over.

Cada escena mantiene su propio conjunto de recursos, objetos y lógica, lo que permite organizar el código de manera modular y mantener separadas las diferentes fases del juego.

## 8.8.2. SceneManager

Phaser 3 incluye un `SceneManager` que se encarga de gestionar todas las escenas del juego. Podemos definir las escenas iniciales en la configuración:

```
var config = {
    scene: [MenuPrincipal, Juego, GameOver]
};
```

Este array define el orden en que se cargarán las escenas. La primera del array (`MenuPrincipal`) será la que se inicie automáticamente.

### 8.8.2.1. Añadir y eliminar escenas dinámicamente

```
// Añadir una nueva escena en tiempo de ejecución
var nuevaEscena = this.scene.add('clave', ConfigEscena, autoStart, datos);

// Eliminar una escena
this.scene.remove('clave');
```

El parámetro `autoStart` determina si la escena se inicia inmediatamente, mientras que `datos` permite pasar información a la nueva escena.

### 8.8.2.2. Gestionar escenas

Phaser ofrece múltiples formas de controlar el estado de las escenas:

```
// Iniciar una escena (apaga la actual)
this.scene.start('clave', datos);

// Lanzar en paralelo (mantiene la actual)
this.scene.launch('clave', datos);

// Pausar (detiene update pero sigue renderizando)
this.scene.pause('clave');

// Reanudar una escena pausada
this.scene.resume('clave');
```

```

// Dormir (detiene update y renderizado)
this.scene.sleep('clave');

// Despertar una escena dormida
this.scene.wake('clave');

// Detener completamente (limpia recursos)
this.scene.stop('clave');

```

La diferencia clave entre `start` y `launch` es que `start` detiene la escena actual, mientras que `launch` permite ejecutar múltiples escenas simultáneamente. Esto es útil para mostrar interfaces como menús de pausa sobre el juego activo.

#### 8.8.2.3. Transiciones entre escenas

Podemos crear transiciones visuales suaves entre escenas:

```

this.scene.transition({
    target: 'SiguienteEscena',
    duration: 1000, // Duración en milisegundos
    moveBelow: true,
    onUpdate: (progress) => {
        // Código durante la transición
    }
});

```

La función `onUpdate` se ejecuta durante toda la transición, recibiendo un valor `progress` que va de 0 a 1, permitiendo crear efectos visuales personalizados como fundidos o desvanecimientos.

#### 8.8.2.4. Reordenar escenas

Las escenas tienen un orden de renderizado que puede modificarse:

```

// Mover arriba/abajo en la pila
this.scene.moveUp('clave');
this.scene.moveDown('clave');

// Intercambiar posición de dos escenas
this.scene.swapPosition('claveA', 'claveB');

```

```

// Mover encima/debajo de otra escena
this.scene.moveAbove('clave', 'claveReferencia');
this.scene.moveBelow('clave', 'claveReferencia');

// Traer al frente o enviar al fondo
this.scene.bringToTop('clave');
this.scene.sendToBack('clave');

```

El orden de renderizado determina qué escenas se dibujan sobre otras. Esto es especialmente útil cuando se ejecutan múltiples escenas en paralelo, como un HUD sobre la pantalla de juego.

**Ejemplo práctico:** Sistema de menú con pausa

```

class Juego extends Phaser.Scene {
    constructor() {
        super('Juego');
    }

    create() {
        // Configurar el juego
        this.input.keyboard.on('keydown-ESC', () => {
            this.scene.pause();
            this.scene.launch('MenuPausa');
        });
    }
}

class MenuPausa extends Phaser.Scene {
    constructor() {
        super('MenuPausa');
    }

    create() {
        let botonContinuar = this.add.text(400, 300, 'Continuar');
        botonContinuar.setInteractive();

        botonContinuar.on('pointerdown', () => {
            this.scene.stop();
            this.scene.resume('Juego');
        });
    }
}

```

Este ejemplo muestra cómo implementar un sistema de pausa: cuando el jugador pulsa ESC durante el juego, se pausa la escena principal y se lanza el menú de pausa sobre ella. Al hacer clic en “Continuar”, el menú se cierra y el juego se reanuda desde donde estaba.

## 8.9. Trabajo con imágenes

### 8.9.1. Cargar y mostrar imágenes

Las imágenes son elementos fundamentales en cualquier juego. En Phaser, primero debemos cargarlas en `preload()` y luego añadirlas al juego en `create()`:

```
function preload() {
    // Cargar la imagen con un identificador único
    this.load.image('personaje', 'assets/personaje.png');
    this.load.image('fondo', 'assets/fondo.jpg');
}

function create() {
    // Añadir la imagen al canvas en posición (x, y)
    this.add.image(400, 300, 'fondo');
    let sprite = this.add.image(200, 150, 'personaje');
}
```

Es importante cargar las imágenes en `preload()` para garantizar que estén disponibles antes de intentar usarlas. El identificador que asignamos ('personaje', 'fondo') es la clave que utilizaremos después para referenciar estas imágenes.

### 8.9.2. Sistema de coordenadas y origen

Por defecto, todas las imágenes se posicionan usando su centro como punto de referencia. Podemos cambiar este comportamiento:

```
// Establecer el origen en la esquina superior izquierda
let imagen = this.add.image(100, 100, 'personaje').setOrigin(0, 0);

// Origen en el centro inferior
let imagen2 = this.add.image(200, 200, 'personaje').setOrigin(0.5, 1);
```

Los valores de origen van de 0 a 1, donde (0,0) es la esquina superior izquierda y (1,1) es la inferior derecha de la imagen. Cambiar el origen es útil para alinear objetos o hacer que rotén alrededor de puntos específicos.

### 8.9.3. Transformaciones básicas

```
function create() {
    let jugador = this.add.image(400, 300, 'personaje');

    // Escalar la imagen (0.5 = mitad del tamaño, 2 = doble)
    jugador.setScale(1.5);

    // Escalar de forma independiente en X e Y
    jugador.setScale(2, 0.5);

    // Voltar horizontalmente
    jugador.flipX = true;

    // Voltar verticalmente
    jugador.flipY = true;

    // Rotar (en radianes)
    jugador.rotation = Math.PI / 4; // 45 grados

    // Cambiar profundidad (controla qué se dibuja encima)
    jugador.depth = 10;
}
```

El escalado independiente en X e Y permite crear efectos de estiramiento. La propiedad `depth` funciona como las capas en programas de diseño: valores mayores se dibujan sobre valores menores. Para convertir grados a radianes, recordemos que  $\pi$  radianes = 180 grados.

**Ejemplo práctico:** Sprite que sigue al ratón

```
let jugador;

function create() {
    jugador = this.add.image(400, 300, 'nave');
}

function update() {
```

```

// Obtener posición del ratón
let pointer = this.input.activePointer;

// Mover suavemente hacia el ratón
jugador.x += (pointer.x - jugador.x) * 0.1;
jugador.y += (pointer.y - jugador.y) * 0.1;

// Rotar hacia la dirección del movimiento
let angulo = Phaser.Math.Angle.Between(
    jugador.x, jugador.y,
    pointer.x, pointer.y
);
jugador.rotation = angulo;
}

```

Este ejemplo crea un efecto de seguimiento suave: en lugar de mover el sprite directamente a la posición del ratón, solo se desplaza el 10 % de la distancia en cada frame (multiplicador 0.1). Esto produce un movimiento más natural y fluido. La función `Angle.Between` calcula automáticamente el ángulo necesario para que el sprite “mire” hacia el cursor.

## 8.10. Motores de físicas en Phaser 3

Phaser 3 incluye tres motores de físicas diferentes, cada uno con sus propias características y casos de uso.

### 8.10.1. Arcade Physics

Es el motor de físicas más simple y el que utilizaremos principalmente. Está optimizado para rendimiento y es perfecto para la mayoría de juegos 2D.

**Características:** - Solo soporta colisiones con rectángulos y círculos - Muy eficiente y rápido - Ideal para juegos arcade clásicos, plataformas simples, etc.

```

var config = {
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    }
}

```

```
    }  
};
```

La gravedad se especifica en píxeles por segundo al cuadrado. Un valor de 300 simula una gravedad similar a la terrestre para juegos de plataformas. El modo `debug: true` muestra los contornos de los cuerpos físicos, muy útil durante el desarrollo.

### 8.10.2. Impact Physics

Originalmente creado para el motor Impact, permite físicas más complejas que Arcade.

**Características:** - Soporta pendientes en tiles (muy útil para plataformas) - Más complejo que Arcade pero menos que Matter - Bueno para juegos de plataformas con terrenos inclinados

### 8.10.3. Matter Physics

Es el motor más avanzado y realista de los tres.

**Características:** - Soporta formas complejas y polígonos - Simulación física muy precisa - Mayor costo de rendimiento - Ideal para puzzles físicos, juegos que requieren realismo

### 8.10.4. Añadir físicas a objetos

Para que un objeto tenga físicas, debe crearse usando `physics.add` en lugar de solo `add`:

```
function create() {  
    // Imagen simple sin físicas  
    this.add.image(400, 300, 'fondo');  
  
    // Imagen con físicas  
    this.jugador = this.physics.add.image(100, 450, 'personaje');  
  
    // Configurar propiedades físicas  
    this.jugador.setCollideWorldBounds(true); // No salir del canvas  
    this.jugador.setBounce(0.3); // Rebote (0-1)  
    this.jugador.setVelocity(100, -50); // Velocidad inicial  
    this.jugador.setAcceleration(50, 0); // Aceleración  
}
```

El rebote (`bounce`) determina cuánta energía conserva el objeto al chocar: 0 significa que no rebota, 1 que rebota perfectamente sin perder energía. La velocidad se mide en píxeles por segundo, mientras que la aceleración en píxeles por segundo al cuadrado.

## 8.11. Sistema de entrada (Input)

Phaser 3 proporciona un sistema completo para gestionar diferentes tipos de entrada del usuario.

### 8.11.1. Teclado

#### 8.11.1.1. Detección de teclas individuales

```
function create() {
    // Método 1: Eventos de teclas específicas
    this.input.keyboard.on('keydown-SPACE', () => {
        console.log(';Salto!');
    });

    // Detectar cuando se suelta la tecla
    this.input.keyboard.on('keyup-SPACE', () => {
        console.log('Tecla soltada');
    });
}
```

El evento `keydown` se dispara en el momento exacto que se presiona la tecla, mientras que `keyup` lo hace al soltarla. Esto es útil para acciones instantáneas como saltos o disparos.

#### 8.11.1.2. Uso de cursores

```
let jugador;
let cursors;

function create() {
    jugador = this.physics.add.image(400, 300, 'personaje');

    // Crear objeto de cursores (flechas + espacio + shift)
```

```

        cursors = this.input.keyboard.createCursorKeys();
    }

    function update() {
        // Resetear velocidad
        jugador.setVelocityX(0);

        // Comprobar teclas presionadas
        if (cursors.left.isDown) {
            jugador.setVelocityX(-160);
        } else if (cursors.right.isDown) {
            jugador.setVelocityX(160);
        }

        if (cursors.up.isDown && jugador.body.touching.down) {
            jugador.setVelocityY(-330);
        }
    }
}

```

Es importante resetear la velocidad horizontal en cada frame para que el personaje se detenga inmediatamente al soltar las teclas. La condición `jugador.body.touching.down` evita el salto infinito, permitiendo saltar solo cuando el jugador toca el suelo.

#### 8.11.1.3. Teclas con modificadores

```

function create() {
    this.input.keyboard.on('keydown-A', (event) => {
        if (event.ctrlKey) {
            console.log('CTRL + A: Seleccionar todo');
        } else if (event.altKey) {
            console.log('ALT + A: Acción alternativa');
        } else if (event.shiftKey) {
            console.log('SHIFT + A: Acción mayúscula');
        } else {
            console.log('Solo A');
        }
    });
}

```

Los modificadores permiten crear atajos de teclado complejos. El objeto `event` contiene

propiedades booleanas para cada modificador (`ctrlKey`, `altKey`, `shiftKey`), permitiendo detectar combinaciones de teclas.

#### 8.11.1.4. Combos de teclas

```
function create() {
    // Combo con letras
    let combo1 = this.input.keyboard.createCombo('KONAMI');

    // Combo con keycodes (arriba, arriba, abajo, abajo, izq, der, izq, der)
    let combo2 = this.input.keyboard.createCombo([38, 38, 40, 40, 37, 39, 37, 39]);

    // Detectar cuando se completa el combo
    this.input.keyboard.on('keycombomatch', (combo) => {
        console.log('¡Combo desbloqueado!');
        this.activarModoEspecial();
    });
}
```

Los combos permiten implementar códigos secretos o trucos. El sistema detecta automáticamente cuando el jugador presiona la secuencia correcta de teclas en orden. El segundo ejemplo muestra el famoso código Konami usando los valores numéricos de las teclas de dirección.

### 8.11.2. Ratón

#### 8.11.2.1. Eventos básicos del ratón

```
function create() {
    // Detectar clic
    this.input.on('pointerdown', (pointer) => {
        if (pointer.leftButtonDown()) {
            console.log('Clic izquierdo en:', pointer.x, pointer.y);
        }

        if (pointer.rightButtonDown()) {
            console.log('Clic derecho');
        }
    });
}
```

```

// Detectar cuando se suelta
this.input.on('pointerup', (pointer) => {
    console.log('Botón soltado');
});

// Detectar movimiento
this.input.on('pointermove', (pointer) => {
    console.log('Ratón movido a:', pointer.x, pointer.y);
});
}

```

El sistema de punteros unifica ratón y táctil bajo la misma API. El objeto `pointer` contiene las coordenadas (x, y) y métodos para detectar qué botón se ha pulsado.

## 8.12. Detección de colisiones

### 8.12.1. Conceptos básicos

Phaser 3 Arcade Physics ofrece dos formas de detectar interacciones entre objetos:

**Collider:** Detecta y resuelve colisiones físicamente, separando los objetos y aplicando propiedades como masa, velocidad y rebote.

**Overlap:** Detecta superposición espacial sin resolución física. Los objetos pueden atravesarse.

### 8.12.2. Colisiones básicas

```

function create() {
    let jugador = this.physics.add.image(100, 450, 'personaje');
    let plataforma = this.physics.add.image(400, 500, 'suelo');
    plataforma.setImmovable(true);

    // Colisión con resolución física
    this.physics.add.collider(jugador, plataforma);

    // Overlap sin resolución física
    let estrella = this.physics.add.image(200, 200, 'estrella');
    this.physics.add.overlap(jugador, estrella, (j, e) => {
        e.destroy();
        console.log('|Estrella recogida!');
    });
}

```

```
});  
}
```

La propiedad `setImmovable(true)` hace que la plataforma no se mueva al recibir impactos. Sin esto, el jugador empujaría la plataforma al caer sobre ella. Los colliders son ideales para plataformas y paredes, mientras que los overlaps funcionan bien para objetos colecciónables.

### 8.12.3. Trabajar con grupos

#### 8.12.3.1. Grupos estáticos

Ideales para objetos inmóviles como plataformas:

```
function create() {  
    let jugador = this.physics.add.image(100, 450, 'personaje');  
  
    let plataformas = this.physics.add.staticGroup();  
    plataformas.create(400, 568, 'suelo').setScale(2).refreshBody();  
    plataformas.create(600, 400, 'suelo');  
    plataformas.create(50, 250, 'suelo');  
  
    this.physics.add.collider(jugador, plataformas);  
}
```

Los grupos estáticos son más eficientes para objetos que nunca se moverán. Tras modificar propiedades de un objeto estático (como `setScale`), es crucial llamar a `refreshBody()` para actualizar su cuerpo físico.

#### 8.12.3.2. Grupos dinámicos

Para objetos con físicas completas:

```
function create() {  
    let jugador = this.physics.add.image(100, 450, 'personaje');  
  
    let estrellas = this.physics.add.group({  
        key: 'estrella',  
        repeat: 11,  
        setXY: { x: 12, y: 0, stepX: 70 }  
    });
```

```

        estrellas.children.iterate((estrella) => {
            estrella.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
        });

        this.physics.add.overlap(jugador, estrellas, (j, e) => {
            e.disableBody(true, true);
        });
    }
}

```

Este código crea 12 estrellas (repeat: 11 significa el objeto inicial más 11 repeticiones) espaciadas horizontalmente cada 70 píxeles. La función `iterate` aplica un rebote aleatorio a cada estrella. El método `disableBody(true, true)` oculta y desactiva la estrella sin destruirla completamente, lo que es más eficiente para objetos que pueden reutilizarse.

#### 8.12.4. Callbacks y condiciones

##### 8.12.4.1. Callback de colisión

```

this.physics.add.collider(jugador, enemigo, (j, e) => {
    console.log('¡Colisión!');
    j.setTint(0xff0000);
    j.setVelocityX(-200);
});

```

El callback se ejecuta cada vez que ocurre la colisión. En este ejemplo, el jugador se tiñe de rojo y retrocede al tocar al enemigo. Los parámetros del callback son los dos objetos que colisionaron.

##### 8.12.4.2. Process callback (condición)

El process callback decide si procesar la colisión. Retorna `true` para procesarla, `false` para ignorarla:

```

function create() {
    this.jugador = this.physics.add.image(100, 450, 'personaje');
    this.jugador.invulnerable = false;

    let enemigo = this.physics.add.image(400, 300, 'enemigo');
}

```

```

        this.physics.add.collider(
            this.jugador,
            enemigo,
            this.dañar,
            this.puedeRecibirDaño,
            this
        );
    }

    function puedeRecibirDaño(jugador, enemigo) {
        return !jugador.invulnerable;
    }

    function dañar(jugador, enemigo) {
        jugador.invulnerable = true;
        jugador.setTint(0xff0000);
        this.time.delayedCall(2000, () => {
            jugador.invulnerable = false;
            jugador.clearTint();
        });
    }
}

```

Este sistema implementa un período de invulnerabilidad temporal: tras recibir daño, el jugador se vuelve inmune durante 2 segundos (2000 milisegundos). El process callback `puedeRecibirDaño` se ejecuta antes de la colisión y determina si debe procesarse según el estado de invulnerabilidad.

#### 8.12.5. Colisiones entre grupos

```

function create() {
    this.balas = this.physics.add.group();
    this.enemigos = this.physics.add.group({
        key: 'enemigo',
        repeat: 5,
        setXY: { x: 100, y: 100, stepX: 100 }
    });

    // Colisión grupo contra grupo
    this.physics.add.collider(this.balas, this.enemigos, (bala, enemigo) => {
        bala.destroy();
    });
}

```

```
        enemigo.destroy();
    });
}
```

Las colisiones entre grupos permiten que cualquier miembro de un grupo colisione con cualquier miembro del otro grupo. Aquí, cada bala puede impactar a cualquier enemigo, destruyendo ambos objetos en el proceso. Esto es mucho más eficiente que crear colliders individuales para cada combinación posible.

### 8.12.6. Detectar colisiones específicas

```
function update() {
    // Verificar contacto con superficies
    if (this.jugador.body.touching.down) {
        console.log('En el suelo');
    }

    // Salto solo si está en el suelo
    if (this.cursors.up.isDown && this.jugador.body.touching.down) {
        this.jugador.setVelocityY(-330);
    }
}
```

Propiedades útiles: `touching.down`, `touching.up`, `touching.left`, `touching.right`, `blocked.down`, `blocked.up`. La diferencia entre `touching` y `blocked` es que `touching` detecta contacto con cualquier objeto, mientras que `blocked` solo detecta contacto con objetos inmóviles. Estas propiedades son esenciales para controlar mecánicas como saltos o detectar si un personaje está contra una pared.

## 8.13. Integración con el Patrón Command

Ahora que conocemos los fundamentos de Phaser 3, podemos entender cómo integrar el **Patrón Command** explicado en el documento sobre arquitectura de juegos en red.

### 8.13.1. ¿Por qué usar el Patrón Command con Phaser?

Cuando desarrollamos un juego multijugador con Phaser, necesitamos una forma estructurada de:

1. Capturar las acciones del jugador (input)
2. Ejecutarlas localmente para respuesta inmediata
3. Transmitirlas por red a otros jugadores
4. Reproducir las acciones recibidas desde la red

El Patrón Command resuelve estos problemas encapsulando cada acción como un objeto independiente.

### 8.13.2. Estructura básica del Patrón Command en Phaser

#### 8.13.2.1. Clase base Command

```
class Command {
    execute() {
        // A implementar por subclases
    }

    serialize() {
        // Convertir a JSON para enviar por red
        return {};
    }

    getPlayer() {
        // Retornar la entidad que controla este comando
        return null;
    }
}
```

Esta clase abstracta define la interfaz común para todos los comandos. El método `execute()` realiza la acción en el juego, `serialize()` prepara el comando para transmitirlo por red, y `getPlayer()` identifica qué entidad del juego está asociada al comando.

#### 8.13.2.2. Comando concreto: Mover paleta

```
class MovePaddleCommand extends Command {
    constructor(paddle, direction) {
        super();
        this.paddle = paddle;
        this.direction = direction; // 'up', 'down', 'stop'
```

```

    }

    execute() {
        const speed = 300;
        if (this.direction === 'up') {
            this.paddle.setVelocityY(-speed);
        } else if (this.direction === 'down') {
            this.paddle.setVelocityY(speed);
        } else {
            this.paddle.setVelocityY(0);
        }
    }

    serialize() {
        return {
            type: 'MOVE_PADDLE',
            playerId: this.paddle.id,
            direction: this.direction
        };
    }

    getPlayer() {
        return this.paddle;
    }
}

```

Este comando encapsula la acción de mover una paleta. El constructor recibe la paleta a mover y la dirección deseada. El método `execute()` aplica la velocidad correspondiente según la dirección. La serialización convierte el comando a un formato JSON simple que puede enviarse por red, incluyendo solo los datos esenciales: tipo de comando, identificador del jugador y dirección.

#### 8.13.2.3. CommandProcessor

```

class CommandProcessor {
    constructor() {
        this.players = new Map();
        this.network = null;
    }
}

```

```

setNetwork(networkManager) {
    this.network = networkManager;
}

process(command) {
    const player = command.getPlayer();

    // Solo ejecutar si tenemos autoridad local
    if (player && player.authority === 'LOCAL') {
        command.execute();

        // Transmitir por red si estamos conectados
        if (this.network && this.network.isConnected()) {
            this.network.send(command.serialize());
        }
    }
}

receiveCommand(data) {
    const player = this.players.get(data.playerId);

    // Solo ejecutar si el jugador remoto tiene autoridad
    if (player && player.authority === 'REMOTE') {
        const command = this.deserialize(data, player);
        if (command) {
            command.execute();
        }
    }
}

deserialize(data, player) {
    switch(data.type) {
        case 'MOVE_PADDLE':
            return new MovePaddleCommand(player, data.direction);
        default:
            console.warn('Comando desconocido:', data.type);
            return null;
    }
}
}

```

El **CommandProcessor** es el componente central que coordina comandos locales y remotos.

El método `process()` ejecuta comandos del jugador local y los envía por red, mientras que `receiveCommand()` procesa comandos recibidos de otros jugadores. El sistema de autoridad (`LOCAL` vs `REMOTE`) evita que un jugador ejecute dos veces el mismo comando: los comandos locales se ejecutan inmediatamente y se transmiten, mientras que los comandos remotos solo se ejecutan cuando se reciben por red. La función `deserialize()` reconstruye objetos `Command` a partir de los datos JSON recibidos.

### 8.13.3. Implementación en una escena de Phaser

```
class GameScene extends Phaser.Scene {
    constructor() {
        super('GameScene');
        this.commandProcessor = new CommandProcessor();
    }

    create() {
        // Crear paletas con autoridad
        this.localPaddle = this.physics.add.image(50, 300, 'paddle');
        this.localPaddle.id = 'player1';
        this.localPaddle.authority = 'LOCAL';
        this.localPaddle.setCollideWorldBounds(true);

        this.remotePaddle = this.physics.add.image(750, 300, 'paddle');
        this.remotePaddle.id = 'player2';
        this.remotePaddle.authority = 'REMOTE';
        this.remotePaddle.setCollideWorldBounds(true);

        // Registrar jugadores en el procesador
        this.commandProcessor.players.set('player1', this.localPaddle);
        this.commandProcessor.players.set('player2', this.remotePaddle);

        // Configurar input
        this.cursors = this.input.keyboard.createCursorKeys();
    }

    update() {
        // Crear comandos basados en input
        if (this.cursors.up.isDown) {
            const command = new MovePaddleCommand(this.localPaddle, 'up');
            this.commandProcessor.process(command);
        } else if (this.cursors.down.isDown) {
```

```

        const command = new MovePaddleCommand(this.localPaddle, 'down');
        this.commandProcessor.process(command);
    } else {
        const command = new MovePaddleCommand(this.localPaddle, 'stop');
        this.commandProcessor.process(command);
    }
}

```

Esta implementación muestra cómo integrar el patrón en una escena real de Phaser. En `create()` se definen dos paletas: una local (controlada por este jugador) y una remota (controlada por otro jugador a través de la red). La asignación de autoridad es crucial para evitar conflictos. En `update()`, cada acción del teclado se convierte en un comando que se procesa inmediatamente, proporcionando respuesta instantánea al jugador local mientras se sincroniza automáticamente con otros jugadores a través de la red.

#### **8.13.4. Ventajas de esta arquitectura**

**Separación de responsabilidades:** La lógica de input, ejecución del juego y networking están claramente separadas.

**Testabilidad:** Podemos probar comandos individuales sin necesidad de tener un juego completo funcionando.

**Flexibilidad:** Es fácil añadir nuevos tipos de comandos sin modificar el código existente.

**Preparado para networking:** La misma estructura funciona para juego local, REST API o WebSockets, solo cambiando la implementación del NetworkManager.

#### **8.13.5. Ventajas del Patrón Command con Phaser 3**

**Transparencia de red:** El mismo comando funciona igual si se ejecuta localmente o se recibe de la red.

**Fácil debugging:** Podemos registrar todos los comandos ejecutados para análisis o replay.

**Testing simplificado:** Podemos probar comandos sin necesidad de input real.

**Predicción del lado del cliente:** Podemos ejecutar comandos inmediatamente para respuesta rápida, y luego reconciliar con el servidor.

## **Parte III**

# **Desarrollo en el servidor y comunicación**

## References

- Analytics, IoT. 2020. «Internet of Things (IoT) and non-IoT active device connections worldwide from 2010 to 2025 (in billions)». Statista. <https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/>.
- BBC Brasil. 2019. «[Image from: Article Title]». <https://www.bbc.com/portuguese/geral-50162526>.
- Claypool, Mark, y Kajal Claypool. 2006. «Latency and player actions in online games». *Commun. ACM* 49 (11): 40-45. <https://doi.org/10.1145/1167838.1167860>.
- Kurose, James F., y Keith W. Ross. 2017. *Computer Networks: A Top-Down Approach*. 7.<sup>a</sup> ed. Boston, MA: Pearson.
- Ritchie, Hannah, Edouard Mathieu, Max Roser, y Esteban Ortiz-Ospina. 2023. «Internet». Our World in Data. <https://ourworldindata.org/internet>.