

# Juegos en Red

Grado en Desarrollo de Videojuegos:

- Guía de estudio
- Notas de la asignatura
- Diapositivas de la asignatura
- Ejercicios de la asignatura
- Códigos de la asignatura

Material docente en abierto de la Universidad Rey Juan Carlos

## AUTORES

- Rubén Rodríguez Fernández
- Natalia Madrueño Sierro

2025-2026



Copyright © 2025 Rubén Rodríguez Fernández, Natalia Madrueño Sierro. Esta obra está licenciada bajo CC BY-SA 4.0, Creative Commons Atribución-Compartir Igual 4.0 Internacional.

# Tabla de contenidos

---

<b>Guía de estudio</b>	<b>8</b>
• Objetivos de la asignatura	8
• Temario de la asignatura	9
• Desarrollo de la asignatura	10
• Recursos adicionales	22
• Bibliografía recomendada	22
<b>Parte: Introducción a los juegos en red y a las redes de comunicaciones</b>	<b>23</b>
<b>1. Introducción a las Redes de Ordenadores</b>	<b>24</b>
• 1.1. Introducción	24
• 1.2. La Historia de Internet	29
• 1.3. Infraestructura de la red y tecnologías de transmisión	33
• 1.4. Modelos de Referencia de Redes	40
• 1.5. Rendimiento	44
<b>2. Capa de Acceso a la Red</b>	<b>49</b>
• 2.1. Funciones principales de la Capa de Acceso a la Red	50
• 2.2. Dispositivos de la Capa de Acceso a la Red	54
• 2.3. Protocolos	56
<b>3. Capa de red</b>	<b>61</b>
• 3.1. Funciones Fundamentales de la Capa de Red	63

• 3.2. Modelos de servicio	64
• 3.3. Dispositivos físicos de la Capa de Red	66
• 3.4. Protocolos	69
<b>4. Capa de transporte</b>	<b>78</b>
• 4.1. Funciones principales	79
• 4.2. Protocolos	81
• 4.3. Comparativa de TCP vs UDP para videojuegos	93
<b>5. Capa de aplicación</b>	<b>95</b>
• 5.1. Socket	98
• 5.2. Arquitecturas de Aplicaciones Distribuidas	108
• 5.3. Protocolos	115
• 5.4. Servicios	122
<b>Desarrollo en el lado del cliente</b>	<b>123</b>
<b>6. JavaScript</b>	<b>124</b>
• 6.1. Introducción	124
• 6.2. Configuración del Entorno de Desarrollo con Node.js	127
• 6.3. El Lenguaje JavaScript	136
• 6.4. Integración con HTML	138
• 6.5. Sintaxis Básica	139
• 6.6. Arrays	143
• 6.7. Sentencias de Control de Flujo	145
• 6.8. Funciones	147
• 6.9. Manejo de Excepciones	150
• 6.10. Almacenamiento de Datos en el Navegador	150
<b>7. JavaScript Orientado a Objetos</b>	<b>158</b>
• 7.1. Introducción	158

• 7.2. Orientación a Objetos en JavaScript	158
<b>8. HTML y CSS</b>	<b>168</b>
• 8.1. Introducción	168
• 8.2. HTML (HyperText Markup Language)	168
• 8.3. CSS (Cascading Style Sheets)	176
• 8.4. Integración HTML, CSS y JavaScript	189
<b>Desarrollo de juegos con tecnología web</b>	<b>191</b>
<b>9. Phaser 3</b>	<b>192</b>
• 9.1. Características principales	192
• 9.2. Requisitos para empezar	192
• 9.3. Estructura básica de un juego en Phaser 3	193
• 9.4. Gestión de escenas	200
• 9.5. Trabajo con imágenes	205
• 9.6. Motores de físicas en Phaser 3	208
• 9.7. Sistema de entrada (Input)	211
• 9.8. Detección de colisiones	214
<b>Desarrollo en el lado del servidor</b>	<b>220</b>
<b>10. API REST</b>	<b>221</b>
• 10.1. Introducción	221
• 10.2. ¿Qué es una API REST?	222
• 10.3. Casos de Uso de APIs REST	223
• 10.4. Formato JSON	223
• 10.5. Funcionamiento de un Servicio REST	224
• 10.6. Ventajas de esta Arquitectura	227

<b>11. Cliente API REST con Javascript</b>	228
• 11.1. GET - Obtener datos	229
• 11.2. POST - Crear recurso	229
• 11.3. PUT - Actualizar recurso	230
• 11.4. DELETE - Eliminar recurso	231
• 11.5. Manejo de errores y códigos de estado	232
<b>12. Servidor API REST con Javascript</b>	233
• 12.1. Controladores	234
• 12.2. Rutas	235
• 12.3. Operaciones CRUD	236
• 12.4. Middlewares en Express	238
• 12.5. Servir archivos estáticos	239
<b>Comunicación asíncrona cliente - servidor</b>	241
<b>13. Introducción a WebSockets</b>	242
• 13.1. Introducción a los Sockets	242
• 13.2. ¿Qué son los WebSockets?	242
• 13.3. WebSockets como Estándar	243
• 13.4. Negociación de Conexión WebSocket	243
• 13.5. Uso Normal de WebSockets	244
<b>14. Cliente WebSockets con JavaScript</b>	246
• 14.1. La API WebSocket en JavaScript	246
• 14.2. Métodos del Objeto WebSocket	246
• 14.3. Event Listeners - La Clave de WebSockets	247
<b>15. Servidor WebSockets con JavaScript</b>	251
• 15.1. Servidor WebSocket con Node.js	251
• 15.2. Ejemplo completo: Servidor Echo	254

• 15.3. Ejemplo completo: Servidor de Chat con JSON	255
• 15.4. Integración con Express	257
<b>Referencias</b>	<b>259</b>
<b>16. Diapositivas de la asignatura</b>	<b>261</b>
• 16.1. Introducción a la asignatura	261
• 16.2. Introducción a las Redes de Ordenadores	280
• 16.3. Capa de Acceso	299
• 16.4. Capa de Red	313
• 16.5. Capa de Transporte	331
• 16.6. Capa de Aplicación	346
• 16.7. JavaScript	363
• 16.8. JavaScript OOP	390
• 16.9. Phaser	409
• 16.10. APIs REST	443
• 16.11. WebSockets	479
<b>17. Ejercicios de la asignatura</b>	<b>502</b>
• 17.1. Introducción a Redes	502
• 17.2. Capa de Acceso	503
• 17.3. Capa de Red	504
• 17.4. Capa de Transporte	505
• 17.5. Capa de Aplicación	506
• 17.6. JavaScript	508
• 17.7. Phaser	509
• 17.8. REST - Introducción	510
• 17.9. REST - Cliente	511
• 17.10. REST - Servidor	513
• 17.11. WebSockets - Introducción	515
• 17.12. WebSockets - Cliente	516

• 17.13. WebSockets - Servidor	518
<b>18. Software utilizado en la asignatura</b>	<b>521</b>
• 18.1. Visualizaciones	521

# Guía de estudio

---

Los videojuegos en red representan uno de los sectores más dinámicos y desafiantes de la industria del software, donde la capacidad de diseñar e implementar sistemas de comunicación eficientes y robustos es fundamental. En este contexto, comprender las arquitecturas de red, los protocolos de comunicación y las técnicas de programación tanto del lado del cliente como del servidor se ha convertido en una habilidad esencial para cualquier desarrollador de juegos modernos. Este libro está diseñado para proporcionar una comprensión profunda y práctica de estas tecnologías, basándose en el contenido de la asignatura **Juegos en Red**.

El desarrollo de juegos en red tiene múltiples facetas que exploraremos en profundidad: desde los **fundamentos de las redes de comunicación** hasta la **programación en el cliente** con JavaScript y Phaser, y finalmente la **comunicación cliente-servidor** mediante APIs REST y WebSockets. Entender esta progresión es clave para desarrollar juegos multijugador escalables y eficientes.

Comenzaremos sentando las bases de las redes de ordenadores, explorando las diferentes capas del modelo TCP/IP. A partir de ahí, aprenderemos a desarrollar aplicaciones web interactivas usando JavaScript y el motor de juegos Phaser. Finalmente, dominaremos las técnicas de comunicación cliente-servidor mediante APIs REST para operaciones síncronas y WebSockets para comunicación en tiempo real.

## Note

La primera parte de este libro sobre redes de ordenadores está basada en los conceptos presentados en Computer Networks: A Top-Down Approach (Kurose y Ross 2021), adaptados al contexto específico del desarrollo de videojuegos en red.

## Objetivos de la asignatura

- Comprender los fundamentos de las redes de ordenadores y los protocolos de comunicación del modelo TCP/IP.
- Dominar el lenguaje JavaScript para el desarrollo de aplicaciones web interactivas.

- Aprender a utilizar el motor de juegos Phaser para crear juegos 2D en el navegador.
- Diseñar e implementar APIs REST para la comunicación cliente-servidor.
- Desarrollar sistemas de comunicación en tiempo real utilizando WebSockets.
- Integrar todas estas tecnologías para crear juegos multijugador funcionales.

## Temario de la asignatura

La asignatura está dividida en 5 partes, que se corresponden con la categorización de la guía docente. Además, cada uno de estas partes está dividida en diferentes capítulos, con el fin de facilitar su lectura.

---

### Parte 1: Introducción a los juegos en red y a las redes de comunicaciones

- **Capítulo 1:** Introducción a las Redes de Ordenadores
  - **Capítulo 2:** Capa de Acceso a la Red
  - **Capítulo 3:** Capa de Red
  - **Capítulo 4:** Capa de Transporte
  - **Capítulo 5:** Capa de Aplicación
- 

### Parte 2: Desarrollo en el lado del cliente

- **Capítulo 6:** JavaScript
  - **Capítulo 7:** HTML y CSS
- 

### Parte 3: Desarrollo de juegos con tecnología web

- **Capítulo 8:** Phaser - Motor de Juegos 2D
- 

### Parte 4: Desarrollo en el lado del servidor

- **Capítulo 9:** APIs REST - Introducción, Cliente y Servidor

---

## Parte 5: Comunicación asíncrona cliente - servidor

- **Capítulo 10:** WebSockets - Introducción, Cliente y Servidor

## Desarrollo de la asignatura

El curso se desarrolla a lo largo de 15 semanas, con dos sesiones semanales. La metodología busca integrar la teoría y la práctica de forma fluida: los conceptos teóricos se presentarán junto con su implementación práctica, fomentando un aprendizaje aplicado en cada clase.

---

## Calendario de clases

Semana	Dia 1	Dia 2
1	Clase 1: Introducción a asignatura y a las Redes de Ordenadores	Clase 2: Capa de Acceso a la Red
2	Clase 3: Capa de Red	Clase 4: Capa de Transporte
3	Clase 5: Capa de Aplicación	Clase 6: Repaso y Prácticas de Redes
4	Clase 7: Repaso y Prácticas de Redes	Clase 8: Presentación e Introducción a JavaScript
5	Clase 9: Examen Parcial	Clase 10: JavaScript Orientado a Objetos
6	Clase 11: Introducción a Phaser	Clase 12: Phaser (continuación)
7	Clase 13: Phaser (continuación)	Clase 14: Phaser (continuación)
8	Clase 15: Phaser (continuación)	Clase 16: APIs REST - Introducción y Cliente
9	Clase 17: APIs REST - Cliente (continuación)	Clase 18: Presentación prácticas

Semana	Dia 1	Dia 2
10	Clase 19: Examen	Clase 20: APIs REST - Servidor
11	Clase 21: APIs REST - Servidor (continuación)	Clase 22: APIs REST - Servidor (continuación)
12	Clase 23: APIs REST - Servidor (continuación)	Clase 24: WebSockets - Cliente y Servidor
13	Clase 25: WebSockets (continuación)	Clase 26: WebSockets (continuación)
14	Clase 27: WebSockets (continuación)	Clase 28: WebSockets (continuación)
15	Clase 29: Examen	Clase 30: Presentación prácticas

## Clase 1: Presentación a asignatura e introducción a las Redes de Ordenadores

- **Contenidos:** Presentación de la asignatura, guía docente y sistema de evaluación. Introducción de los conceptos fundamentales de redes de ordenadores. Arquitectura de red por capas. Modelo TCP/IP. Protocolos y estándares. Historia y evolución de Internet.
- **Objetivos de Aprendizaje:**
  - Comprender la arquitectura en capas de las redes de ordenadores.
  - Identificar los componentes fundamentales de Internet.
  - Entender el papel de los protocolos en la comunicación de red.
- **Materiales Utilizados:**
  - Apuntes: [Capítulo 1](#)
  - Diapositivas: [Sección 16.1](#) [Sección 16.2](#)
  - Ejercicios: [Sección 17.1](#)
- **Actividades Planificadas:**
  - Discusión sobre la importancia de las redes en los videojuegos.
  - Análisis de la arquitectura de red de juegos multijugador populares.

- **Trabajo Personal Recomendado:**

- Lectura completa de los apuntes de introducción a redes.
  - Realización del ejercicio práctico de trazado de rutas.
- 

## Clase 2: Capa de Acceso a la Red

- **Contenidos:** Funciones de la capa de acceso. Tecnologías de acceso físico. Ethernet. WiFi. Direccionamiento MAC. Comutación y dominios de colisión.

- **Objetivos de Aprendizaje:**

- Comprender las funciones de la capa de acceso a la red.
- Identificar las diferencias entre distintas tecnologías de acceso.
- Entender el funcionamiento de las direcciones MAC.

- **Materiales Utilizados:**

- Apuntes: [Capítulo 2](#)
- Diapositivas: [Sección 16.3](#)
- Ejercicios: [Sección 17.2](#)

- **Actividades Planificadas:**

- Análisis de tramas Ethernet.
- Configuración de redes locales.

- **Trabajo Personal Recomendado:**

- Lectura de los apuntes sobre la capa de acceso.
  - Realización del ejercicio práctico.
- 

## Clase 3: Capa de Red

- **Contenidos:** Protocolo IP. Direccionamiento IPv4 e IPv6. Subredes y máscaras. Enrutamiento. Protocolo ARP. ICMP y herramientas de diagnóstico (ping, traceroute).

- **Objetivos de Aprendizaje:**

- Comprender el funcionamiento del protocolo IP.
- Dominar el direccionamiento IP y el concepto de subredes.
- Utilizar herramientas de diagnóstico de red.

• **Materiales Utilizados:**

- Apuntes: [Capítulo 3](#)
- Diapositivas: [Sección 16.4](#)
- Ejercicios: [Sección 17.3](#)

• **Actividades Planificadas:**

- Prácticas con herramientas de diagnóstico (ping, traceroute).
- Análisis de tablas de enrutamiento.

• **Trabajo Personal Recomendado:**

- Lectura de los apuntes sobre la capa de red.
  - Realización del ejercicio de direccionamiento IP.
- 

## Clase 4: Capa de Transporte

• **Contenidos:** Protocolo TCP. Protocolo UDP. Control de flujo y congestión. Establecimiento y cierre de conexiones. Puertos y sockets. Comparación TCP vs UDP para videojuegos.

• **Objetivos de Aprendizaje:**

- Comprender las diferencias entre TCP y UDP.
- Entender cuándo utilizar cada protocolo en el desarrollo de juegos.
- Dominar el concepto de puertos y sockets.

• **Materiales Utilizados:**

- Apuntes: [Capítulo 4](#)
- Diapositivas: [Sección 16.5](#)
- Ejercicios: [Sección 17.4](#)

• **Actividades Planificadas:**

- Análisis del tráfico TCP y UDP.
- Discusión sobre las necesidades de latencia en juegos multijugador.

• **Trabajo Personal Recomendado:**

- Lectura de los apuntes sobre la capa de transporte.
- Realización del ejercicio práctico.

---

## Clase 5: Capa de Aplicación

- **Contenidos:** Protocolos de aplicación. HTTP y HTTPS. DNS. Arquitecturas cliente-servidor y peer-to-peer. APIs web. Introducción a los servicios web.
- **Objetivos de Aprendizaje:**
  - Comprender los protocolos de la capa de aplicación.
  - Entender el funcionamiento de HTTP/HTTPS.
  - Conocer diferentes arquitecturas de aplicación en red.
- **Materiales Utilizados:**
  - Apuntes: [Capítulo 5](#)
  - Diapositivas: [Sección 16.6](#)
  - Ejercicios: [Sección 17.5](#)
- **Actividades Planificadas:**
  - Análisis de peticiones HTTP.
  - Exploración de APIs públicas.
- **Trabajo Personal Recomendado:**
  - Lectura de los apuntes sobre la capa de aplicación.
  - Realización del ejercicio de análisis de tráfico HTTP.

---

## Clase 6-7: Repaso y Prácticas de Redes

- **Contenidos:** Repaso integral de los conceptos de redes de ordenadores. Resolución de ejercicios prácticos. Preparación para la siguiente fase del curso.
- **Objetivos de Aprendizaje:**
  - Consolidar los conocimientos de todas las capas del modelo TCP/IP.
  - Integrar los conceptos en casos prácticos.
- **Materiales Utilizados:**
  - Todos los materiales de las semanas 1-5.
  - Ejercicios de repaso.
- **Actividades Planificadas:**
  - Sesión de resolución de dudas.
  - Prácticas guiadas de configuración y análisis de redes.

- **Trabajo Personal Recomendado:**
    - Repaso de todos los capítulos de redes.
    - Preparación del entorno de desarrollo para JavaScript.
- 

## Clase 8: Presentación e Introducción a JavaScript

- **Contenidos:** Presentación de la segunda parte del curso. Fundamentos de JavaScript: variables, tipos de datos, operadores, estructuras de control. Funciones y ámbito.
- **Objetivos de Aprendizaje:**
  - Comprender la sintaxis básica de JavaScript.
  - Dominar las estructuras de control de flujo.
  - Crear y utilizar funciones en JavaScript.
- **Materiales Utilizados:**
  - Apuntes: [Capítulo 6](#)
  - Diapositivas: [Sección 16.7](#)
  - Ejercicios: [Sección 17.6](#)
- **Actividades Planificadas:**
  - Configuración del entorno de desarrollo.
  - Primeros programas en JavaScript.
- **Trabajo Personal Recomendado:**
  - Lectura de las secciones introductorias de JavaScript.
  - Práctica con ejercicios básicos de programación.

---

## Clase 9: Examen Parcial

- **Contenidos:** Evaluación de los contenidos de redes de ordenadores (Clases 1-7).
- **Objetivos de Aprendizaje:**
  - Demostrar comprensión de los fundamentos de redes.
  - Aplicar los conocimientos a problemas prácticos.
- **Materiales Utilizados:**
  - Todos los materiales de las clases 1-7.

- **Actividades Planificadas:**
  - Examen parcial teórico-práctico.

- **Trabajo Personal Recomendado:**
    - Continuar con el aprendizaje de JavaScript.
- 

## Clase 10: JavaScript Orientado a Objetos

- **Contenidos:** Objetos y arrays. Programación orientada a objetos en JavaScript. Clases y prototipos. Manipulación del DOM. Eventos.

- **Objetivos de Aprendizaje:**
  - Dominar el trabajo con objetos y arrays.
  - Comprender la programación orientada a objetos en JavaScript.
  - Manipular el DOM y gestionar eventos.

- **Materiales Utilizados:**
  - Apuntes: [Capítulo 6](#)
  - Diapositivas: [Sección 16.8](#)
  - Ejercicios: [Sección 17.6](#)

- **Actividades Planificadas:**
  - Creación de aplicaciones interactivas simples.
  - Prácticas con eventos del navegador.

- **Trabajo Personal Recomendado:**
    - Lectura de las secciones avanzadas de JavaScript.
    - Desarrollo de pequeños proyectos personales.
- 

## Clase 11-15: Introducción a Phaser

- **Contenidos:** Introducción al motor de juegos Phaser. Escenas y ciclo de vida. Sprites y texturas. Sistema de físicas. Gestión de entrada (teclado, ratón, táctil).

- **Objetivos de Aprendizaje:**
  - Comprender la arquitectura de Phaser.
  - Crear escenas y gestionar el ciclo de vida del juego.
  - Implementar físicas básicas y controles.

- **Materiales Utilizados:**

- Apuntes: [Capítulo 9](#)
- Diapositivas: [Sección 16.9](#)
- Ejercicios: [Sección 17.7](#)

- **Actividades Planificadas:**

- Desarrollo de un proyecto de juego completo.
- Sesión de depuración y optimización.

- **Trabajo Personal Recomendado:**

- Finalización del proyecto de juego.
- Lectura de [Capítulo 8](#) para crear la base web del juego.
- Preparación para la integración con servicios de red.

---

## Clase 16-17: APIs REST - Introducción y Cliente

- **Contenidos:** Arquitectura REST. Métodos HTTP (GET, POST, PUT, DELETE). Formato JSON. Consumo de APIs REST desde JavaScript. Fetch API. Manejo de promesas y async/await.

- **Objetivos de Aprendizaje:**

- Comprender los principios de la arquitectura REST.
- Consumir APIs REST desde el cliente.
- Manejar operaciones asíncronas en JavaScript.

- **Materiales Utilizados:**

- Apuntes: [Capítulo 10](#), [Capítulo 11](#)
- Diapositivas: [Sección 16.10](#)
- Ejercicios: [Sección 17.8](#), [Sección 17.9](#)

- **Actividades Planificadas:**

- Consumo de APIs públicas.
- Integración de datos externos en aplicaciones.

- **Trabajo Personal Recomendado:**

- Lectura de los apuntes sobre REST.
- Realización de los ejercicios de consumo de APIs.

---

## Clase 18: Presentación prácticas

- **Contenidos:** Presentación del proyecto de juego desarrollado con Phaser. Demostración de funcionalidades implementadas. Revisión del código y arquitectura del juego.
  - **Objetivos de Aprendizaje:**
    - Presentar de forma efectiva un proyecto de desarrollo de juegos.
    - Explicar las decisiones de diseño y arquitectura del juego.
    - Demostrar dominio de JavaScript y Phaser.
  - **Materiales Utilizados:**
    - Proyecto personal de juego con Phaser.
    - Código fuente del proyecto.
  - **Actividades Planificadas:**
    - Presentaciones de los proyectos de juego.
    - Sesión de preguntas y respuestas sobre las implementaciones.
  - **Trabajo Personal Recomendado:**
    - Preparación de la presentación del proyecto.
    - Revisión y refinamiento del código del juego.
    - Estudio para el examen de JavaScript y Phaser.
- 

## Clase 19: Examen

- **Contenidos:** Evaluación de los contenidos de JavaScript y Phaser (Clases 8-15).
- **Objetivos de Aprendizaje:**
  - Demostrar dominio de JavaScript y programación orientada a objetos.
  - Aplicar los conocimientos de Phaser en problemas prácticos.
  - Resolver ejercicios de desarrollo de juegos.
- **Materiales Utilizados:**
  - Todos los materiales de las clases 8-17.
  - Apuntes de JavaScript y Phaser.
- **Actividades Planificadas:**
  - Examen teórico-práctico de JavaScript y Phaser.

- **Trabajo Personal Recomendado:**

- Continuar con el aprendizaje de APIs REST.
  - Lectura de los apuntes sobre desarrollo de servidores.
- 

## Clase 20-23: APIs REST - Servidor

- **Contenidos:** Desarrollo de APIs REST con Node.js y Express. Diseño de endpoints.

Middleware. Gestión de errores. Validación de datos. Persistencia de datos.

- **Objetivos de Aprendizaje:**

- Diseñar e implementar una API REST.
- Gestionar peticiones y respuestas HTTP.
- Implementar la lógica de negocio del servidor.

- **Materiales Utilizados:**

- Apuntes: [Capítulo 12](#)
- Diapositivas: [Sección 16.10](#)
- Ejercicios: [Sección 17.10](#)

- **Actividades Planificadas:**

- Desarrollo de una API REST completa.
- Pruebas de endpoints con herramientas como Postman.

- **Trabajo Personal Recomendado:**

- Lectura de los apuntes sobre desarrollo de servidores REST.
  - Implementación de la API del proyecto final.
- 

## Clase 24-28: WebSockets - Cliente y Servidor

- **Contenidos:** Introducción a WebSockets. Comunicación bidireccional en tiempo real. Socket.IO. Gestión de conexiones. Salas y broadcast. Sincronización de estado en juegos multijugador.

- **Objetivos de Aprendizaje:**

- Comprender las ventajas de WebSockets sobre HTTP.
- Implementar comunicación en tiempo real.
- Sincronizar el estado del juego entre múltiples clientes.

- **Materiales Utilizados:**

- Apuntes: [Capítulo 13](#), [Capítulo 14](#), [Capítulo 15](#)
- Diapositivas: [Sección 16.11](#)
- Ejercicios: [Sección 17.11](#), [Sección 17.12](#), [Sección 17.13](#)

- **Actividades Planificadas:**

- Implementación de chat en tiempo real.
- Desarrollo de la comunicación multijugador para el proyecto final.

- **Trabajo Personal Recomendado:**

- Lectura de todos los apuntes sobre WebSockets.
- Integración de WebSockets en el proyecto final.

---

## Clase 29: Examen

- **Contenidos:** Evaluación de los contenidos de APIs REST y WebSockets (Clases 16-17, 20-28).

- **Objetivos de Aprendizaje:**

- Demostrar comprensión de la arquitectura REST y WebSockets.
- Aplicar los conocimientos de comunicación cliente-servidor.
- Implementar soluciones de comunicación en tiempo real.

- **Materiales Utilizados:**

- Todos los materiales de las clases 16-17 y 20-28.
- Apuntes de REST y WebSockets.

- **Actividades Planificadas:**

- Examen teórico-práctico de REST y WebSockets.

- **Trabajo Personal Recomendado:**

- Finalización del proyecto final con comunicación en tiempo real.
- Preparación de la presentación final del proyecto.

---

## Clase 30: Presentación prácticas

- **Contenidos:** Presentación del proyecto de juego multijugador con API REST. Demostración de la integración cliente-servidor. Arquitectura del sistema completo.

**• Objetivos de Aprendizaje:**

- Presentar un proyecto completo de juego en red con API REST.
- Explicar la arquitectura cliente-servidor implementada.
- Demostrar dominio de todas las tecnologías del curso.

**• Materiales Utilizados:**

- Proyecto final de juego multijugador.
- Código fuente del cliente y servidor.
- Documentación de la API REST implementada.

**• Actividades Planificadas:**

- Presentaciones de los proyectos finales.
- Demostración en vivo de las funcionalidades multijugador.
- Sesión de retroalimentación y evaluación.

**• Trabajo Personal Recomendado:**

- Preparación exhaustiva de la presentación final.
- Documentación completa del proyecto.
- Estudio para el examen de la evaluación ordinaria.

---

## Examen de la evaluación ordinaria (Proyecto final)

**• Contenidos:** Desarrollo de un juego multijugador completo con WebSockets.

**• Objetivos de Aprendizaje:**

- Demostrar comprensión integral de redes de ordenadores, JavaScript, Phaser, APIs REST y WebSockets.
- Implementar un juego multijugador funcional con comunicación en tiempo real.
- Aplicar todos los conocimientos adquiridos durante el curso.

**• Materiales Utilizados:**

- Todos los materiales del curso (Clases 1-30).
- Apuntes completos de redes, JavaScript, Phaser, REST y WebSockets.
- Documentación de todas las tecnologías utilizadas.

**• Actividades Planificadas:**

- Desarrollo y presentación final de un juego multijugador con WebSockets.

- **Trabajo Personal Recomendado:**

- Repaso exhaustivo de todos los contenidos de la asignatura.
- Práctica con proyectos personales integrando todas las tecnologías.
- Revisión de todos los ejercicios y proyectos del curso.

## Recursos adicionales

Los siguientes recursos complementarios pueden ser útiles durante el curso:

- Documentación oficial de JavaScript (Mozilla Developer Network 2024): [MDN Web Docs](#)
- Documentación oficial de Phaser (Photonstorm 2024): [Phaser.io](#)
- Documentación de Node.js: [Node.js Docs](#)
- Documentación de Socket.IO: [Socket.IO](#)
- Plataforma con visualizaciones: [JER Games](#) (Para información sobre el código fuente ver [Capítulo 18](#))

## Bibliografía recomendada

Para profundizar en los conceptos del curso, se recomienda consultar:

- **Redes de ordenadores:** Los fundamentos de redes están basados en (Kurose y Ross 2021). También es recomendable (Tanenbaum y Wetherall 2021).
- **JavaScript:** Para dominar el lenguaje, consultar (Flanagan 2020) y (Haverbeke 2018).
- **Desarrollo de juegos en red:** Recursos esenciales incluyen (Gambetta 2014), (Fiedler 2024), (Claypool y Claypool 2006), (Bernier 2001), (Aldridge 2011), y (Glazer y Madhav 2015).
- **APIs REST:** La arquitectura REST está definida en (Fielding 2000). Guías prácticas incluyen (Masse 2011), (Richardson y Ruby 2013), y (Sturgeon 2016).
- **WebSockets:** El protocolo está especificado en (Fette y Melnikov 2011). Para implementación práctica, ver (Wang, Salim, y Moskovits 2013) y (Doglio 2015).
- **Node.js y Express:** Referencias recomendadas incluyen (Wilson 2018), (Hahn 2014), y (Young, Meck, y Cantelon 2017).

Todas las referencias completas se encuentran en la sección de referencias del libro.

## **Parte: Introducción a los juegos en red y a las redes de comunicaciones**

---

Los videojuegos en red representan uno de los sectores más dinámicos de la industria del software, donde comprender las redes de comunicación es fundamental para desarrollar sistemas eficientes y robustos. Esta primera parte del curso establece los cimientos necesarios explorando el modelo TCP/IP capa por capa: desde la capa de acceso a la red con tecnologías como Ethernet y WiFi, pasando por la capa de red con el protocolo IP y el direccionamiento, la capa de transporte con TCP y UDP, hasta llegar a la capa de aplicación con protocolos como HTTP y DNS. Comprenderemos cómo funcionan las comunicaciones en Internet, analizaremos las diferencias entre TCP y UDP en el contexto de los videojuegos, y dominaremos herramientas de diagnóstico esenciales. Estos conocimientos son imprescindibles para tomar decisiones arquitectónicas acertadas al diseñar juegos multijugador.

# **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.

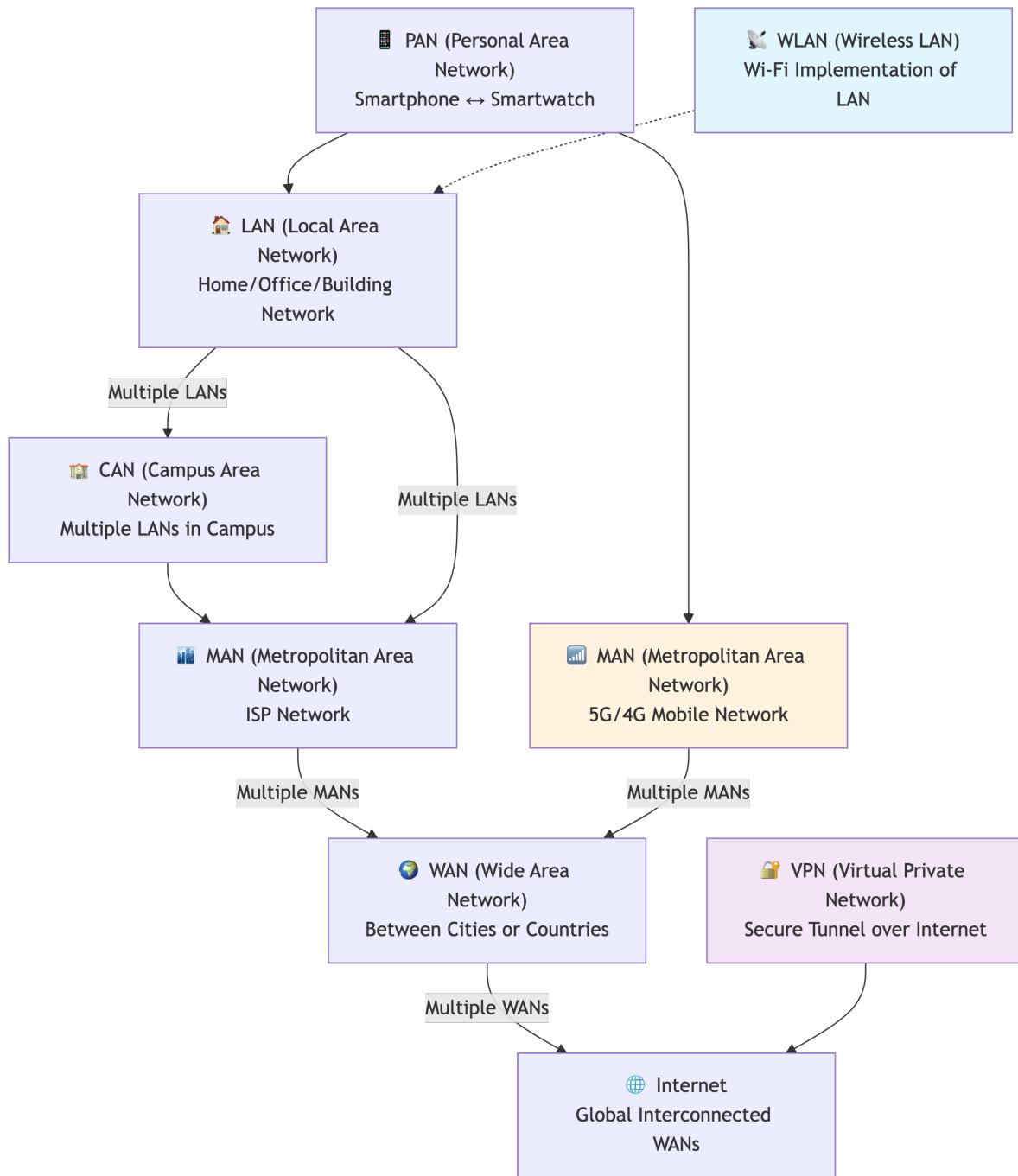


Figura 1.1: Topología jerárquica de Internet.

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 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 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 a lo largo de esta introducción.

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 definen 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 conmutación 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.

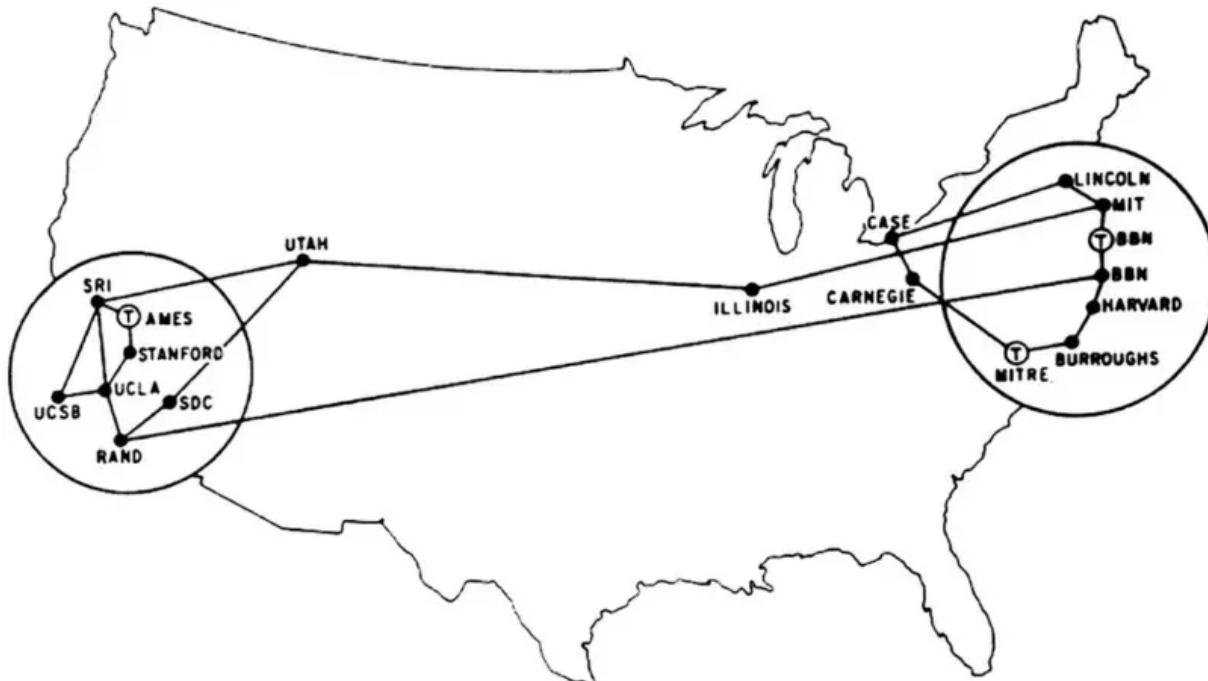


Figura 1.2: Distribución geográfica de los nodos de ARPANET en 1971 (BBC Brasil 2019).

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 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.

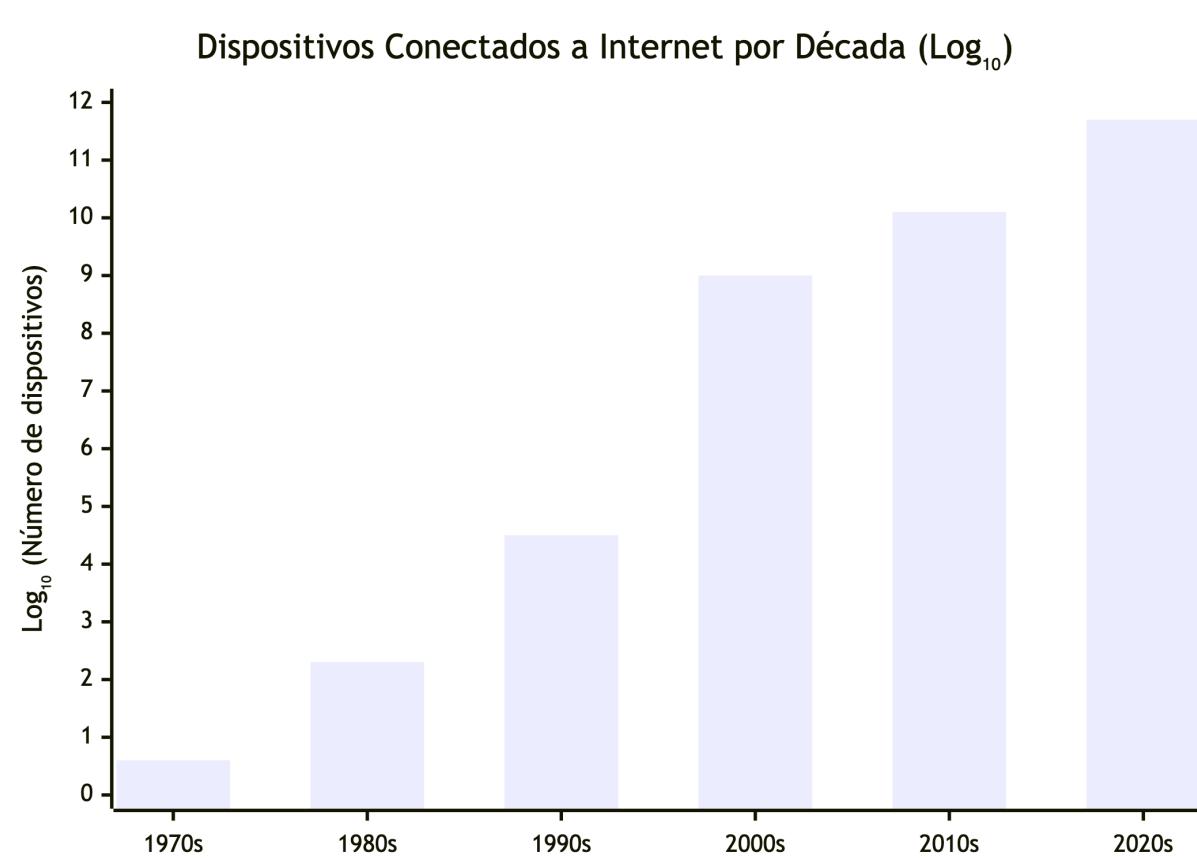


Figura 1.3: Número de dispositivos conectados por década desde ARPANET hasta Internet en 2025 (Ritchie et al. 2023; Analytics 2020)

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.

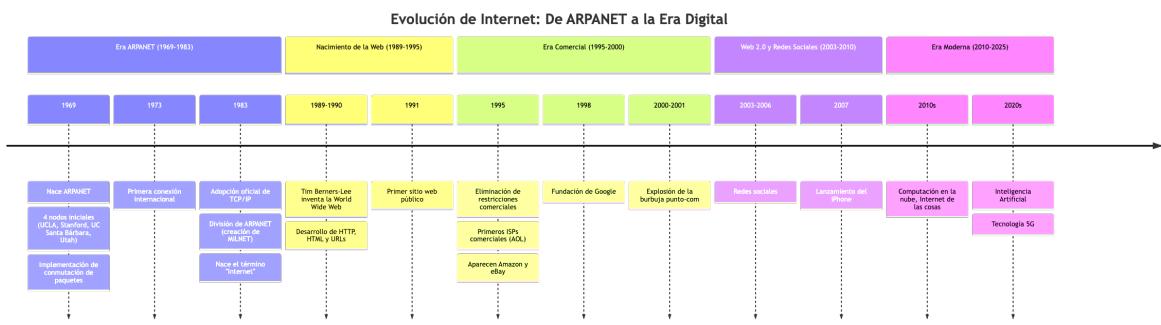


Figura 1.4: Esquema de tiempo de los eventos más significativos desde la creación de ARPANET hasta 2025.

### 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.

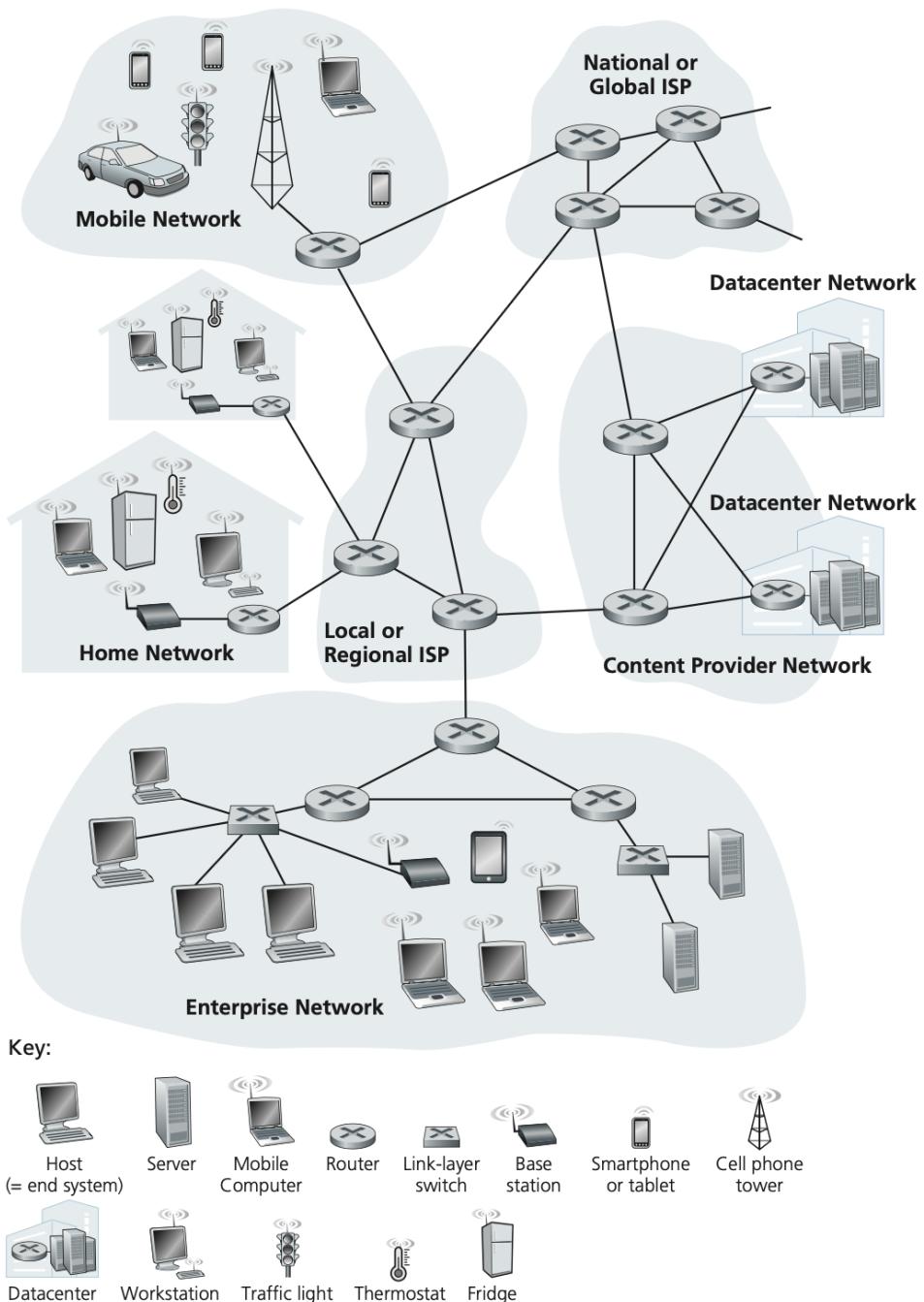


Figura 1.5: Red simplificada (Kurose y Ross 2017).

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 [Tabla 1.1](#) podéis ver una comparativa de sus principales características.

Tabla 1.1: Tabla comparativa de tecnologías de acceso host-router. \*Simétrica en teoría, asimétrica en la práctica.

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

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.

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

Comparativa de diferentes tecnologías {#tab-ni-infra-edge-core}

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 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.

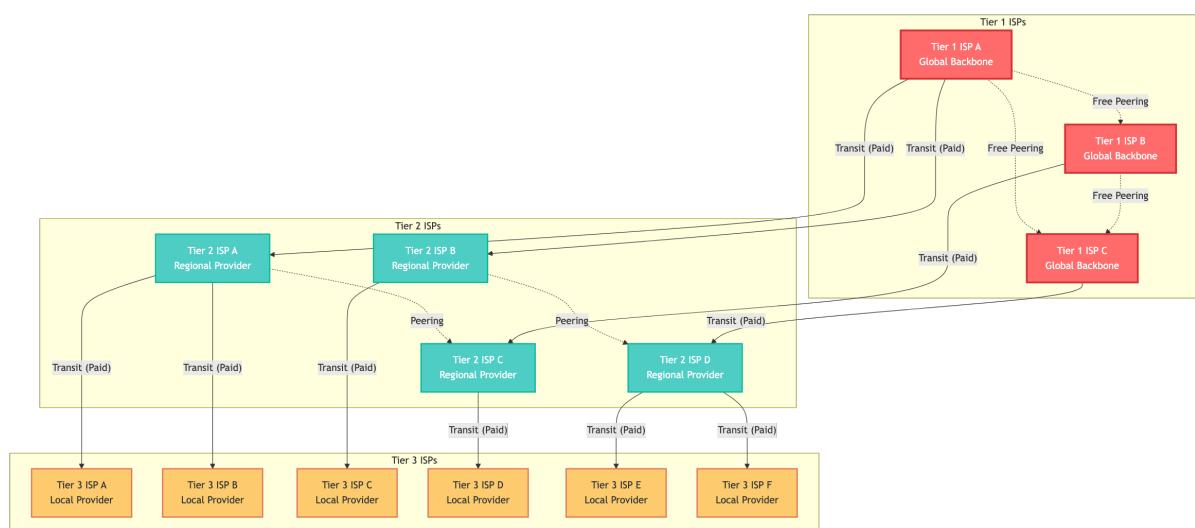


Figura 1.6: Jerarquía de ISPs

## 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 1.7](#) podemos ver los modelos OSI y TCP/IP divididos en sus diferentes capas y cuál es la equivalencia entre ambos.

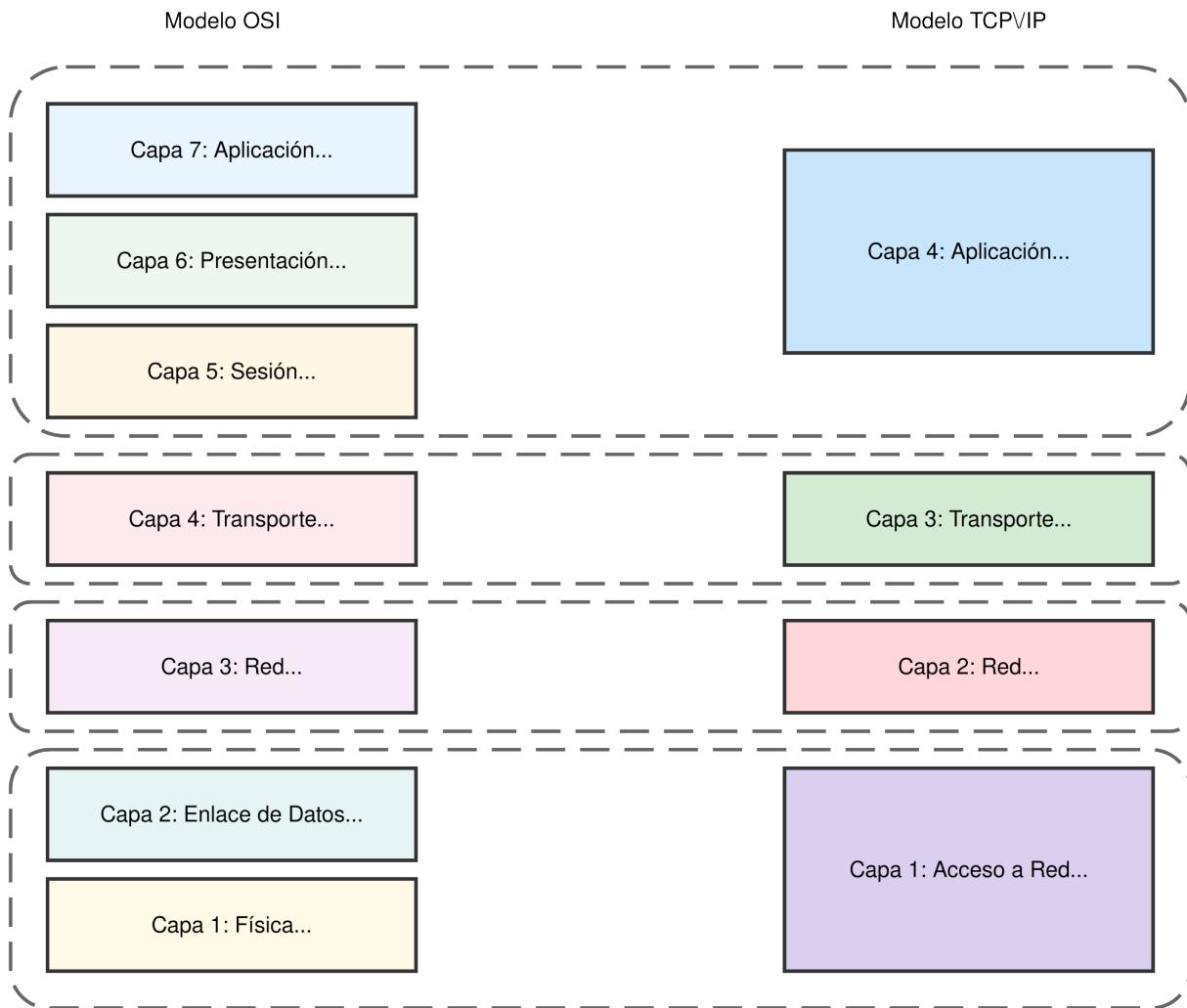


Figura 1.7: Modelos OSI y TCP/IP.

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.

### Note

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 también 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.

## 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.

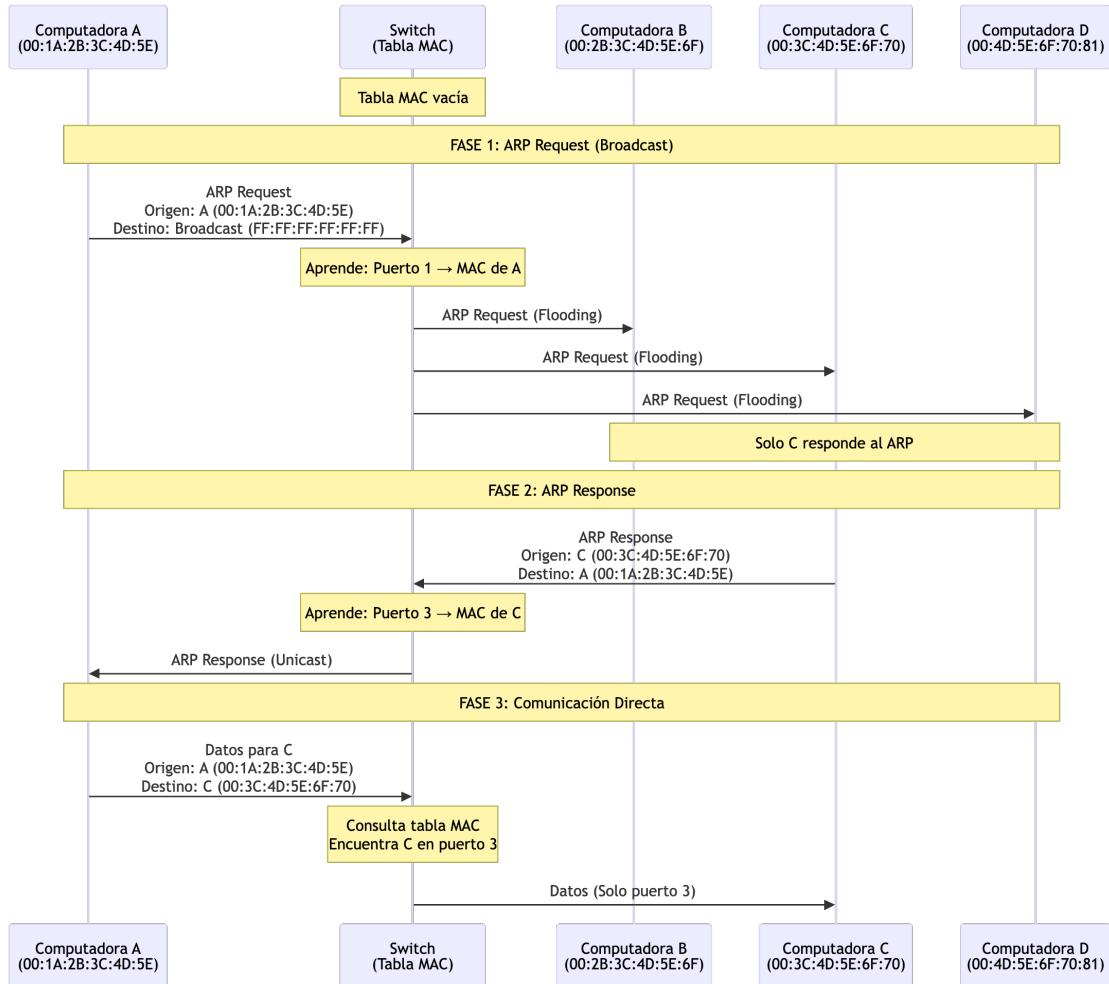


Figura 2.1: Ejemplo de envío de paquetes en una red local donde no se conoce la MAC del destinatario.

## 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 comparten 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, monitorización 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 automático, full-duplex MAC	Redes pequeñas y domésticas
<b>Switches Gestiónados</b>	Conectividad avanzada LAN	VLANs, QoS, SNMP, seguridad 802.1X, port mirroring	Redes empresariales
<b>Puntos de acceso</b>	Conectividad inalámbrica	CSMA/CA, traducción cableado-wireless, beamforming	Redes empresariales Wi-Fi
<b>Repetidores</b>	Extensión alcance	Regeneración de señal, sin filtrado	Superación de límites de distancia
<b>Amplificadores</b>	Amplificación señal	Aumento de potencia RF/óptica	Enlaces de larga distancia
<b>Conversores de medio</b>	Conversión medios	Fibra ↔ cobre, adaptación de velocidades	Migración gradual, extensión

Dispositivo	Función Principal	Características Clave	Aplicación Típica
<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.

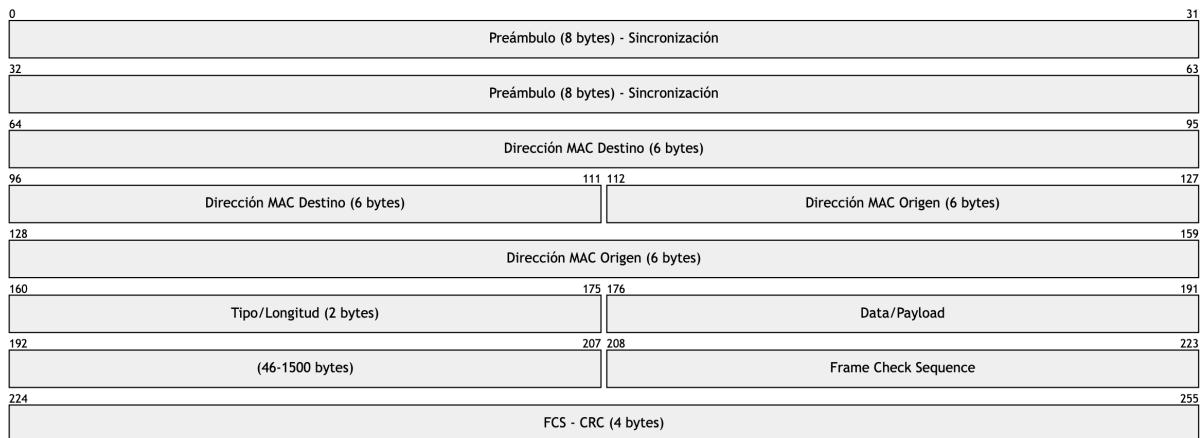


Figura 2.2: Cabeceras de un paquete de Ethernet.

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.

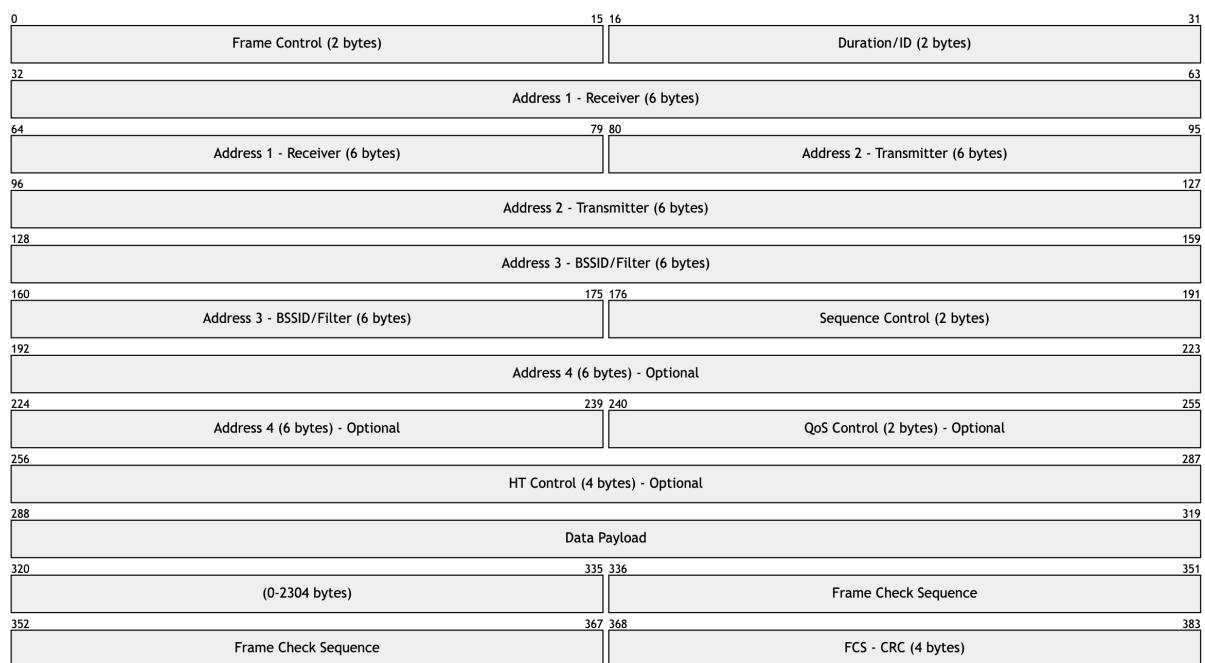


Figura 2.3: Cabeceras de un paquete Wi-Fi

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.

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:

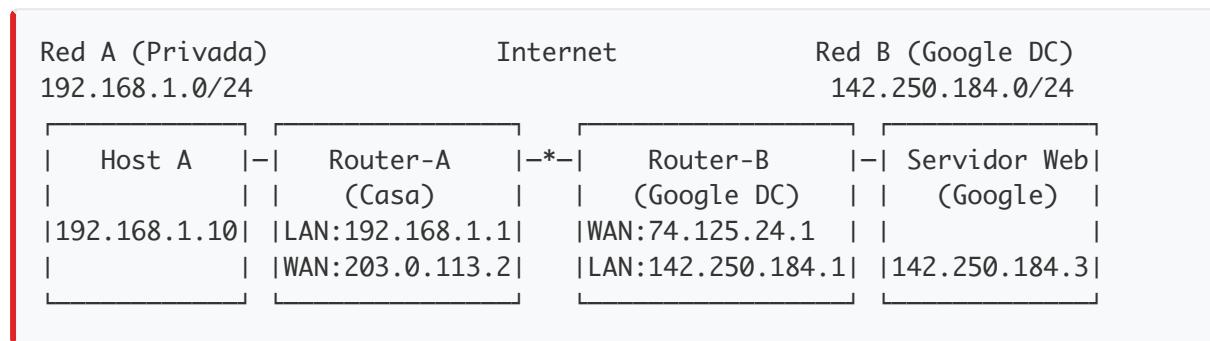


Figura 3.1: Ejemplo simplificado de la estructura de red.

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.

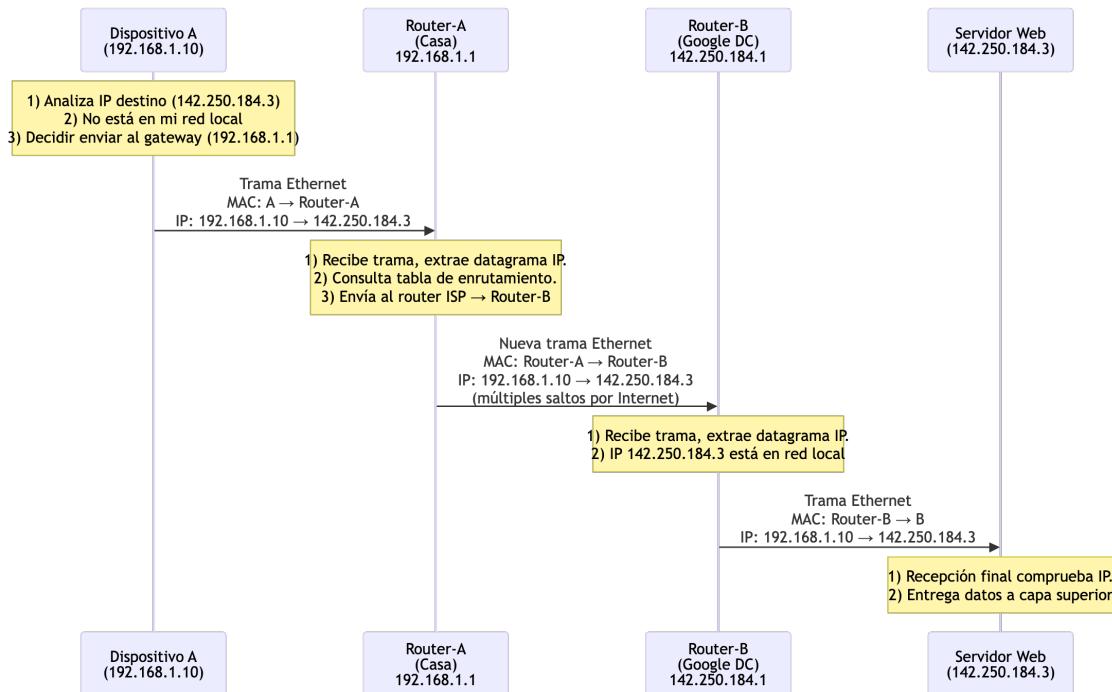


Figura 3.2: Ejemplo de envío de datagrama IP entre dos ordenadores en diferentes redes.

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 el 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, 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>	Diffícil	Automática
<b>Escalabilidad</b>	Limitada	Alta
<b>Overhead</b>	Setup/teardown	Por paquete

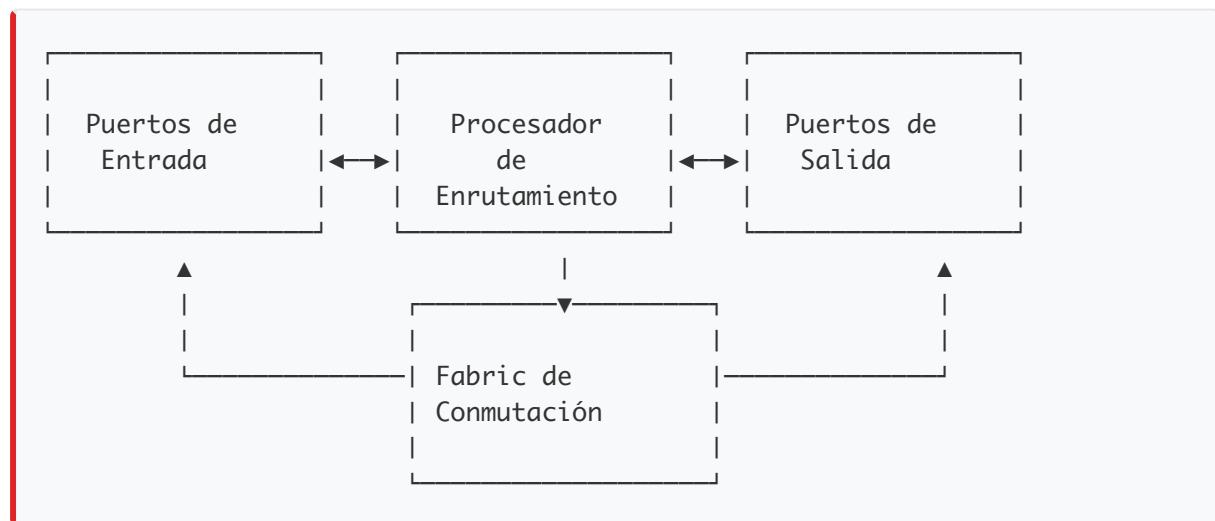
### 3.3. Dispositivos físicos de la Capa de Red

Los dispositivos de capa de red son los componentes hardware que hacen posible la interconexión de redes y la implementación de las funciones de enruteamiento 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 enruteamiento mediante software especializado y generan tablas de enruteamiento 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 enruteamiento generado 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.

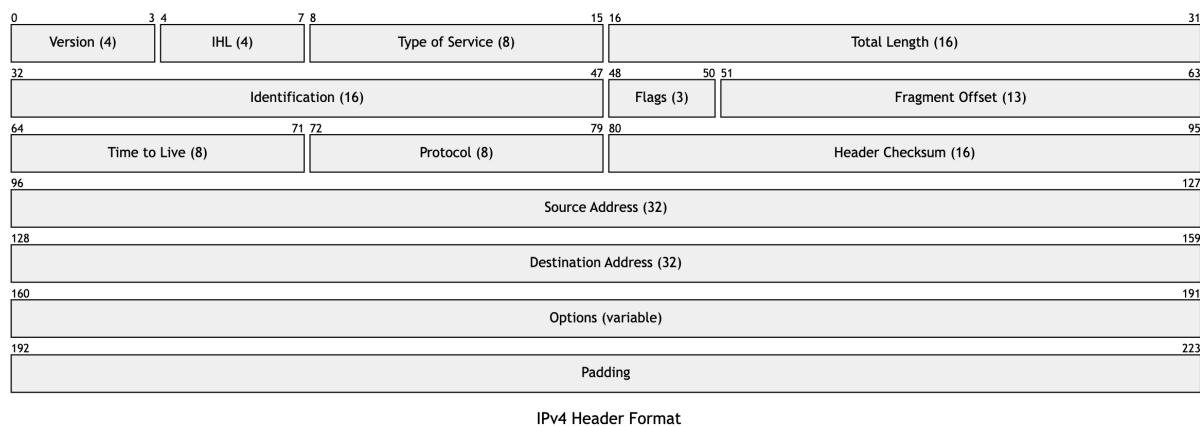


Figura 3.3: Formato de cabeceras de IPv4.

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 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.

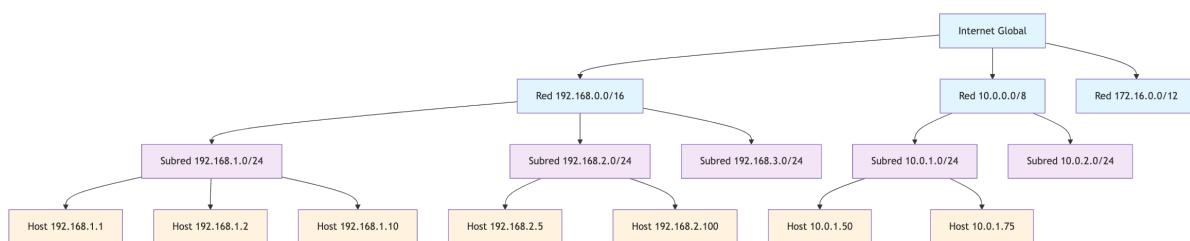


Figura 3.4: Ejemplo de estructuras de subredes.

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
A	0.0.0.0 - 127.255.255.255	0	7	24	126 <sup>8</sup>	16,777,214	ISPs, gobiernos, organizaciones masivas
B	128.0.0.0 - 191.255.255.255	10	14	16	16,384	65,534	Universidades, empresas medianas
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 una 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.

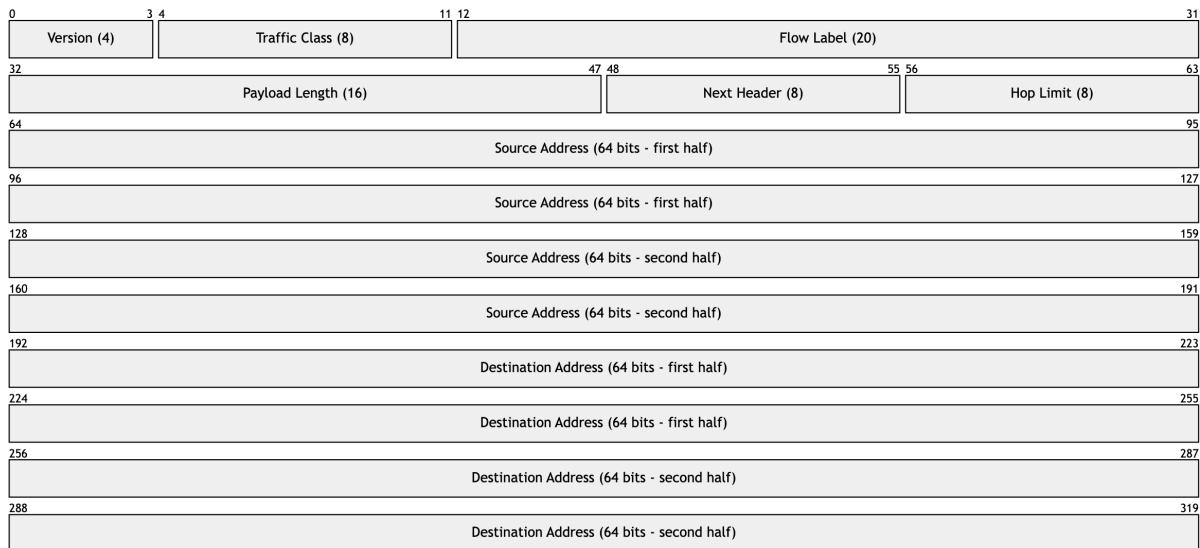


Figura 3.5: Formato de cabeceras de IPv6.

### 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 balanceadores 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  $\leftrightarrow$  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.

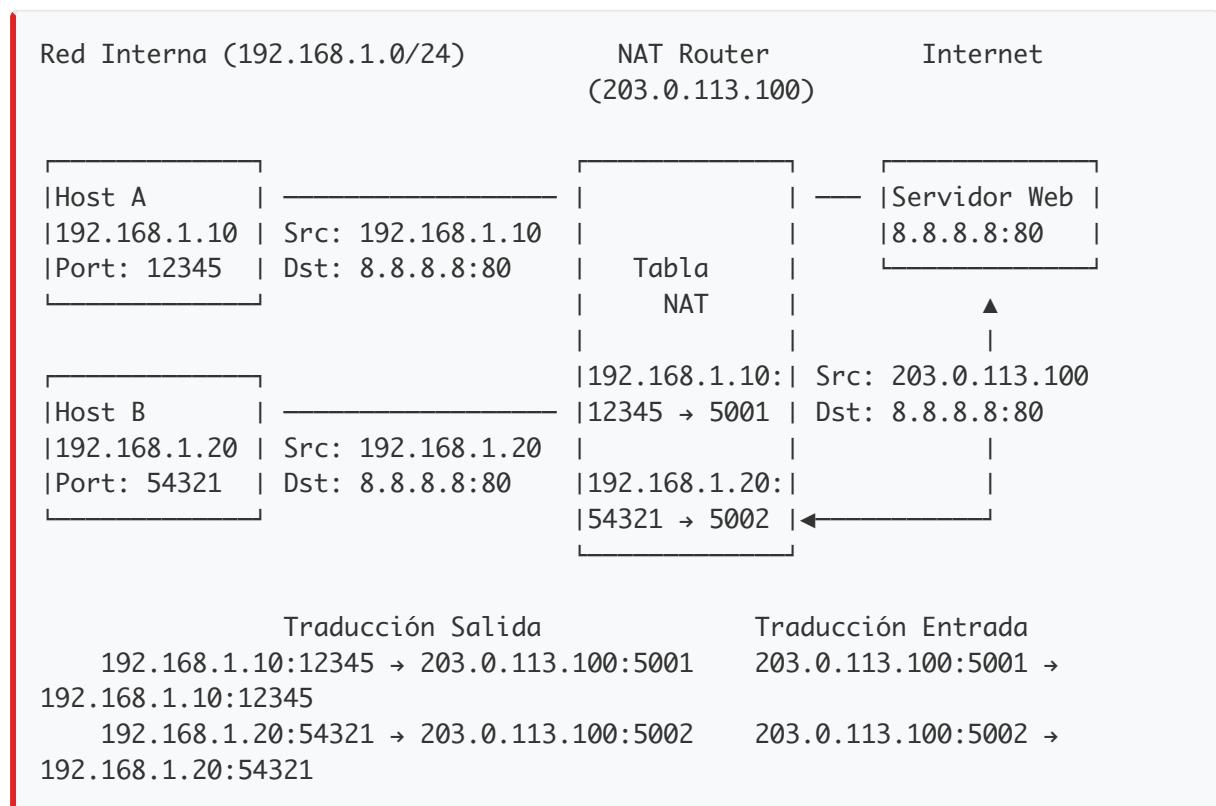


Figura 3.6: Ejemplo de NAT con dos Host que se comunican con un servidor web utilizando una única IP pública.

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.

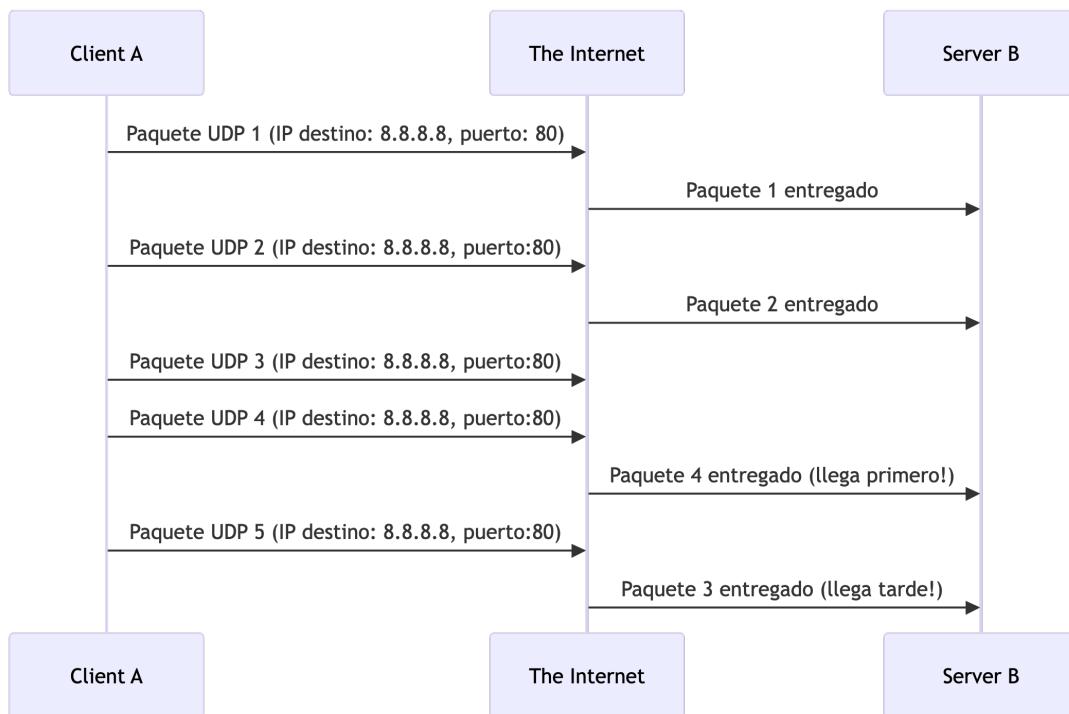


Figura 4.1: Ejemplo de envío de 5 paquetes a través de Internet con UDP. “Internet” en este diagrama representa todo el proceso de envío.

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.

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 vídeos. 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:

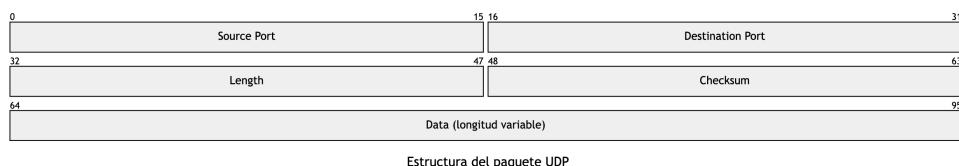


Figura 4.2: Estructura del paquete UDP

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.

```

// Cálculo de checksum en UDP
function calculateUDPChecksum(srcIP, dstIP, srcPort, dstPort, data) {
    // Convert string data to Uint8Array if needed
    if (typeof data === 'string') {
        data = new TextEncoder().encode(data);
    }

    const udpLength = 8 + data.length;

    // Build complete packet: pseudo-header + UDP header + data
    const packet = [
        // Pseudo-header (12 bytes)
        ...srcIP.split('.').map(x => parseInt(x)),           // Source IP
        (4 bytes)
        ...dstIP.split('.').map(x => parseInt(x)),           // Dest IP (4
        bytes)
        0, 17,                                                 // Zero +
        Protocol UDP (2 bytes)
        (udpLength >> 8) & 0xFF, udpLength & 0xFF,          // UDP length
        (2 bytes)

        // UDP header (8 bytes)
        (srcPort >> 8) & 0xFF, srcPort & 0xFF,             // Source port
        (2 bytes)
        (dstPort >> 8) & 0xFF, dstPort &
        0xFF,                                              // Dest port (2 bytes)
        (udpLength >> 8) & 0xFF, udpLength & 0xFF,          // UDP length
        (2 bytes)
        0, 0,                                                 // Checksum
        placeholder (2 bytes)

        // Data payload
        ...Array.from(data)
    ];

    // Calculate 16-bit one's complement checksum
    let sum = 0;
    for (let i = 0; i < packet.length - 1; i += 2) {
        sum += (packet[i] << 8) + packet[i + 1];
    }

    // Handle odd length
    if (packet.length % 2 === 1) {
        sum += packet[packet.length - 1] << 8;
    }

    // Add carry bits and return one's complement
    while (sum >> 16) sum = (sum & 0xFFFF) + (sum >> 16);
    return (~sum) & 0xFFFF;
}

```

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:

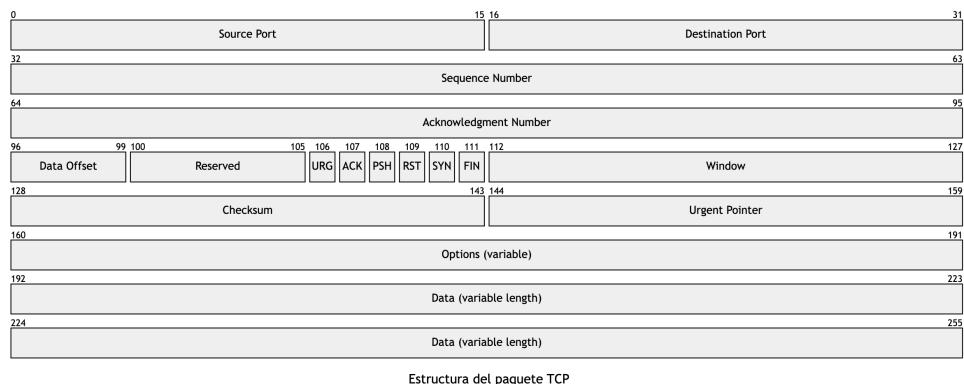


Figura 4.3: Estructura del paquete TCP

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:

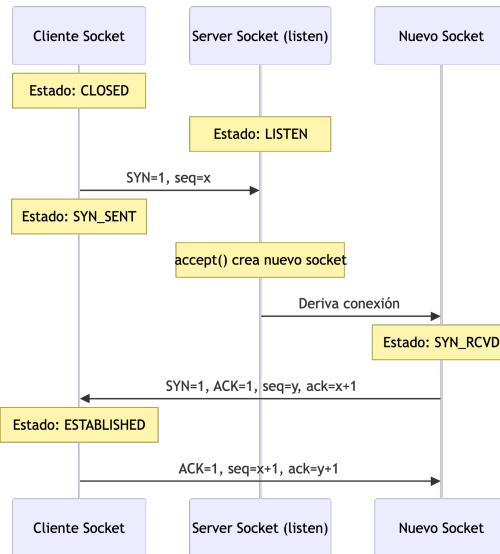


Figura 4.4: Handshake de tres fases TCP

**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).

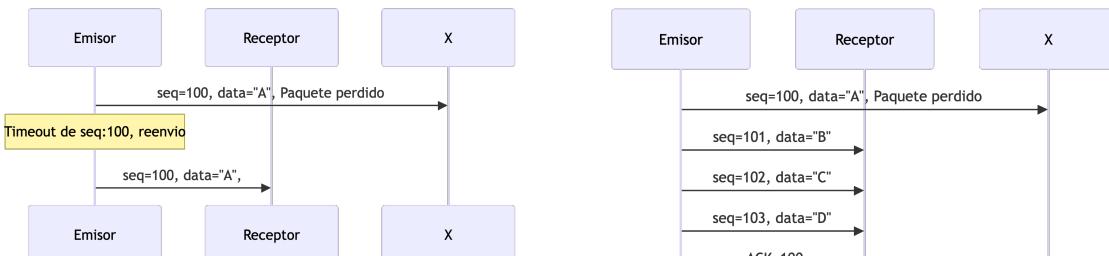


Figura 4.5: Detección de pérdidas por timeout

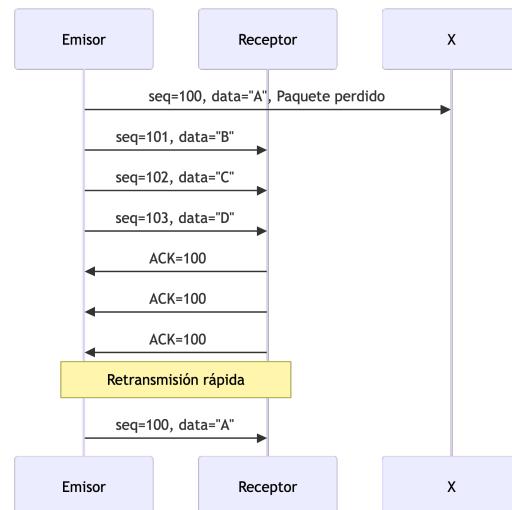


Figura 4.6: Detección de pérdidas por ACKs duplicados

#### 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:

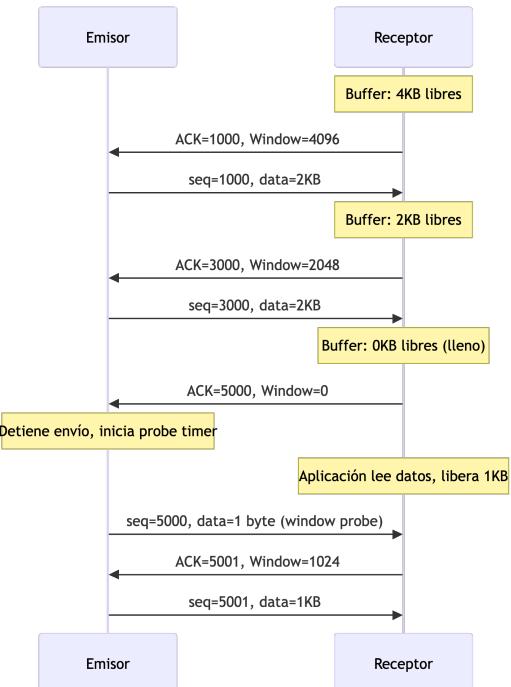


Figura 4.7: Control de flujo TCP

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 [Figura 4.8](#)) 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 [Figura 4.9](#)).

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.

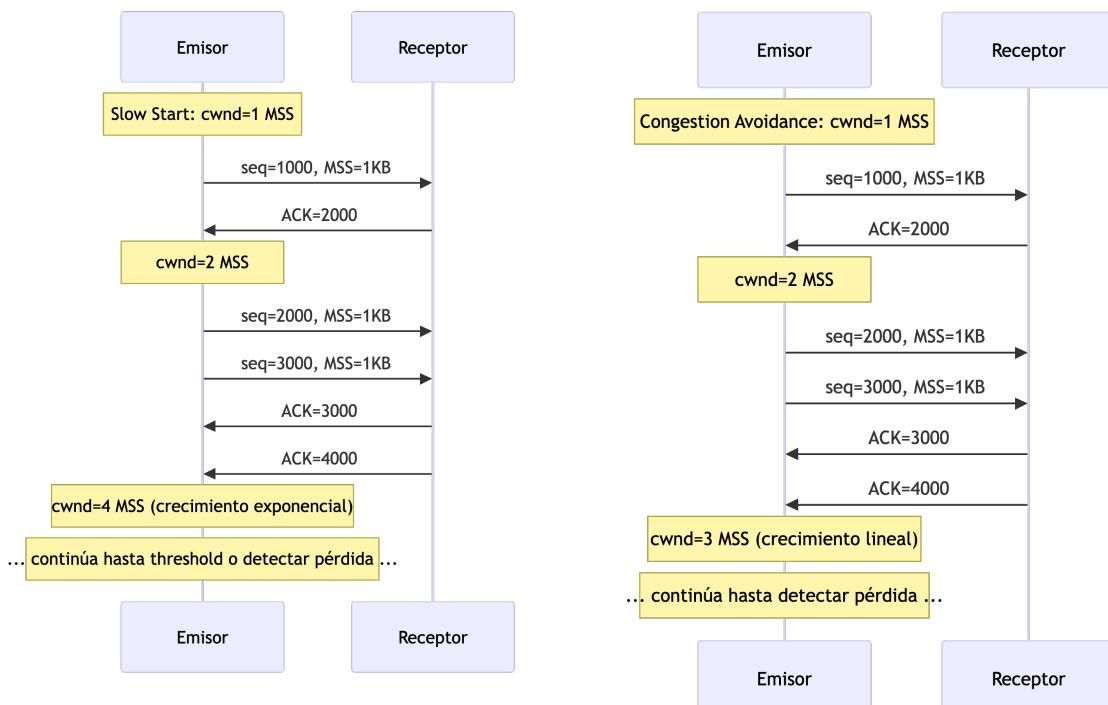


Figura 4.8: A) Mecanismo de congestión slow start.

Figura 4.9: B) Mecanismo de congestión congestion avoidance.

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>

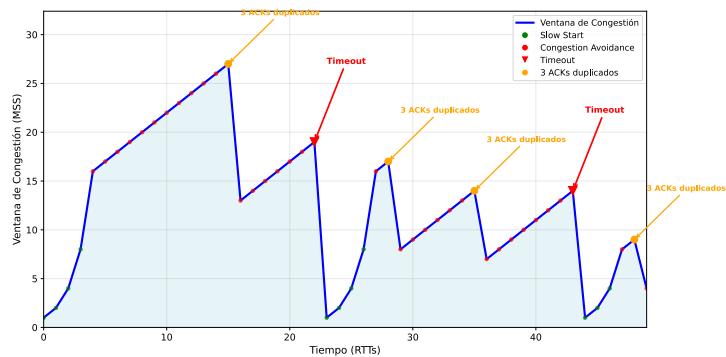


Figura 4.10: Patrón de diente de sierra en el control de congestión TCP

#### 4.2.2.5. Terminación de Conexión

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

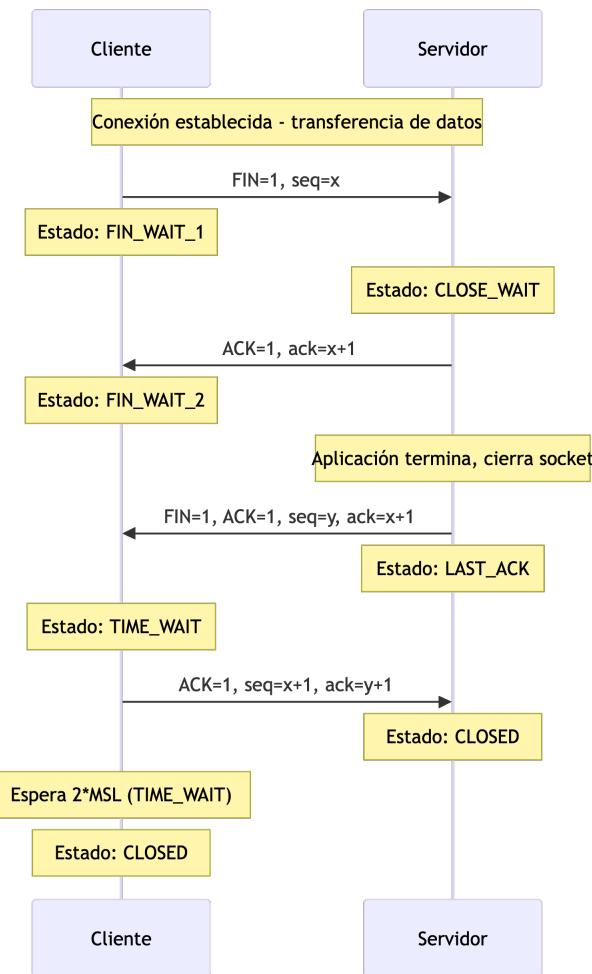


Figura 4.11: Terminación de conexión TCP

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>9</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>10</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 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 sature 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 donde 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
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 (DoH/ DoT)	UDP/TCP

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.

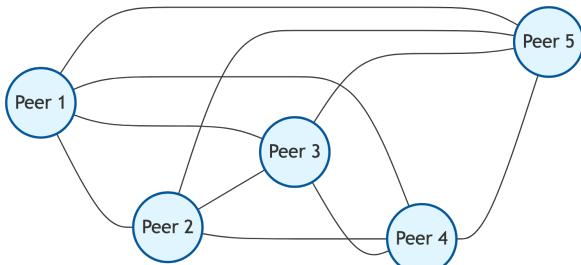


Figura 5.1: Full Mesh Topology

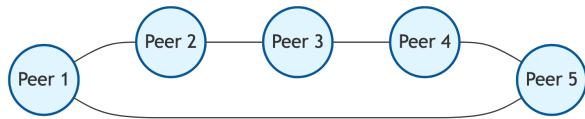


Figura 5.2: Ring Topology

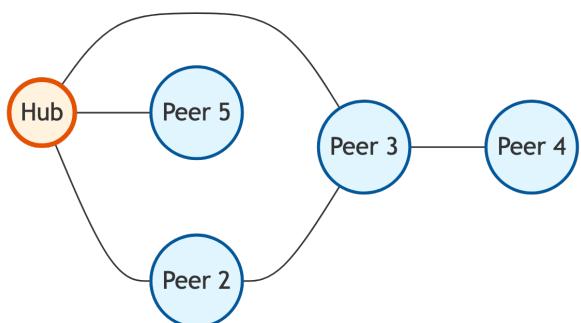


Figura 5.3: Hybrid Topology

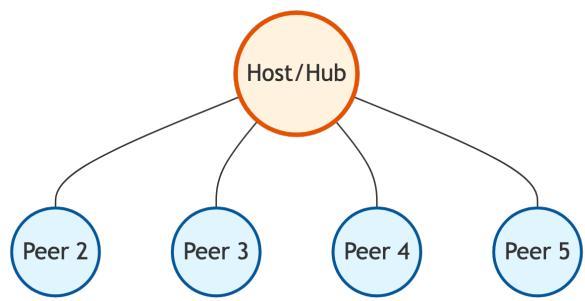


Figura 5.4: Star Topology (Pseudo-P2P)

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 [Figura 5.3](#)) 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 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.

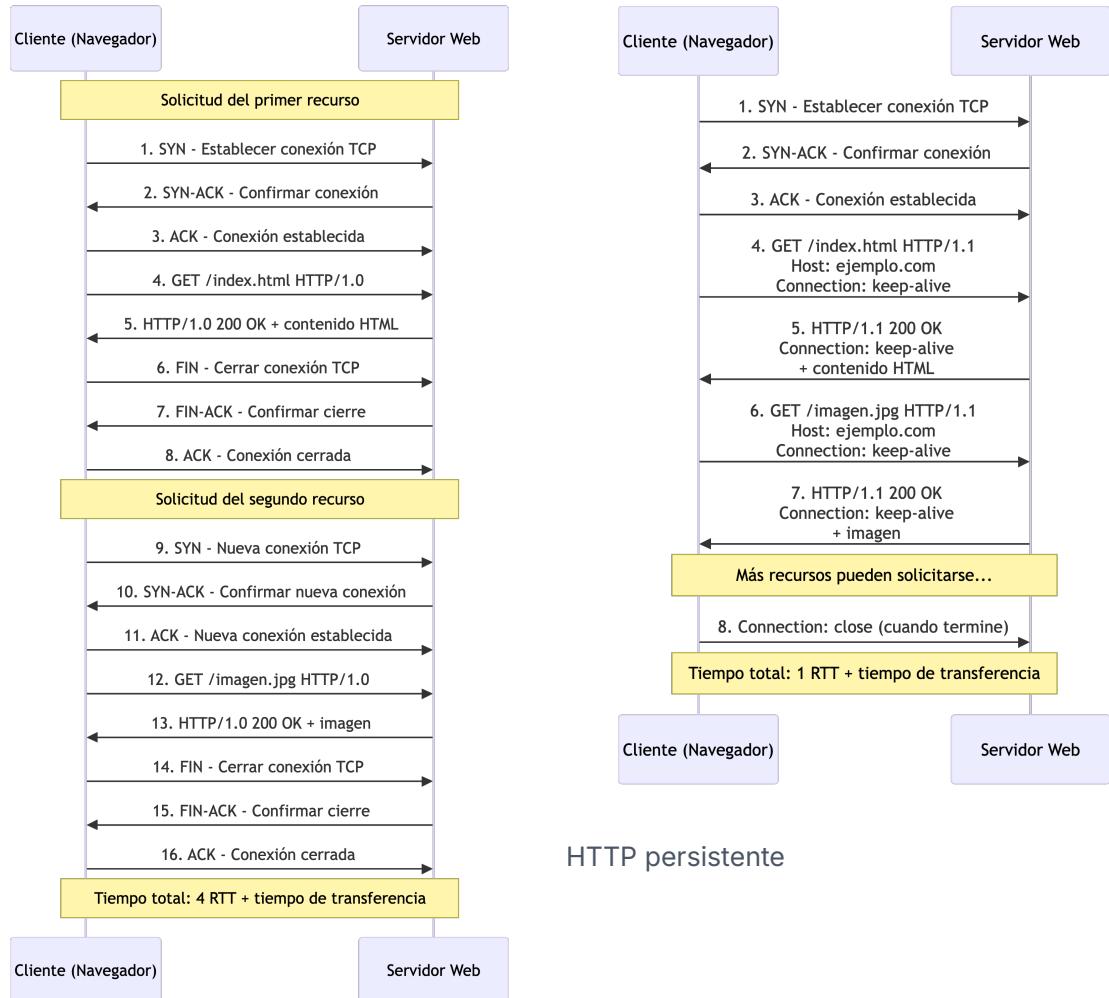
Cabe destacar que este uso esperado de los verbos lo tenemos que implementar nosotros. Nada nos quita de hacer que un GET borre cosas, o se utilice para acciones para las que no estaba diseñado. Sin embargo, seguir la especificación nos va a permitir que otros usuarios de nuestra API la puedan utilizar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



### HTTP no persistente

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 exigen

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 cachean 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 devuelve 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 [Figura 5.5](#). 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.

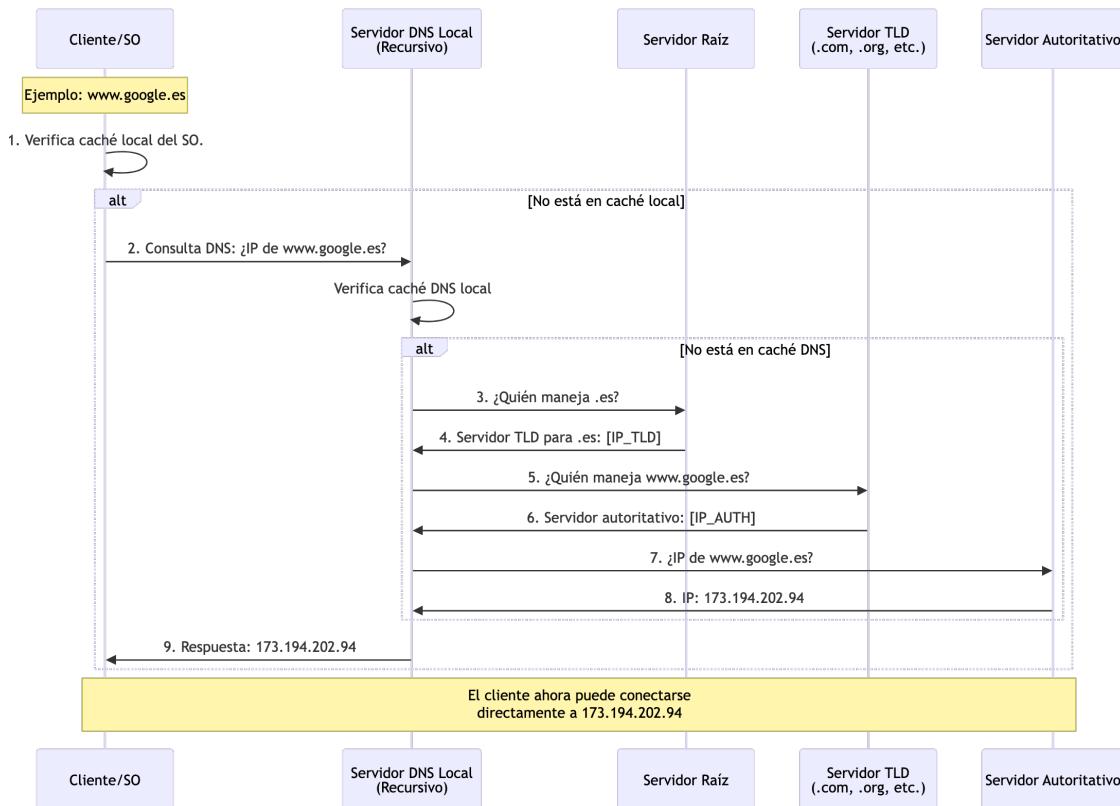


Figura 5.5: Proceso de resolución DNS

### 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 reconnexiones, 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 balanceo 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.

## Desarrollo en el lado del cliente

---

Una vez comprendidos los fundamentos de las redes de comunicación, es momento de adentrarnos en el desarrollo del lado del cliente, donde JavaScript emerge como el lenguaje fundamental para crear aplicaciones web interactivas. En esta parte del curso, comenzaremos dominando los conceptos esenciales de JavaScript, desde sus fundamentos básicos como variables, tipos de datos y estructuras de control, hasta aspectos más avanzados como la programación orientada a objetos, la manipulación del DOM y la gestión de eventos.

Una vez establecidas las bases de JavaScript, exploraremos HTML y CSS, las tecnologías que definen la estructura y presentación de las páginas web. HTML nos permitirá crear los contenedores y elementos de interfaz necesarios para nuestros juegos, mientras que CSS controlará su apariencia visual. Este conocimiento combinado será crucial para desarrollar la interfaz de nuestros juegos en red y para comprender cómo el cliente interactúa con los servicios del servidor en un entorno multijugador.

# 6. JavaScript

---

## 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

El navegador web proporciona al JavaScript dos APIs fundamentales para interactuar con la página y el entorno del navegador. Entender la diferencia entre estas dos es crucial para el desarrollo de videojuegos web.

**DOM (Document Object Model):** Es una representación en forma de árbol del documento HTML cargado en el navegador. Cada elemento HTML (divs, canvas, botones, etc.) se convierte en un objeto que JavaScript puede manipular. Para videojuegos, el DOM es esencial para:

- Acceder al elemento `<canvas>` donde se renderiza el juego
- Crear y manipular elementos de la interfaz de usuario (menús, HUD, puntuaciones)
- Gestionar eventos de entrada del usuario (clicks, teclas presionadas, movimientos del ratón)
- Modificar estilos CSS dinámicamente (pantallas de carga, efectos visuales)
- Insertar y eliminar elementos HTML en tiempo real (mensajes, notificaciones)

**BOM (Browser Object Model):** Proporciona acceso a funcionalidades del navegador más allá del documento HTML. El BOM incluye al DOM como uno de sus componentes, pero añade capacidades adicionales críticas para videojuegos:

- **window:** El objeto global que representa la ventana del navegador
  - Control del tamaño de la ventana
  - Detección de cambio de tamaño (responsive design)
  - Gestión del foco de la ventana (pausar el juego cuando se cambia de pestaña)
- **navigator:** Información sobre el navegador y el dispositivo
  - Detección de características soportadas (WebGL, audio, gamepad)
  - Información del sistema operativo y navegador
  - Acceso a la geolocalización y otros sensores
- **location:** Control de la URL actual
  - Navegación entre diferentes pantallas del juego
  - Gestión del historial del navegador
- **localStorage/sessionStorage:** Almacenamiento persistente de datos
  - Guardar progreso del jugador
  - Almacenar configuraciones y preferencias
- **fetch/XMLHttpRequest:** Comunicación con servidores
  - Cargar recursos del juego dinámicamente
  - Enviar y recibir datos de partidas multijugador
  - Comunicación con APIs REST
- **console:** Herramienta de debugging
  - Logging de errores y mensajes de debug
  - Medición de rendimiento del juego

En resumen: el **DOM** se centra en el contenido de la página HTML, mientras que el **BOM** proporciona acceso a todas las capacidades del navegador. Para videojuegos web, ambos son fundamentales y trabajan juntos para crear experiencias interactivas completas.

---

### 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

#### 6.2.4.1. Inicializar un proyecto

Todo proyecto de JavaScript moderno comienza con la inicialización de npm, que crea el archivo de configuración principal del proyecto. Este archivo, llamado `package.json`, actúa como el “documento de identidad” del proyecto, describiendo sus características, dependencias y comandos de ejecución.

```
# Crear directorio del proyecto  
mkdir mi-juego-web  
cd mi-juego-web  
  
# Inicializar proyecto npm  
npm init -y
```

El comando `npm init -y` crea automáticamente el archivo `package.json` con valores por defecto. La opción `-y` (yes) acepta todas las opciones por defecto sin preguntar, acelerando el proceso. Si omitimos `-y`, npm nos hará preguntas interactivas sobre cada campo.

Esto crea un archivo `package.json` que podemos personalizar para nuestro videojuego:

```
{
  "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"
}
```

### Explicación de cada campo del package.json:

- `name`: Identificador único del proyecto. Debe ser en minúsculas, sin espacios (usar guiones). Este nombre se usa si publicamos el proyecto en npm. Para nuestro videojuego, elegimos un nombre descriptivo como "mi-juego-web".
- `version`: Número de versión siguiendo el formato semántico (semantic versioning): MAJOR.MINOR.PATCH. Comenzamos en "1.0.0" y actualizamos según los cambios: cambios mayores incrementan el primer número, nuevas características el segundo, y correcciones de bugs el tercero.
- `description`: Texto breve que explica qué es el proyecto. Útil cuando otros desarrolladores encuentren nuestro código o cuando busquemos proyectos en npm. Para videojuegos, describimos el género o concepto del juego.
- `main`: Punto de entrada principal del proyecto. Indica qué archivo JavaScript se ejecuta cuando alguien importa nuestro proyecto. Para videojuegos web, típicamente apunta al archivo principal del juego en `src/index.js` o `src/main.js`.
- `scripts`: Objeto que define comandos personalizados que podemos ejecutar con `npm run <nombre>`. Estos scripts automatizan tareas comunes:
  - `"start"`: Comando para ejecutar el proyecto en producción
  - `"dev"`: Inicia el servidor de desarrollo con recarga automática
  - `"build"`: Genera los archivos optimizados para producción

Ejecutamos estos scripts con `npm start`, `npm run dev`, o `npm run build`.

- `keywords` : Array de palabras clave que describen el proyecto. Útil para que otros encuentren el proyecto si lo publicamos en npm. Para videojuegos, incluimos términos como el género, tecnologías usadas, etc.
- `author` : Nombre del creador o equipo de desarrollo. Puede incluir email: "Tu Nombre <email@email.com>" .
- `license` : Tipo de licencia del código. "MIT" es una licencia permisiva común para proyectos open source. Otras opciones populares son "GPL-3.0", "Apache-2.0", o "UNLICENSED" para proyectos privados.

#### **6.2.4.2. Estructura de carpetas recomendada**

```
mi-juego-web/
├── package.json
├── webpack.config.js
├── src/
├── 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 descargarán:

```
# 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 (bundler). 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. Además, Webpack incluye un servidor de desarrollo con recarga automática que facilita enormemente el proceso de desarrollo.

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

```

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

module.exports = {
  entry: './src/main.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true
  },
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
    static: './dist',
    hot: true,
    port: 8080
  },
  externals: {
    phaser: 'Phaser'
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './public/index.html',
      inject: false
    })
  ],
  resolve: {
    extensions: ['.js']
  }
};

```

### Explicación detallada de cada sección del webpack.config.js:

#### Importaciones iniciales:

```

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

```

- `path`: Módulo de Node.js para trabajar con rutas de archivos de forma independiente del sistema operativo. Garantiza que las rutas funcionen tanto en Windows como en Mac/Linux.
- `HtmlWebpackPlugin`: Plugin que genera automáticamente el archivo HTML final incluyendo las referencias a los archivos JavaScript generados.

#### Punto de entrada (entry):

```
entry: './src/main.js',
```

Especifica el archivo JavaScript principal desde donde Webpack comienza a construir el grafo de dependencias del proyecto. Webpack analiza este archivo, encuentra todos los imports/requires, y recursivamente incluye todos los módulos necesarios. Para videojuegos, este suele ser el archivo que inicializa el motor de juego y carga la primera escena.

### Configuración de salida (output):

```
output: {  
  filename: 'bundle.js',           // Nombre del archivo JavaScript  
  generated  
  path: path.resolve(__dirname, 'dist'), // Directorio donde se guardan  
  los archivos generados  
  clean: true                    // Limpia el directorio dist antes  
  de cada build  
}
```

- `filename` : Nombre del archivo JavaScript final que contendrá todo nuestro código empaquetado y optimizado.
- `path` : Directorio de salida. `__dirname` es el directorio actual, `'dist'` es donde se guardarán los archivos de producción.
- `clean: true` : Elimina archivos antiguos antes de generar nuevos, evitando acumulación de builds anteriores.

### Modo de desarrollo (mode):

```
mode: 'development',
```

Define el entorno de ejecución. Puede ser:

- `'development'` : Código sin minificar, con nombres de variables legibles, útil para debugging.
- `'production'` : Código minificado y optimizado, menor tamaño, más rápido, pero difícil de debuggear.

### Herramientas de debugging (devtool):

```
devtool: 'inline-source-map',
```

Genera source maps que permiten debuggear el código original (antes de empaquetarlo) en las DevTools del navegador. Cuando hay un error, el navegador muestra la línea del archivo fuente original en lugar del código empaquetado, facilitando enormemente el debugging.

### Servidor de desarrollo (devServer):

```
devServer: {  
  static: './dist',    // Directorio desde donde servir archivos estáticos  
  hot: true,          // Hot Module Replacement - actualiza módulos sin  
                      // recargar la página  
  port: 8080          // Puerto donde se ejecuta el servidor de desarrollo  
}
```

Configura el servidor web integrado de Webpack:

- `static`: Directorio con archivos estáticos (HTML, imágenes, etc.) que el servidor debe servir.
- `hot: true`: Habilita Hot Module Replacement (HMR), que permite actualizar módulos JavaScript en tiempo real sin recargar toda la página. Muy útil durante desarrollo para mantener el estado del juego.
- `port`: Puerto donde se ejecutará el servidor (`http://localhost:8080`).

### Librerías externas (externals):

```
externals: {  
  phaser: 'Phaser'  
}
```

Indica a Webpack que ciertas dependencias se cargarán desde fuera del bundle (típicamente desde un CDN). En este caso, Phaser se carga desde un `<script>` en el HTML en lugar de incluirse en el bundle, reduciendo el tamaño del archivo final. La clave es el nombre del módulo en el código (`import Phaser from 'phaser'`) y el valor es la variable global que expone (`window.Phaser`).

### Plugins:

```
plugins: [
  new HtmlWebpackPlugin({
    template: './public/index.html', // Archivo HTML plantilla
    inject: false // No inyecta automáticamente los
      scripts
  })
]
```

- `HtmlWebpackPlugin`: Genera el archivo HTML final a partir de una plantilla.
- `template` : Ruta al archivo HTML base que usará como plantilla.
- `inject: false`: Desactiva la inyección automática de scripts. Útil cuando queremos control manual sobre dónde y cómo se cargan los scripts en el HTML (por ejemplo, cuando usamos librerías externas como Phaser desde CDN).

#### Resolución de módulos (resolve):

```
resolve: {
  extensions: ['.js']
}
```

Define qué extensiones de archivo Webpack intentará resolver automáticamente cuando importamos módulos sin especificar extensión. Esto permite escribir `import Player from './player'` en lugar de `import Player from './player.js'`. Útil para proyectos grandes donde escribir las extensiones constantemente es tedioso.

### 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 a excepción de los tipos primitivos son objetos,
  - Sin embargo, a diferencia de Java, el boxing es automático.
  - 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. 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 immutable, 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

## 6.5.5. 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}
`;
```

## 6.5.6. 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. 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.

---

### 6.10.1. 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

#### 6.10.1.1. Guardar datos

LocalStorage 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));
```

#### 6.10.1.2. 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;
```

#### 6.10.1.3. 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');
}
```

#### 6.10.1.4. Eliminar datos

Para eliminar datos específicos:

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

Para eliminar todos los datos del dominio:

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

#### 6.10.1.5. 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);
}
```

#### 6.10.1.6. 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;
    }
}
```

#### 6.10.2. 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));
```

### 6.10.3. 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

#### 6.10.3.1. 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=/
`;
```

#### 6.10.3.2. 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');
```

### 6.10.3.3. 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=/";
```

### 6.10.3.4. Opciones de seguridad

Las cookies pueden incluir opciones adicionales de seguridad:

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

### 6.10.3.5. 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');

```

#### 6.10.4. 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

### 6.10.5. 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

---

## 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 prototipoJugador = Object.create(personajeBase);
prototipoJugador.atacar = function(objetivo) {
    return `${this.nombre} ataca a ${objetivo.nombre} por ${this.fuerza} puntos`;
};

prototipoJugador.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 comparten 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.estáVivo) { // No necesita paréntesis
    console.log("El goblin sigue vivo");
}

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

// 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.

# 8. HTML y CSS

---

## 8.1. Introducción

HTML (HyperText Markup Language) y CSS (Cascading Style Sheets) son las tecnologías fundamentales para crear páginas web. HTML define la estructura y el contenido de la página mediante etiquetas, mientras que CSS controla su presentación visual. En el desarrollo de videojuegos web, HTML proporciona el contenedor donde se ejecuta el juego y los elementos de la interfaz de usuario, mientras que CSS da estilo a estos elementos.

## 8.2. HTML (HyperText Markup Language)

---

### 8.2.1. ¿Qué es HTML?

HTML es un lenguaje de marcado que utiliza **etiquetas** (tags) para estructurar el contenido. Una etiqueta se escribe entre corchetes angulares `<>` y generalmente viene en pares: una etiqueta de apertura `<etiqueta>` y una de cierre `</etiqueta>`. El contenido se coloca entre ambas.

```
<p>Este es un párrafo de texto.</p>
```

Algunas etiquetas no tienen contenido y se auto-cierran:

```

<br>
```

### 8.2.2. Estructura Básica de un Documento HTML

Todo documento HTML sigue una estructura estándar:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi Primera Página Web</title>
</head>
<body>
    <h1>Bienvenido</h1>
    <p>Este es el contenido visible de la página.</p>
</body>
</html>
```

#### Explicación de cada parte:

- `<!DOCTYPE html>`: Declara que este es un documento HTML5 (la versión más reciente).
- `<html>`: Elemento raíz que contiene todo el documento. El atributo `lang="es"` indica el idioma.
- `<head>`: Contiene metadatos e información que no se muestra directamente (título, enlaces a CSS, etc.).
- `<meta charset="UTF-8">`: Define la codificación de caracteres para soportar acentos y símbolos especiales.
- `<title>` : El texto que aparece en la pestaña del navegador.
- `<body>` : Contiene todo el contenido visible de la página.

---

### 8.2.3. Elementos de Texto

HTML ofrece diversas etiquetas para estructurar texto:

```
<!-- Encabezados (h1 es el más importante, h6 el menos) -->
<h1>Título Principal</h1>
<h2>Subtítulo</h2>
<h3>Sección</h3>

<!-- Párrafos -->
<p>Este es un párrafo de texto normal.</p>

<!-- Formato de texto -->
<strong>Texto importante (negrita)</strong>
<em>Texto enfatizado (cursiva)</em>
<mark>Texto resaltado</mark>
<small>Texto pequeño</small>

<!-- Saltos de línea -->
<p>Primera línea<br>Segunda línea</p>

<!-- Línea horizontal -->
<hr>
```

Los encabezados `<h1>` a `<h6>` crean una jerarquía de títulos, siendo `<h1>` el más importante. Solo debe haber un `<h1>` por página. Los elementos `<strong>` y `<em>` no solo cambian la apariencia sino que también indican importancia semántica.

#### 8.2.4. Listas

Las listas organizan información relacionada:

```

<!-- Lista no ordenada (viñetas) -->
<ul>
    <li>Primer elemento</li>
    <li>Segundo elemento</li>
    <li>Tercer elemento</li>
</ul>

<!-- Lista ordenada (numerada) -->
<ol>
    <li>Paso 1</li>
    <li>Paso 2</li>
    <li>Paso 3</li>
</ol>

<!-- Listas anidadas -->
<ul>
    <li>Jugadores
        <ul>
            <li>Jugador 1</li>
            <li>Jugador 2</li>
        </ul>
    </li>
    <li>Enemigos</li>
</ul>

```

`<ul>` crea listas no ordenadas (con viñetas), `<ol>` crea listas ordenadas (numeradas), y `<li>` define cada elemento de la lista. Las listas pueden anidarse para crear jerarquías.

## 8.2.5. Enlaces e Imágenes

```

<!-- Enlaces -->
<a href="https://www.ejemplo.com">Visitar Ejemplo</a>
<a href="pagina2.html">Ir a otra página del sitio</a>
<a href="#seccion">Ir a una sección de esta página</a>
<a href="documento.pdf" download>Descargar PDF</a>

<!-- Imágenes -->



```

El elemento `<a>` (anchor) crea enlaces. El atributo `href` especifica la URL de destino. Puede ser una URL completa, una ruta relativa a otro archivo, o un ancla a una sección de la misma página usando `#`.

El elemento `<img>` inserta imágenes. El atributo `src` indica la ruta de la imagen y `alt` proporciona texto alternativo para accesibilidad o cuando la imagen no carga. Los atributos `width` y `height` definen las dimensiones en píxeles.

---

### 8.2.6. Contenedores: Div y Span

Los elementos `<div>` y `<span>` son contenedores genéricos sin significado semántico, pero son fundamentales para organizar y dar estilo al contenido:

```
<!-- div: contenedor de bloque (ocupa todo el ancho disponible) -->
<div class="game-container">
  <h2>Mi Juego</h2>
  <p>Puntuación: 100</p>
</div>

<!-- span: contenedor en línea (solo ocupa el espacio de su contenido) -->
<p>Vidas: <span class="lives-count">3</span></p>
<p>Estado: <span id="status">Jugando</span></p>
```

`<div>` es un contenedor de nivel de bloque: cada `<div>` comienza en una nueva línea y ocupa todo el ancho disponible. Se usa para agrupar secciones grandes de contenido.

`<span>` es un contenedor en línea: no interrumpe el flujo del texto y solo ocupa el espacio necesario para su contenido. Se usa para aplicar estilos a partes específicas de texto.

---

### 8.2.7. Elementos Semánticos

HTML5 introdujo elementos semánticos que describen el propósito del contenido:

```

<header>
    <h1>Título del Sitio</h1>
    <nav>
        <a href="#inicio">Inicio</a>
        <a href="#juego">Juego</a>
        <a href="#contacto">Contacto</a>
    </nav>
</header>

<main>
    <section id="juego">
        <h2>Área de Juego</h2>
        <canvas id="game-canvas"></canvas>
    </section>

    <aside>
        <h3>Instrucciones</h3>
        <p>Usa las flechas para moverte.</p>
    </aside>
</main>

<footer>
    <p>&copy; 2024 Mi Juego</p>
</footer>

```

- `<header>` : Encabezado de la página o sección
- `<nav>` : Bloque de navegación con enlaces
- `<main>` : Contenido principal del documento (solo uno por página)
- `<section>` : Sección temática de contenido
- `<aside>` : Contenido complementario o lateral
- `<footer>` : Pie de página con información adicional

Estos elementos mejoran la accesibilidad y la estructura del código comparado con usar solo `<div>`.

### 8.2.8. Formularios

Los formularios permiten recopilar información del usuario:

```

<form id="player-form">
    <!-- Campo de texto -->
    <label for="name">Nombre:</label>
    <input type="text" id="name" name="player-name" placeholder="Tu nombre">

    <!-- Campo numérico -->
    <label for="age">Edad:</label>
    <input type="number" id="age" min="1" max="100">

    <!-- Botones de radio (una opción) -->
    <p>Dificultad:</p>
    <input type="radio" id="easy" name="difficulty" value="easy">
    <label for="easy">Fácil</label>

    <input type="radio" id="hard" name="difficulty" value="hard">
    <label for="hard">Difícil</label>

    <!-- Checkbox (múltiples opciones) -->
    <input type="checkbox" id="sound" name="sound" checked>
    <label for="sound">Activar sonido</label>

    <!-- Menú desplegable -->
    <label for="character">Personaje:</label>
    <select id="character" name="character">
        <option value="warrior">Guerrero</option>
        <option value="mage">Mago</option>
        <option value="archer">Arquero</option>
    </select>

    <!-- Botón de envío -->
    <button type="submit">Comenzar Juego</button>
</form>

```

El elemento `<label>` asocia texto descriptivo con un campo. El atributo `for` del label debe coincidir con el `id` del input. Los diferentes `type` de input determinan qué tipo de dato se espera: `text`, `number`, `email`, `password`, etc.

Los inputs `radio` con el mismo `name` forman un grupo donde solo se puede seleccionar una opción. Los `checkbox` permiten múltiples selecciones. El elemento `<select>` crea menús desplegables con opciones `<option>`.

## 8.2.9. Canvas (Elemento para Videojuegos)

El elemento `<canvas>` es fundamental para videojuegos web, proporcionando una superficie de dibujo que JavaScript puede manipular:

```
<canvas id="game-canvas" width="800" height="600"></canvas>
```

El canvas es un área rectangular donde se pueden dibujar gráficos, formas, imágenes y animaciones mediante JavaScript. Los atributos `width` y `height` definen su tamaño en píxeles.

### 8.2.10. Referencias a Scripts y Hojas de Estilo

Para integrar JavaScript y CSS con HTML:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Mi Juego</title>

    <!-- Enlazar CSS externo -->
    <link rel="stylesheet" href="css/styles.css">
    <link rel="stylesheet" href="css/game.css">

    <!-- CSS interno -->
    <style>
        body { margin: 0; padding: 0; }
    </style>
</head>
<body>
    <div id="game-container"></div>

    <!-- Script externo - al final del body -->
    <script src="js/game.js"></script>
    <script src="js/controls.js"></script>

    <!-- Script interno -->
    <script>
        console.log('Página cargada');
    </script>
</body>
</html>
```

**Hojas de estilo CSS:** - `<link rel="stylesheet" href="ruta.css">` en el `<head>` enlaza archivos CSS externos - `<style>` permite escribir CSS directamente en el HTML

**Scripts JavaScript:** - `<script src="ruta.js">` enlaza archivos JavaScript externos - Se recomienda colocarlos al final del `<body>` para que el HTML cargue primero - `<script>` sin `src` permite escribir JavaScript directamente en el HTML

---

### 8.2.11. Atributos Comunes

Los atributos proporcionan información adicional sobre los elementos:

```
<!-- id: identificador único -->
<div id="game-canvas"></div>

<!-- class: clasificación para estilos (puede repetirse) -->
<button class="menu-btn primary">Jugar</button>
<button class="menu-btn secondary">Salir</button>

<!-- style: estilos CSS inline -->
<p style="color: red; font-size: 20px;">Texto rojo</p>

<!-- data-*: atributos personalizados -->
<div class="enemy" data-type="orc" data-health="100"></div>

<!-- title: tooltip al pasar el mouse -->
<button title="Haz click para pausar">Pausa</button>
```

El `id` debe ser único en toda la página y se usa para identificar un elemento específico. La `class` puede repetirse en múltiples elementos y se usa para aplicar estilos compartidos. Los atributos `data-*` permiten almacenar información personalizada en elementos HTML.

## 8.3. CSS (Cascading Style Sheets)

---

### 8.3.1. ¿Qué es CSS?

CSS (Hojas de Estilo en Cascada) es el lenguaje que controla la presentación visual de HTML. Permite definir colores, tamaños, posiciones, animaciones y mucho más. CSS separa el contenido (HTML) de la presentación (estilos).

### 8.3.2. Sintaxis Básica

Una regla CSS consta de un **selector** que identifica qué elementos afectar, y **declaraciones** que definen los estilos:

```
selector {  
    propiedad: valor;  
    otra-propiedad: otro-valor;  
}
```

Ejemplo real:

```
h1 {  
    color: blue;  
    font-size: 32px;  
    text-align: center;  
}
```

Esto hace que todos los elementos `<h1>` sean azules, con tamaño de fuente de 32 píxeles, y centrados.

### 8.3.3. Formas de Incluir CSS

#### 1. CSS Externo (recomendado):

```
<link rel="stylesheet" href="estilos.css">
```

El archivo `estilos.css` contiene todas las reglas CSS. Es la mejor opción porque permite reutilizar estilos en múltiples páginas.

#### 2. CSS Interno:

```
<head>  
    <style>  
        p { color: green; }  
    </style>  
</head>
```

Los estilos se escriben dentro de `<style>` en el HTML. Útil para estilos específicos de una página.

### 3. CSS Inline:

```
<p style="color: red; font-size: 18px;">Texto rojo</p>
```

Los estilos se aplican directamente en el atributo `style` del elemento. No se recomienda excepto para estilos dinámicos con JavaScript.

#### 8.3.4. Selectores CSS

Los selectores determinan a qué elementos se aplican los estilos:

```
/* Selector de elemento - afecta a TODOS los elementos del tipo */
p {
    color: black;
}

/* Selector de ID - afecta al elemento con ese id específico */
#game-canvas {
    border: 2px solid black;
}

/* Selector de clase - afecta a todos los elementos con esa clase */
.menu-btn {
    background-color: blue;
    color: white;
}

/* Selector de múltiples elementos */
h1, h2, h3 {
    font-family: Arial, sans-serif;
}

/* Selector descendiente - elementos dentro de otros */
.game-container p {
    margin: 10px;
}

/* Selector hijo directo */
nav > a {
    display: inline-block;
}
```

- **Elemento** (`p`, `h1`, `div`): Selecciona todos los elementos de ese tipo
- **ID** (`#nombre`): Selecciona el elemento con `id="nombre"` (solo uno)
- **Clase** (`.nombre`): Selecciona todos los elementos con `class="nombre"`

- **Descendiente** (`div p`): Selecciona todos los `<p>` dentro de cualquier `<div>`
- **Hijo directo** (`div > p`): Selecciona solo los `<p>` que son hijos directos de `<div>`

### 8.3.5. Colores

CSS permite definir colores de varias formas. La propiedad `color` controla el color del texto, mientras que `background-color` controla el color de fondo del elemento.

```
.ejemplo {  
    /* Nombres de colores predefinidos */  
    color: red;  
    background-color: white;  
}
```

**Nombres de colores:** CSS reconoce 140 nombres de colores en inglés como `red`, `blue`, `green`, `black`, `white`, etc. Son fáciles de recordar pero ofrecen pocas opciones.

```
.hexadecimal {  
    /* Hexadecimal - formato más común */  
    color: #ff0000;          /* rojo: ff=255 rojo, 00=0 verde, 00=0 azul */  
    color: #00ff00;          /* verde */  
    color: #0000ff;          /* azul */  
    color: #fff;              /* blanco (forma corta de #ffffff) */  
    color: #000;              /* negro (forma corta de #000000) */  
}
```

**Hexadecimal:** El formato más usado. Los colores se representan con 6 dígitos hexadecimales (0-9, A-F) precedidos por `#`. Los primeros dos dígitos son el rojo, los siguientes dos el verde, y los últimos dos el azul. Cuando los pares de dígitos son iguales, se puede usar la forma corta de 3 dígitos: `#f00` es igual a `#ff0000`.

```
.rgb {  
    /* RGB - valores de 0 a 255 */  
    color: rgb(255, 0, 0);      /* rojo */  
    color: rgb(0, 255, 0);      /* verde */  
    color: rgb(128, 128, 128);  /* gris medio */  
}
```

**RGB (Red, Green, Blue):** Especifica colores usando tres números del 0 al 255 que representan la intensidad de rojo, verde y azul. Es más intuitivo que hexadecimal para algunas personas.

```
.transparencia {  
    /* RGBA - RGB con transparencia (alpha) */  
    background-color: rgba(0, 0, 0, 0.5);      /* negro semi-transparente */  
    background-color: rgba(255, 0, 0, 0.8);    /* rojo casi opaco */  
    background-color: rgba(0, 0, 255, 0.2);   /* azul muy transparente */  
}
```

**RGBA:** Igual que RGB pero añade un cuarto valor (alpha) que controla la transparencia. El valor alpha va de 0 (completamente transparente) a 1 (completamente opaco). Un valor de 0.5 significa 50% transparente.

### 8.3.6. Texto y Fuentes

CSS ofrece múltiples propiedades para controlar la apariencia del texto:

```
.titulo {  
    font-family: Arial, sans-serif;  
}
```

**font-family:** Define qué tipografía se usa. Se especifica una lista de fuentes separadas por comas porque no todos los sistemas tienen todas las fuentes instaladas. El navegador usa la primera fuente disponible. Es recomendable terminar con una familia genérica (`serif`, `sans-serif`, `monospace`) como respaldo.

```
.texto {  
    font-size: 16px;        /* Tamaño absoluto en píxeles */  
    font-size:  
        1.5em;           /* Tamaño relativo al parent (1.5 veces más grande) */  
    font-size: 100%;       /* Tamaño relativo (100% = igual al parent) */  
}
```

**font-size:** Controla el tamaño del texto. Las unidades más comunes son píxeles (`px`), que son fijas, y unidades relativas como `em` (relativa al tamaño de fuente del elemento parent) o porcentajes.

```
.negrita {  
    font-weight: bold;      /* Negrita */  
    font-weight: normal;    /* Normal */  
    font-weight: 700;        /* Negrita (valor numérico) */  
    font-weight: 400;        /* Normal (valor numérico) */  
}
```

**font-weight:** Controla el grosor del texto. Puede usar palabras clave (`normal`, `bold`) o valores numéricos de 100 a 900, donde 400 es normal y 700 es negrita.

```
.cursiva {  
    font-style: italic; /* Cursiva */  
    font-style: normal; /* Normal (no cursiva) */  
}
```

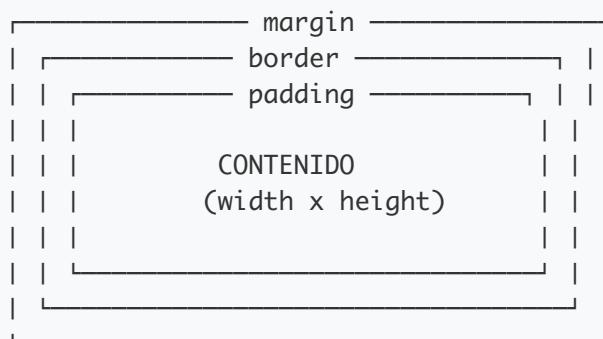
**font-style:** Define si el texto está en cursiva (`italic`) o normal.

```
.alineacion {  
    text-align: left; /* Alineado a la izquierda (predeterminado) */  
    text-align: center; /* Centrado */  
    text-align: right; /* Alineado a la derecha */  
    text-align: justify; /* Justificado (alineado a ambos lados) */  
}
```

**text-align:** Controla la alineación horizontal del texto dentro de su contenedor.

### 8.3.7. Modelo de Caja (Box Model)

El concepto más importante de CSS es el **modelo de caja**. Cada elemento HTML es una caja rectangular compuesta por cuatro áreas que se suman para determinar el espacio total que ocupa:



**1. Contenido (content):** Es el área donde aparece el texto, imágenes o contenido del elemento. Se controla con `width` (ancho) y `height` (alto).

```
.contenido {  
    width: 200px; /* Ancho del área de contenido */  
    height: 100px; /* Alto del área de contenido */  
}
```

**2. Relleno (padding):** Es el espacio entre el contenido y el borde. Es transparente y adopta el color de fondo del elemento. Sirve para dar "respiración" al contenido.

```
.con-padding {  
    padding: 20px; /* 20px de espacio en los 4 lados */  
}  
  
.padding-individual {  
    padding-top: 10px; /* Arriba */  
    padding-right: 15px; /* Derecha */  
    padding-bottom: 10px; /* Abajo */  
    padding-left: 15px; /* Izquierda */  
}  
  
.padding-forma-corta {  
    /* Forma corta: arriba derecha abajo izquierda (sentido horario) */  
    padding: 10px 15px 10px 15px;  
  
    /* Si vertical y horizontal son iguales, se puede reducir a 2 valores */  
    padding: 10px 15px; /* 10px arriba/abajo, 15px izquierda/derecha */  
}
```

**3. Borde (border):** Es una línea que rodea el padding y el contenido. Se puede definir su grosor, estilo y color.

```
.con-borde {  
    border: 2px solid black; /* Grosor Estilo Color */  
}  
  
.borde-detallado {  
    border-width: 3px; /* Grosor del borde */  
    border-style: solid; /* Estilo: solid, dashed, dotted, double */  
    border-color: blue; /* Color del borde */  
    border-radius: 10px; /* Esquinas redondeadas */  
}  
  
.bordes-individuales {  
    border-top: 1px solid red; /* Solo borde superior */  
    border-bottom: 2px dashed blue; /* Solo borde inferior */  
}
```

**4. Margen (margin):** Es el espacio externo entre el elemento y otros elementos. Es completamente transparente y sirve para separar elementos entre sí.

```
.con-margen {  
    margin: 20px; /* 20px de espacio exterior en los 4 lados */  
}  
  
.margen-individual {  
    margin-top: 10px;  
    margin-right: 15px;  
    margin-bottom: 10px;  
    margin-left: 15px;  
}  
  
.centrado {  
    width: 200px;  
    margin: 0 auto; /* Centra horizontalmente: 0 arriba/abajo, auto  
izquierda/derecha */  
}
```

### Cálculo del tamaño total:

Por defecto, el tamaño total de un elemento se calcula así:

Ancho total = margin-left + border-left + padding-left + width + padding-right + border-right + margin-right

Alto total = margin-top + border-top + padding-top + height + padding-bottom + border-bottom + margin-bottom

Ejemplo: si un elemento tiene `width: 200px`, `padding: 10px`, `border: 2px`, el ancho visible será 224px (2 + 10 + 200 + 10 + 2).

**Box-sizing:** Para simplificar este cálculo, existe la propiedad `box-sizing`:

```

.caja-tradicional {
    box-sizing: content-box; /* Predeterminado: width solo afecta al
                               contenido */
    width: 200px;
    padding: 10px;
    border: 2px;
    /* Ancho total = 224px (2+10+200+10+2) */
}

.caja-moderna {
    box-sizing: border-
        box; /* width incluye contenido + padding + border */
    width: 200px;
    padding: 10px;
    border: 2px;
    /* Ancho total = 200px (el padding y border se restan del contenido) */
}

```

La mayoría de desarrolladores modernos usan `box-sizing: border-box` porque hace los cálculos más intuitivos. Es común ver esto al inicio de las hojas de estilo:

```

* {
    box-sizing: border-box; /* Aplica a todos los elementos */
}

```

### 8.3.8. Display: Bloque vs Inline

La propiedad `display` es fundamental porque controla cómo se comporta un elemento en el flujo del documento. Los dos valores principales son `block` e `inline`.

#### **Elementos de bloque (block):**

Los elementos de bloque ocupan todo el ancho disponible de su contenedor y siempre comienzan en una nueva línea. Son como cajas apiladas verticalmente.

```

div {
    display: block; /* Comportamiento predeterminado de div */
}

```

Elementos que son `block` por defecto: `<div>`, `<p>`, `<h1>-<h6>`, `<ul>`, `<li>`, `<section>`, `<header>`, etc.

Características: - Ocupan todo el ancho disponible - Empiezan en una nueva línea - Aceptan propiedades de ancho (`width`) y alto (`height`) - Se pueden apilar verticalmente con márgenes

### Elementos en línea (inline):

Los elementos inline solo ocupan el espacio de su contenido y no interrumpen el flujo del texto. Son como palabras en una oración.

```
span {  
    display: inline; /* Comportamiento predeterminado de span */  
}
```

Elementos que son inline por defecto: `<span>` , `<a>` , `<strong>` , `<em>` , `<img>` , etc.

Características: - Solo ocupan el espacio de su contenido - No interrumpen el flujo del texto - NO aceptan `width` ni `height` - Los márgenes verticales no funcionan correctamente

### Inline-block (lo mejor de ambos mundos):

```
.boton {  
    display: inline-block;  
    width: 100px;  
    height: 40px;  
}
```

`inline-block` combina ambos comportamientos: el elemento fluye como inline (no interrumpe la línea) pero acepta dimensiones como block.

### Ocultar elementos:

```
.oculto {  
    display: none; /* Elimina completamente el elemento del flujo */  
}  
  
.invisible {  
    visibility: hidden; /* Oculta el elemento pero mantiene su espacio */  
}
```

### 8.3.9. Posicionamiento

El posicionamiento controla dónde aparece un elemento en la página. Por defecto, los elementos siguen el flujo normal del documento (uno después de otro), pero podemos cambiar esto.

#### Static (posición normal):

```
.normal {  
    position: static; /* Valor predeterminado */  
}
```

El elemento sigue el flujo normal del documento. Las propiedades `top`, `right`, `bottom`, `left` no tienen efecto.

#### Relative (posición relativa):

```
.desplazado {  
    position: relative;  
    top: 10px; /* Se mueve 10px HACIA ABAJO desde su posición original */  
    left: 20px; /* Se mueve 20px HACIA LA DERECHA desde su posición original */  
}
```

El elemento se desplaza desde su posición original, pero su espacio original se mantiene reservado. Otros elementos se comportan como si este elemento siguiera en su lugar original.

#### Absolute (posición absoluta):

```
.contenedor {  
    position: relative; /* Esto es importante */  
}  
  
.flotante {  
    position: absolute;  
    top: 20px; /* 20px desde el BORDE SUPERIOR del contenedor */  
    right: 10px; /* 10px desde el BORDE DERECHO del contenedor */  
}
```

El elemento se extrae completamente del flujo normal y se posiciona respecto al ancestro posicionado más cercano (cualquier elemento con `position` diferente de `static`). Si no hay ninguno, se posiciona respecto al `<body>`.

Usos comunes: - Menús desplegables - Tooltips - Elementos superpuestos sobre otros

#### Fixed (posición fija):

```
.barra-superior {  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    background-color: black;  
}
```

El elemento se posiciona respecto a la ventana del navegador y permanece en el mismo lugar incluso al hacer scroll. Útil para barras de navegación fijas o botones flotantes.

#### Z-index (orden de apilamiento):

Cuando los elementos se superponen, `z-index` controla cuál aparece encima:

```
.fondo {  
    position: absolute;  
    z-index: 1; /* Aparece debajo */  
}  
  
.encima {  
    position: absolute;  
    z-index: 10; /* Aparece encima (número mayor = más arriba) */  
}
```

**Importante:** `z-index` solo funciona en elementos posicionados (con `position` diferente de `static`).

---

### 8.3.10. Pseudo-clases

Las pseudo-clases permiten aplicar estilos a elementos según su estado o posición, sin necesidad de añadir clases en el HTML.

#### Estados interactivos:

```
/* Cuando el mouse pasa sobre el elemento */
a:hover {
    color: red;
    text-decoration: underline;
}

/* Mientras se hace click en el elemento */
button:active {
    background-color: darkblue;
}

/* Cuando el elemento tiene el foco (navegación con teclado) */
input:focus {
    border-color: blue;
    outline: 2px solid lightblue;
}
```

La pseudo-clase `:hover` es muy útil para dar feedback visual al usuario. `:active` captura el momento exacto del click. `:focus` es importante para accesibilidad cuando se navega con teclado.

### Selectores de posición:

```
/* El primer elemento de su tipo */
li:first-child {
    font-weight: bold;
}

/* El último elemento de su tipo */
li:last-child {
    border-bottom: none;
}

/* Elementos pares o impares */
tr:nth-child(even) {
    background-color: #f0f0f0; /* Filas pares en gris claro */
}

tr:nth-child(odd) {
    background-color: white; /* Filas impares en blanco */
}
```

Estas pseudo-clases son útiles para crear efectos visuales como tablas con filas alternadas sin tener que añadir clases manualmente a cada elemento.

## 8.4. Integración HTML, CSS y JavaScript

Estas tres tecnologías trabajan juntas en el desarrollo web:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Ejemplo Integrado</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div id="game-container">
        <h1>Mi Juego</h1>
        <div id="score">Puntuación: <span id="score-value">0</span></div>
        <button id="start-btn">Iniciar</button>
    </div>

    <script src="game.js"></script>
</body>
</html>
```

```

/* styles.css */
#game-container {
    text-align: center;
    padding: 20px;
    background-color: #f0f0f0;
}

#score {
    font-size: 24px;
    margin: 20px 0;
}

#score-value {
    color: green;
    font-weight: bold;
}

#start-btn {
    padding: 10px 30px;
    font-size: 18px;
    background-color: #4CAF50;
    color: white;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

#start-btn:hover {
    background-color: #45a049;
}

```

```

// game.js
const scoreElement = document.getElementById('score-value');
const startBtn = document.getElementById('start-btn');

let score = 0;

startBtn.addEventListener('click', function() {
    score += 10;
    scoreElement.textContent = score;
});

```

En este ejemplo: HTML define la estructura (título, puntuación, botón), CSS controla la apariencia (colores, tamaños, efectos hover), y JavaScript añade interactividad (actualiza la puntuación al hacer click).

## Desarrollo de juegos con tecnología web

---

Con los fundamentos de JavaScript dominados, estamos listos para dar el siguiente paso: crear juegos completos utilizando tecnología web. Phaser es un motor de juegos 2D potente y versátil que nos permitirá desarrollar juegos directamente en el navegador, aprovechando todo el conocimiento de JavaScript que hemos adquirido. En esta parte exploraremos la arquitectura de Phaser, aprenderemos a gestionar escenas y el ciclo de vida del juego, trabajaremos con sprites y texturas, implementaremos sistemas de físicas, y dominaremos la gestión de entrada del usuario mediante teclado, ratón y dispositivos táctiles. Al finalizar, seremos capaces de desarrollar juegos 2D completos que servirán como base para integrar las funcionalidades multijugador en las siguientes partes del curso.

# 9. Phaser 3

---

En el capítulo anterior hemos aprendido los fundamentos de JavaScript, incluyendo su orientación a objetos, el manejo del DOM y BOM, y cómo configurar un entorno de desarrollo moderno con Node.js y Webpack. Ahora vamos a aplicar todos estos conocimientos para crear videojuegos web utilizando Phaser 3, uno de los frameworks más populares y potentes para el desarrollo de juegos en navegadores.

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. Phaser aprovecha todo lo que hemos aprendido sobre JavaScript: utiliza clases ES2015+ para estructurar el código, manipula el DOM para renderizar gráficos en el canvas, y se integra perfectamente con las herramientas de desarrollo modernas como npm y Webpack que ya conocemos.

## 9.1. 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.

## 9.2. Requisitos para empezar

Para comenzar a desarrollar con Phaser 3 necesitamos:

---

### 9.2.1. 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.

---

### 9.2.2. 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>
```

---

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

---

### 9.3.1. 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 300×300 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.

---

### 9.3.2. 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>
```

<!-- También podemos cargar directamente el juego con la creación, etc. Recomendado. -->

```
<script src="bundle.js"></script>
</body>
</html>
```

### 9.3.2.1. Objeto de configuración

El objeto `config` es el punto de partida de cualquier juego en Phaser 3. Este objeto JavaScript define todos los parámetros fundamentales del juego: cómo se renderizará, qué tamaño tendrá, qué motor de físicas usará, y qué escenas contendrá. Es el equivalente a la “configuración inicial” o “settings” del juego.

Veamos un ejemplo completo con las opciones más comunes:

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    backgroundColor: '#2d2d2d',
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    },
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);
```

#### Explicación detallada de cada propiedad:

- `type`: Especifica el sistema de renderizado que Phaser usará para dibujar los gráficos. Las opciones son:
  - `Phaser.AUTO`: Phaser elige automáticamente el mejor disponible (intenta WebGL primero, si no está disponible usa Canvas)
  - `Phaser.WEBGL`: Fuerza el uso de WebGL (más rápido, gráficos avanzados, requiere soporte GPU)
  - `Phaser.CANVAS`: Fuerza el uso de Canvas 2D (más compatible, funciona en dispositivos antiguos)

**Recomendación:** Usar `Phaser.AUTO` en la mayoría de casos para máxima compatibilidad.

- `width` y `height`: Dimensiones del canvas en píxeles. Definen el área de juego visible. En el ejemplo, 800×600 píxeles es un tamaño clásico que funciona bien en la mayoría de pantallas. Otras opciones comunes:
  - 1024×768 para juegos más grandes
  - 320×240 para juegos retro de pixel art
  - Se puede usar `scale` para hacer el juego responsive (más sobre esto en configuración avanzada)
- `backgroundColor`: Color de fondo del canvas en formato hexadecimal. Si no se especifica, será negro por defecto. Útil para dar un estilo visual inicial antes de cargar recursos.
- `physics`: Objeto de configuración del motor de físicas. Tiene dos propiedades principales:
  - `default`: Motor de físicas a usar ('`arcade`', '`impact`', o '`matter`'). `arcade` es el más usado por su simplicidad y rendimiento.
  - `arcade`: Objeto con configuración específica del motor `Arcade Physics`:
    - `gravity`: Objeto con `x` e `y` que define la gravedad en píxeles/segundo<sup>2</sup>. `{ y: 300 }` simula gravedad terrestre hacia abajo. Si fuera `{ y: -300 }` sería hacia arriba.
    - `debug`: Boolean. Si es `true`, muestra los contornos de los cuerpos físicos en verde/azul, muy útil durante desarrollo para visualizar colisiones.
- `scene`: Define las escenas del juego. Puede ser:
  - Un objeto con las funciones `preload`, `create`, `update` (forma simple para juegos pequeños)
  - Un array de clases de escena: `scene: [MenuPrincipal, Juego, GameOver]` (forma recomendada para juegos más complejos)
  - Una sola clase: `scene: MiEscena`

### 9.3.2.2. Las tres funciones principales

Todo juego en Phaser 3 se estructura alrededor de tres funciones fundamentales que se ejecutan en un orden específico y cumplen roles claramente diferenciados. Estas funciones forman el ciclo de vida básico de una escena del juego.

#### `preload()` - Carga de recursos

Esta es la primera función que se ejecuta cuando se inicia una escena. Su propósito exclusivo es cargar todos los recursos (assets) que necesitará el juego: imágenes, spritesheets, audio, datos JSON, tilemaps, etc. Phaser ejecuta esta función y espera a que todos los recursos se carguen completamente antes de continuar con `create()`.

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

    // Cargar spritesheets para animaciones
    this.load.spritesheet('jugador', 'assets/jugador.png', {
        frameWidth: 32,
        frameHeight: 48
    });

    // Cargar audio
    this.load.audio('musica', 'assets/musica.mp3');
}
```

Cada recurso se identifica con una clave única ('cielo', 'suelo', 'estrella') que usaremos después para referenciarlos en el juego. Es crucial cargar todos los recursos aquí para evitar problemas de recursos no disponibles durante el juego.

**Otros tipos de recursos:** Phaser 3 soporta muchos más tipos de recursos como atlas de texturas, tilemaps, fuentes de bitmap, videos, plugins, shaders y más. Para una lista completa de métodos de carga, consulta la [documentación oficial de Phaser 3 Loader](#).

### **create() - Inicialización del juego**

Se ejecuta una sola vez después de que `preload()` haya terminado de cargar todos los recursos. Aquí creamos e inicializamos todos los objetos del juego: colocamos sprites, creamos grupos, configuramos físicas, establecemos colisiones, inicializamos variables, y preparamos todo el estado inicial del juego.

```

function create() {
    // Añadir fondo (imagen estática sin físicas)
    this.add.image(400, 300, 'cielo');

    // Crear plataforma con físicas
    var plataforma = this.physics.add.image(400, 500, 'suelo');
    plataforma.setImmovable(true);           // No se mueve al colisionar
    plataforma.setCollideWorldBounds(true); // No sale de los límites

    // Crear jugador
    this.jugador = this.physics.add.sprite(100, 450, 'jugador');
    this.jugador.setBounce(0.2);
    this.jugador.setCollideWorldBounds(true);

    // Configurar colisión entre jugador y plataforma
    this.physics.add.collider(this.jugador, plataforma);

    // Configurar controles
    this.cursors = this.input.keyboard.createCursorKeys();

    // Inicializar puntuación
    this.puntuacion = 0;
    this.textoPuntuacion = this.add.text(16, 16, 'Puntos: 0', {
        fontSize: '32px',
        fill: '#000'
    });
}

```

El orden de creación es importante: los elementos se renderizan en el orden en que se añaden, por lo que si queremos que algo aparezca sobre otra cosa, debemos crearlo después. Observa cómo primero añadimos el fondo, luego las plataformas y finalmente el jugador y el texto, asegurando que cada elemento aparezca en la capa visual correcta.

### **update(time, delta) - Bucle principal del juego**

Esta función se ejecuta continuamente, aproximadamente 60 veces por segundo (60 FPS), formando el bucle principal del juego (game loop). Aquí colocamos toda la lógica que debe actualizarse constantemente: movimiento del jugador, comportamiento de enemigos, detección de condiciones de victoria o derrota, actualización de puntuación, etc.

Los parámetros son:

- `time` : Tiempo total transcurrido desde que se inició el juego, en milisegundos
- `delta` : Tiempo transcurrido desde el último frame, en milisegundos (normalmente ~16ms a 60 FPS)

```

function update(time, delta) {
    // Resetear velocidad horizontal
    this.jugador.setVelocityX(0);

    // Movimiento izquierda/derecha
    if (this.cursors.left.isDown) {
        this.jugador.setVelocityX(-160);
        this.jugador.anims.play('izquierda', true);
    } else if (this.cursors.right.isDown) {
        this.jugador.setVelocityX(160);
        this.jugador.anims.play('derecha', true);
    } else {
        this.jugador.anims.play('quieto');
    }

    // Saltar solo si está tocando el suelo
    if (this.cursors.up.isDown && this.jugador.body.touching.down) {
        this.jugador.setVelocityY(-330);
    }

    // Usar delta para movimiento independiente de framerate
    // this.enemigo.x += (100 * delta / 1000); // 100 píxeles por segundo
}

```

Es importante resetear velocidades o estados al inicio de `update()` para evitar que se acumulen. En el ejemplo, reseteamos la velocidad horizontal a 0 para que el jugador se detenga cuando no se presiona ninguna tecla.

El parámetro `delta` es especialmente útil para movimientos independientes del framerate: si multiplicamos velocidades por `delta / 1000`, el movimiento será consistente incluso si el juego no mantiene exactamente 60 FPS. Esto es crucial para que el juego se comporte igual en diferentes dispositivos con distintas capacidades de procesamiento.

### Resumen del flujo de ejecución:

1. **preload()** → Carga recursos una vez al inicio
2. **create()** → Inicializa objetos y configuración una vez después de la carga
3. **update()** → Se ejecuta continuamente 60 veces por segundo durante todo el juego

## 9.4. Gestión de escenas

---

### 9.4.1. Concepto de escena

Una escena en Phaser 3 es una unidad autónoma y autocontenido que representa un estado o pantalla específica del juego. Podemos pensar en las escenas como los diferentes "momentos" o "pantallas" por los que pasa un jugador: el menú principal, la pantalla de carga, cada nivel del juego, el menú de pausa, la pantalla de game over, la tabla de puntuaciones, etc.

Lo revolucionario del sistema de escenas es que cada una es completamente independiente: tiene su propio ciclo de vida con sus propias funciones `preload()`, `create()` y `update()`, mantiene sus propios objetos del juego (sprites, grupos, texto, etc.), gestiona su propia lógica, y puede cargar sus propios recursos específicos. Esta arquitectura modular tiene ventajas enormes:

**Organización del código:** En lugar de tener todo el código del juego en un único archivo gigante, podemos separarlo en escenas lógicas. Por ejemplo, `MenuPrincipal.js`, `Nivel1.js`, `Nivel2.js`, `GameOver.js`. Esto hace que el código sea mucho más fácil de entender, mantener y depurar.

**Reutilización:** Podemos tener una escena genérica de pausa que se lance sobre cualquier nivel de juego, o una escena de transición que se use entre diferentes niveles.

**Gestión de memoria:** Cuando una escena termina, Phaser puede limpiar automáticamente todos sus recursos, liberando memoria. Esto es crucial para juegos grandes con muchos niveles.

**Desarrollo en equipo:** Diferentes programadores pueden trabajar en diferentes escenas simultáneamente sin pisarse el código entre sí.

**Ejemplo de estructura típica de un juego:**

```

// Escena 1: Pantalla de carga inicial
class Boot extends Phaser.Scene {
    preload() {
        // Cargar recursos mínimos (logo, barra de carga)
    }
    create() {
        this.scene.start('MenuPrincipal');
    }
}

// Escena 2: Menú principal
class MenuPrincipal extends Phaser.Scene {
    create() {
        // Mostrar título, botones de jugar, opciones, créditos
    }
}

// Escena 3: El juego real
class Nivel1 extends Phaser.Scene {
    preload() {
        // Cargar recursos específicos del nivel 1
    }
    create() {
        // Crear plataformas, enemigos, jugador
    }
    update() {
        // Lógica del juego
    }
}

// Escena 4: Pantalla final
class GameOver extends Phaser.Scene {
    create() {
        // Mostrar puntuación final, botón para reiniciar
    }
}

```

Cada escena es una clase que extiende `Phaser.Scene`, permitiendo aprovechar la orientación a objetos de JavaScript que aprendimos en el capítulo anterior.

## 9.4.2. SceneManager

Phaser 3 incluye un `SceneManager` que se encarga de gestionar el ciclo de vida de todas las escenas del juego: iniciarlas, pausarlas, detenerlas y cambiar entre ellas. Este gestor nos proporciona una API sencilla para controlar el flujo del juego.

Para configurar las escenas de nuestro juego, las definimos en el objeto de configuración inicial:

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 }
        }
    },
    scene: [Boot, MenuPrincipal, Nivel1, Nivel2, GameOver]
};

var game = new Phaser.Game(config);
```

El array de escenas define todas las escenas disponibles en el juego. **Importante:** La primera escena del array ( `Boot` en este ejemplo) es la que se iniciará automáticamente cuando arranque el juego. El orden de las demás escenas no importa, ya que las activaremos explícitamente cuando las necesitemos.

---

#### 9.4.3. 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.

#### 9.4.3.1. 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.

#### 9.4.3.2. 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.

## 9.5. Trabajo con imágenes

### 9.5.1. 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.

### 9.5.2. 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.

### 9.5.3. 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);  
  
    // Voltear horizontalmente  
    jugador.flipX = true;  
  
    // Voltear 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.

## 9.6. Motores de físicas en Phaser 3

Los motores de físicas son componentes esenciales en el desarrollo de videojuegos que simulan el comportamiento del mundo real: gravedad, colisiones, rebotes, fricción y fuerzas. Sin un motor de físicas, tendríamos que programar manualmente cada uno de estos comportamientos, lo cual sería extremadamente complejo y propenso a errores. Un motor de físicas nos proporciona estas funcionalidades de manera optimizada y consistente.

Phaser 3 incluye tres motores de físicas diferentes, cada uno diseñado para diferentes niveles de complejidad y casos de uso específicos. La elección del motor adecuado dependerá del tipo de juego que estemos desarrollando y del nivel derealismo físico que necesitemos.

### 9.6.1. 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.

En todos los ejemplos usaremos el motor Arcade, para los otros motores podría ser necesario hacer cambios.

### 9.6.2. 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

### 9.6.3. 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

### 9.6.4. 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.

## 9.7. Sistema de entrada (Input)

La interacción del jugador es el corazón de cualquier videojuego. En el capítulo de JavaScript aprendimos sobre el sistema de eventos del DOM para detectar clics y pulsaciones de teclas. Phaser 3 construye sobre estos conceptos y proporciona un sistema de entrada (input) mucho más potente y especializado para videojuegos, que simplifica la detección de teclado, ratón, táctil y gamepad.

El sistema de input de Phaser 3 unifica diferentes tipos de entrada bajo una API consistente, permitiendo que nuestro juego funcione tanto en escritorio (con teclado y ratón) como en dispositivos móviles (con pantalla táctil) sin necesidad de escribir código específico para cada plataforma. Además, Phaser optimiza la detección de entrada para el contexto de videojuegos, donde necesitamos respuestas instantáneas y la capacidad de detectar múltiples entradas simultáneas.

### 9.7.1. Teclado

#### 9.7.1.1. 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.

### 9.7.1.2. 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.

### 9.7.1.3. 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.

#### 9.7.1.4. 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.

## 9.7.2. Ratón

### 9.7.2.1. 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.

## 9.8. Detección de colisiones

La detección de colisiones es uno de los aspectos más fundamentales en el desarrollo de videojuegos. Determina cuándo dos objetos del juego se tocan o superponen, permitiendo implementar mecánicas como que el jugador camine sobre plataformas, recoja objetos, reciba daño de enemigos, o dispare proyectiles que impacten en objetivos. Sin un sistema robusto de detección de colisiones, los objetos del juego simplemente se atravesarían unos a otros sin ningún tipo de interacción.

Phaser 3, a través de su sistema Arcade Physics, proporciona herramientas optimizadas para detectar y responder a colisiones de manera eficiente. El sistema está diseñado para manejar desde colisiones simples entre dos objetos hasta complejas interacciones entre grupos de cientos de entidades, todo ello manteniendo un rendimiento fluido de 60 frames por segundo.

---

### 9.8.1. Conceptos básicos

Phaser 3 Arcade Physics ofrece dos formas fundamentales 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.

---

### 9.8.2. 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.

## 9.8.3. Trabajar con grupos

### 9.8.3.1. 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.

### 9.8.3.2. 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.

---

## 9.8.4. Callbacks y condiciones

### 9.8.4.1. 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.

### 9.8.4.2. 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.puedeRecibirDanio,
        this
    );
}

function puedeRecibirDanio(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 `puedeRecibirDanio` se ejecuta antes de la colisión y determina si debe procesarse según el estado de invulnerabilidad.

### 9.8.5. 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.

### 9.8.6. 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.

## Desarrollo en el lado del servidor

---

Para crear juegos multijugador verdaderamente funcionales, necesitamos implementar el lado del servidor que gestione la lógica del juego, la persistencia de datos y las interacciones entre jugadores. En esta parte del curso nos sumergiremos en el desarrollo de APIs REST utilizando Node.js y Express, aprendiendo a diseñar endpoints eficientes, implementar middleware, gestionar errores y validar datos. Comenzaremos comprendiendo los principios de la arquitectura REST y los métodos HTTP fundamentales, para luego consumir APIs desde el cliente utilizando la Fetch API y técnicas asíncronas como promesas y `async/await`. Finalmente, desarrollaremos nuestras propias APIs REST completas que servirán como backend para nuestros juegos, manejando la lógica de negocio y la persistencia de información del juego de forma robusta y escalable.

# 10. API REST

---

## 10.1. Introducción

En el desarrollo de juegos multijugador modernos, la comunicación entre el cliente y el servidor es fundamental. Después de haber implementado un juego con Phaser 3 el siguiente paso natural es conectar nuestro juego con un servidor mediante una API REST.

En una aplicación web tradicional, el cliente (navegador) se comunica con el servidor utilizando el protocolo HTTP. En aplicaciones web sin AJAX, las peticiones HTTP devuelven un documento HTML que será visualizado por el navegador. Sin embargo, en las aplicaciones con AJAX y las aplicaciones SPA (Single Page Application), las peticiones HTTP se utilizan para intercambiar información entre el navegador y el servidor, pero no HTML.

Las APIs REST permiten esta comunicación mediante el intercambio de datos estructurados. En el contexto de nuestro juego, necesitaremos la API REST para gestionar registro y autenticación de jugadores, almacenamiento de puntuaciones y estadísticas, gestión de partidas y salas de juego, configuración de perfiles de usuario, y consulta de rankings. Posteriormente, cuando implementemos WebSockets, usaremos esa tecnología para la comunicación en tiempo real durante el juego, pero la API REST seguirá siendo esencial para todas las operaciones que no requieren actualización instantánea.

Imaginemos nuestra aplicación de juego haciendo una petición HTTP para obtener información de un jugador:

```
Petición:  
GET http://www.mygame.com/players/alice  
  
Respuesta:  
{  
  "id": "alice",  
  "name": "Alice Smith",  
  "level": 15,  
  "score": 8500,  
  "achievements": ["first_win", "speed_demon"]  
}
```

## 10.2. ¿Qué es una API REST?

REST es un estilo de arquitectura de software basado en HTTP que sigue una serie de principios específicos. Cuando un servicio web cumple estos principios, se denomina RESTful. Una API REST es el servicio que devuelve un servidor web cuando quiere intercambiar información estructurada en lugar de devolver HTML.

El término REST fue acuñado en el año 2000 por **Roy Fielding**, uno de los principales autores de la especificación del protocolo HTTP, en su tesis doctoral. Aunque existen otros tipos de servicios web como SOAP (basados en XML y mucho más complejos), las APIs REST se han convertido en el estándar de facto debido a su simplicidad y eficiencia.

---

### 10.2.1. Niveles de madurez REST

Existe un modelo de madurez de Richardson que clasifica las APIs en 4 niveles según su conformidad con los principios REST:

**Nivel 0 - The Swamp of POX (Plain Old XML):** En este nivel más básico, se usa HTTP simplemente como un sistema de transporte para llamadas a procedimientos remotos. Toda la comunicación se realiza típicamente mediante peticiones POST a una única URL, y el cuerpo de la petición contiene información sobre qué operación realizar. Es similar a usar HTTP como un túnel para otros protocolos (como SOAP). No se aprovecha ninguna característica del protocolo HTTP más allá del transporte.

**Nivel 1 - Resources:** En este nivel, la API introduce el concepto de recursos individuales. En lugar de tener una única URL para todo, cada recurso tiene su propia URI específica. Por ejemplo, `/players/alice` y `/matches/123` son URIs distintas para recursos diferentes. Sin embargo, todavía se usan métodos HTTP de forma incorrecta, típicamente solo POST para todas las operaciones, sin aprovechar la semántica de GET, PUT, DELETE, etc.

**Nivel 2 - HTTP Verbs:** Este es el nivel más común y el que usaremos en este curso. Aquí se utilizan correctamente los métodos HTTP (GET para obtener, POST para crear, PUT para actualizar, DELETE para eliminar) y los códigos de estado HTTP (200 para éxito, 404 para no encontrado, 500 para error del servidor, etc.). Este nivel aprovecha plenamente la semántica del protocolo HTTP, haciendo que las APIs sean más intuitivas y fáciles de usar.

**Nivel 3 - Hypermedia Controls (HATEOAS):** El nivel más avanzado, donde las respuestas incluyen enlaces hipermedia que guían al cliente sobre qué acciones puede realizar a continuación. Por ejemplo, al obtener información de un jugador, la respuesta incluiría enlaces a recursos relacionados como sus partidas, amigos, o acciones disponibles. Este nivel permite que los clientes descubran dinámicamente la API, aunque en la práctica pocas APIs implementan HATEOAS por su complejidad.

En este curso nos centraremos en el Nivel 2 (HTTP Verbs), que es el enfoque más habitual y práctico, donde se hace uso correcto de los métodos HTTP (GET, POST, PUT, DELETE) y códigos de estado.

Un servicio REST ofrece operaciones CRUD (Create, Read, Update, Delete) sobre recursos (ítems de información) del servidor web. Se aprovecha de todos los aspectos del protocolo HTTP: URL, métodos, códigos de estado, cabeceras, etc. La información se intercambia típicamente en formato JSON.

## 10.3. Casos de Uso de APIs REST

Además de navegadores web, muchos otros tipos de aplicaciones utilizan APIs REST. Las aplicaciones móviles son consumidoras intensivas: la aplicación de Google Maps para Android usa la misma API REST que la versión web, permitiendo mantener una única fuente de verdad en el servidor y consistencia entre plataformas. En comunicación Backend a Backend, un servidor puede consumir APIs REST de otros servidores; por ejemplo, el Aula Virtual de la URJC podría usar la API REST de Google Calendar para publicar eventos. Los videojuegos multijugador utilizan APIs REST para sincronizar estado, gestionar perfiles, implementar matchmaking, guardar partidas en la nube y sistemas de logros, incluyendo juegos en consolas y navegador. Finalmente, dispositivos IoT como Smart TVs, wearables y dispositivos embebidos usan APIs REST para comunicarse con servicios en la nube.

## 10.4. Formato JSON

JSON (JavaScript Object Notation) es el formato estándar para el intercambio de datos en APIs REST. Es un formato de texto ligero, fácil de leer para humanos y fácil de parsear para máquinas. JSON construye estructuras con dos tipos principales: objetos (colección de pares clave-valor en `{}`) y arrays (lista ordenada en `[]`). Los valores pueden ser cadenas de texto, números, booleanos, `null`, objetos o arrays.

```
{
  "game": {
    "id": "match_12345",
    "type": "pong",
    "status": "active",
    "players": [
      {
        "id": "alice",
        "name": "Alice Smith",
        "score": 5,
        "ready": true
      },
      {
        "id": "bob",
        "name": "Bob Johnson",
        "score": 3,
        "ready": true
      }
    ],
    "settings": {
      "max_score": 11,
      "ball_speed": 300
    }
  }
}
```

Este ejemplo muestra cómo representar el estado completo de una partida de Pong. JSON también se usa en ficheros de configuración, almacenamiento de datos en disco, y bases de datos NoSQL como MongoDB.

## 10.5. Funcionamiento de un Servicio REST

El enfoque más habitual en los servicios REST establece cuatro principios fundamentales. En REST, todo es un recurso identificado por una URI única. La URI está compuesta por una parte fija (dominio y ruta base) y una parte variable que identifica el recurso específico. Por ejemplo: `http://api.game.com/players/alice`, `http://api.game.com/matches/12345`, `http://api.game.com/players/alice/scores`, o `http://api.game.com/leaderboard`.

Los principios de diseño de URIs incluyen usar sustantivos, no verbos (`/players/alice` ✓ vs `/getPlayer?id=alice` ✗); usar plurales para colecciones (`/players` para colección, `/players/alice` para uno específico); crear jerarquías lógicas (`/players/alice/matches` refleja relación entre recursos); y usar minúsculas y guiones (`/game-sessions/active`).

REST aprovecha los métodos HTTP para indicar qué operación realizar sobre un recurso. **GET** obtiene información, es seguro (no modifica el servidor), idempotente (múltiples peticiones producen el mismo resultado) y cacheable. **POST** crea nuevos recursos, donde el servidor decide el ID y lo devuelve en la respuesta; no es seguro ni idempotente. **PUT** actualiza un recurso existente enviando el recurso completo; no es seguro pero sí idempotente. **DELETE** elimina un recurso; no es seguro pero sí idempotente (eliminar varias veces produce el mismo resultado: el recurso no existe).

La información se intercambia en formato JSON. Una petición HTTP REST completa tiene la siguiente anatomía:

```
POST /matches HTTP/1.1
Host: api.game.com
Content-Type: application/json
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Length: 89

{
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "player_id": "alice"
}
```

Esta petición tiene tres partes: la línea de petición ( `POST /matches HTTP/1.1` ) indica el método (POST), la ruta (/matches) y la versión HTTP; los headers (cabeceras) son metadatos sobre la petición donde `Content-Type: application/json` indica que enviamos JSON, `Accept: application/json` indica que queremos recibir JSON, `Authorization: Bearer ...` incluye el token de autenticación, y `Content-Length: 89` indica el tamaño del cuerpo en bytes; y el body (cuerpo) contiene los datos JSON que enviamos al servidor.

Una respuesta HTTP REST completa tiene la siguiente estructura:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: http://api.game.com/matches/67890
Content-Length: 156

{
  "id": "67890",
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "status": "waiting",
  "created_at": "2025-01-15T14:30:00Z",
  "players": ["alice"]
}
```

Esta respuesta tiene tres partes: la línea de estado (`HTTP/1.1 201 Created`) indica la versión HTTP, el código de estado (201) y el mensaje (Created); los headers (cabeceras) son metadatos sobre la respuesta donde `Content-Type: application/json` indica que la respuesta es JSON, `Location: http://...` proporciona la URI del recurso creado, y `Content-Length: 156` indica el tamaño de la respuesta; y el body (cuerpo) contiene los datos JSON del recurso creado.

Los códigos de estado HTTP son números de tres dígitos que comunican el resultado de la operación:

**1xx - Respuestas informativas:** Indican que la petición fue recibida y el proceso continúa. Raramente usados en REST. El más relevante es 101 Switching Protocols, usado para cambiar de HTTP a WebSockets.

**2xx - Respuestas exitosas:** La petición fue procesada correctamente. Los más comunes son 200 OK (éxito general en GET/PUT), 201 Created (recurso creado tras POST, debe incluir header Location), y 204 No Content (éxito sin contenido, típico en DELETE).

**3xx - Redirecciones:** El cliente debe tomar acciones adicionales. Incluyen 301 Moved Permanently (recurso movido permanentemente) y 304 Not Modified (usado con caché, el recurso no ha cambiado).

**4xx - Errores del cliente:** Error en la petición. Los más importantes son 400 Bad Request (petición mal formada), 401 Unauthorized (autenticación requerida), 403 Forbidden (sin permisos), 404 Not Found (recurso no existe), 409 Conflict (conflicto con estado actual, ej: usuario ya existe), y 422 Unprocessable Entity (formato correcto pero errores de validación).

**5xx - Errores del servidor:** El servidor no pudo completar la petición. Los principales son 500 Internal Server Error (error genérico), 503 Service Unavailable (servidor temporalmente no disponible, debe incluir Retry-After), y 504 Gateway Timeout (timeout esperando respuesta de otro servidor).

## 10.6. Ventajas de esta Arquitectura

La arquitectura REST ofrece varias ventajas clave. Proporciona separación de responsabilidades, donde la API REST maneja operaciones de gestión mientras que el sistema de comandos se encarga de la lógica del juego, permitiendo que cada componente evolucione independientemente. Ofrece escalabilidad, ya que al separar las operaciones de gestión de las de tiempo real, podemos escalar cada servicio según sus necesidades específicas. Mejora la testabilidad, permitiendo probar la API REST de forma independiente al juego, y el sistema de comandos puede probarse sin necesidad de conexión a red. Finalmente, proporciona flexibilidad, ya que la misma API REST puede servir a múltiples clientes: el juego web, una aplicación móvil, o herramientas de administración.

# 11. Cliente API REST con Javascript

---

La función `fetch()` es la API estándar moderna de JavaScript para realizar peticiones HTTP. Está integrada en todos los navegadores modernos y utiliza promesas (Promises) para manejar operaciones asincrónicas de manera elegante.

Cuando trabajamos con `fetch()`, tenemos dos formas principales de manejar las respuestas asincrónicas. La primera opción es usar callbacks con `.then()`, donde encadenamos promesas para procesar la respuesta. Primero verificamos si la petición fue exitosa con `response.ok`, luego convertimos la respuesta a JSON, y finalmente procesamos los datos. Si algo falla en cualquier punto de la cadena, el `.catch()` captura el error.

La segunda opción, más recomendada por su legibilidad, es usar `async/await`. Aquí declaramos una función asíncrona y usamos `await` para esperar las promesas. Esto hace que el código se lea de forma más secuencial y natural, similar a código síncrono, aunque sigue siendo asíncrono. Siempre envolvemos el código en un bloque `try/catch` para manejar posibles errores.

```
// Con callbacks (.then)
fetch('https://api.game.com/players/alice')
  .then(response => {
    if (!response.ok) throw new Error('HTTP error ' + response.status);
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));

// Con async/await (recomendado)
async function getPlayer() {
  try {
    const response = await fetch('https://api.game.com/players/alice');
    if (!response.ok) throw new Error('HTTP error ' + response.status);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

## 11.1. GET - Obtener datos

Para obtener datos de una API, usamos el método GET, que es el predeterminado de `fetch()`. En el primer ejemplo obtenemos información de un jugador y accedemos directamente a una propiedad específica del objeto returned. En el segundo ejemplo creamos una función reutilizable que acepta un ID dinámico mediante template literals, lo que nos permite obtener diferentes jugadores cambiando solo el parámetro.

```
// Con callbacks
fetch('https://api.game.com/players/alice')
  .then(response => response.json())
  .then(player => console.log(player.name))
  .catch(error => console.error(error));

// Con async/await
async function getPlayer(id) {
  const response = await fetch(`https://api.game.com/players/${id}`);
  return await response.json();
}
```

## 11.2. POST - Crear recurso

Cuando necesitamos crear un nuevo recurso, usamos el método POST. Aquí debemos especificar tres elementos clave en el segundo parámetro de `fetch()`: el método HTTP, las cabeceras indicando que enviamos JSON, y el cuerpo de la petición con los datos convertidos a string JSON mediante `JSON.stringify()`. El servidor procesará estos datos, creará el nuevo jugador, y típicamente nos devolverá el objeto creado con su nuevo ID asignado.

```

// Con callbacks
fetch('https://api.game.com/players', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ username: 'charlie', email:
    'charlie@example.com' })
})
  .then(response => response.json())
  .then(newPlayer => console.log('Creado:', newPlayer.id))
  .catch(error => console.error(error));

// Con async/await
async function createPlayer(data) {
  const response = await fetch('https://api.game.com/players', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
  });
  return await response.json();
}

```

### 11.3. PUT - Actualizar recurso

Para actualizar un recurso existente, usamos PUT, que reemplaza completamente el recurso en el servidor. La estructura es similar al POST, pero aquí incluimos el ID del recurso en la URL para indicar cuál queremos actualizar. Enviamos los nuevos datos que queremos aplicar, y el servidor actualiza el recurso y nos devuelve la versión actualizada. Es importante notar que PUT típicamente reemplaza todo el recurso, a diferencia de PATCH que solo actualizaría los campos especificados.

```

// Con callbacks
fetch('https://api.game.com/players/charlie', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ email: 'newemail@example.com', level: 5 })
})
  .then(response => response.json())
  .then(updated => console.log('Actualizado'))
  .catch(error => console.error(error));

// Con async/await
async function updatePlayer(id, updates) {
  const response = await fetch(`https://api.game.com/players/${id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(updates)
  });
  return await response.json();
}

```

## 11.4. DELETE - Eliminar recurso

El método DELETE es el más simple de todos, ya que solo necesitamos especificar el método y la URL con el ID del recurso a eliminar. No necesitamos enviar un cuerpo ni cabeceras especiales. El servidor eliminará el recurso y típicamente responde con un código 204 (sin contenido) o 200 (éxito). Verificamos que la operación fue exitosa usando `response.ok`, que será `true` para códigos de estado 2xx.

```

// Con callbacks
fetch('https://api.game.com/players/charlie', {
  method: 'DELETE'
})
  .then(response => {
    if (response.ok) console.log('Eliminado');
  })
  .catch(error => console.error(error));

// Con async/await
async function deletePlayer(id) {
  const response = await fetch(`https://api.game.com/players/${id}`, {
    method: 'DELETE'
  });
  return response.ok;
}

```

## 11.5. Manejo de errores y códigos de estado

En aplicaciones reales, necesitamos manejar diferentes situaciones que pueden ocurrir durante las peticiones HTTP. Este ejemplo muestra cómo implementar un manejo robusto de errores usando un `switch` sobre el código de estado. Cada código HTTP tiene un significado específico: 200 indica éxito y devolvemos los datos, 401 significa que el usuario necesita autenticarse, 403 indica que aunque esté autenticado no tiene permisos para ese recurso, 404 significa que el recurso no existe, y 500 indica un error interno del servidor. Capturando estos casos específicos, podemos proporcionar mensajes de error claros y tomar acciones apropiadas para cada situación, mejorando significativamente la experiencia del usuario.

```
async function fetchWithErrorHandling(url) {
  try {
    const response = await fetch(url);

    switch (response.status) {
      case 200:
        return await response.json();
      case 401:
        throw new Error('No autenticado');
      case 403:
        throw new Error('Sin permisos');
      case 404:
        throw new Error('Recurso no encontrado');
      case 500:
        throw new Error('Error del servidor');
      default:
        throw new Error(`Error: ${response.status}`);
    }
  } catch (error) {
    console.error('Error en petición:', error);
    throw error;
  }
}
```

## 12. Servidor API REST con Javascript

---

Express.js es un framework web minimalista y flexible para Node.js que se ha convertido en el estándar para crear APIs REST. Proporciona un sistema de enrutamiento robusto, soporte para middlewares que procesan peticiones, manejo de errores integrado y compatibilidad completa con todos los métodos HTTP y códigos de estado. Su popularidad se debe a su simplicidad y a que permite construir servidores HTTP de manera rápida y eficiente.

Para comenzar a trabajar con Express, primero necesitamos inicializar un proyecto Node.js y luego instalar el framework. El comando `npm init -y` crea automáticamente un archivo `package.json` con la configuración por defecto, y posteriormente instalamos Express como dependencia del proyecto:

```
npm init -y  
npm install express
```

Para usar la sintaxis de módulos ES6 (`import/export`) en lugar de CommonJS (`require/module.exports`), debemos añadir `"type": "module"` en el `package.json`:

```
{  
  "name": "mi-api-juego",  
  "version": "1.0.0",  
  "type": "module",  
  "dependencies": {  
    "express": "^4.18.0"  
  }  
}
```

Con esta configuración, podremos usar `import` y `export` en nuestro código.

Un proyecto Express típico organiza el código en varias carpetas. La carpeta `node_modules` contiene todas las dependencias instaladas y no debe subirse al repositorio. El código fuente se ubica en `src`, donde creamos subcarpetas para los controladores y las rutas. El archivo `package.json` define las dependencias y scripts del proyecto, mientras que `package-lock.json` asegura que las versiones de las dependencias sean consistentes.

```
mi-api-juego/
└── node_modules/
└── src/
    ├── controllers/
    ├── routes/
    └── app.js
└── package.json
└── package-lock.json
```

Para crear un servidor básico con Express, importamos el módulo y creamos una instancia de la aplicación. El middleware `express.json()` es fundamental porque permite que Express pueda leer y parsear automáticamente el cuerpo de las peticiones que vienen en formato JSON. Sin este middleware, no podríamos acceder a los datos que el cliente envía en las peticiones POST o PUT. Finalmente, el método `listen()` inicia el servidor en el puerto especificado:

```
import express from 'express';

const app = express();

app.use(express.json());

app.listen(8080, () => {
  console.log('Servidor ejecutándose en http://localhost:8080');
});
```

Para ejecutar el servidor simplemente usamos `node src/app.js` desde la terminal.

## 12.1. Controladores

Los controladores son las funciones que contienen la lógica de negocio de cada endpoint. Son las responsables de procesar las peticiones HTTP, manipular los datos recibidos, realizar operaciones sobre ellos y devolver las respuestas apropiadas al cliente. Mantener esta lógica separada en controladores hace que el código sea más organizado, reutilizable y fácil de mantener.

Una forma elegante de mantener estado privado en los controladores es mediante closures. Creamos una función factory que inicializa el estado y devuelve las funciones del controlador. Estas funciones tienen acceso al estado compartido pero éste no es accesible desde fuera:

```

// src/controllers/anunciosController.js
const createAnunciosController = () => {
  // Estado privado compartido
  const anuncios = [];
  let nextId = 1;

  const getAll = (req, res) => {
    res.json(anuncios);
  };

  const create = (req, res) => {
    const { nombre, asunto, comentario } = req.body;

    const nuevoAnuncio = {
      id: nextId++,
      nombre,
      asunto,
      comentario
    };

    anuncios.push(nuevoAnuncio);
    res.status(201).json(nuevoAnuncio);
  };

  return { getAll, create };
};

export default createAnunciosController;

```

Este patrón permite que múltiples funciones del controlador comparten el mismo estado (`anuncios` y `nextId`) sin exponerlo globalmente. El estado está encapsulado dentro del closure y solo las funciones retornadas pueden acceder a él.

## 12.2. Rutas

Las rutas son la capa que conecta las URLs de nuestra API con los controladores correspondientes. Definen qué controlador se ejecutará cuando llegue una petición a una URL específica con un método HTTP determinado. Express proporciona un objeto Router que permite agrupar rutas relacionadas y aplicarles prefijos comunes.

Para definir rutas, creamos un archivo que importa el Router de Express. Como nuestro controlador es una función factory, la invocamos para obtener las funciones del controlador y luego las asociamos con las rutas:

```
// src/routes/anunciosRoutes.js
import express from 'express';
import createAnunciosController from '../controllers/
    anunciosController.js';

const router = express.Router();
const controller = createAnunciosController();

router.get('/', controller.getAll);
router.post('/', controller.create);

export default router;
```

Para integrar estas rutas en nuestra aplicación, las importamos y las registramos con un prefijo usando `app.use()`. Esto significa que todas las rutas definidas en el router tendrán ese prefijo automáticamente:

```
// src/app.js
import express from 'express';
import anunciosRoutes from './routes/anunciosRoutes.js';

const app = express();
app.use(express.json());

app.use('/anuncios', anunciosRoutes);

app.listen(8080);
```

Con esta configuración, la ruta `router.get('/')` se convierte en `GET /anuncios` y la ruta `router.post('/')` se convierte en `POST /anuncios`.

## 12.3. Operaciones CRUD

La operación GET se utiliza para recuperar información del servidor sin modificarla. Express proporciona el objeto `req.params` para acceder a los parámetros de la URL. Por ejemplo, en la ruta `/anuncios/:id`, el `:id` es un parámetro dinámico cuyo valor estará disponible en `req.params.id`. El método `res.json()` convierte automáticamente un objeto JavaScript a formato JSON y lo envía como respuesta con las cabeceras correctas:

```
app.get('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const anuncio = anuncios.find(a => a.id === parseInt(id));

  if (!anuncio) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  res.json(anuncio);
});
```

La operación POST crea nuevos recursos en el servidor. Los datos que el cliente envía vienen en el cuerpo de la petición y están disponibles en `req.body` gracias al middleware `express.json()`. Es importante validar siempre los datos recibidos antes de procesarlos para evitar errores o datos incorrectos. Cuando se crea un recurso exitosamente, el código de estado apropiado es 201 (Created):

```
app.post('/anuncios', (req, res) => {
  const { nombre, asunto, comentario } = req.body;

  if (!nombre || !asunto) {
    return res.status(400).json({
      error: 'Nombre y asunto son requeridos'
    });
  }

  const nuevoAnuncio = {
    id: Date.now(),
    nombre,
    asunto,
    comentario
  };

  anuncios.push(nuevoAnuncio);
  res.status(201).json(nuevoAnuncio);
});
```

La operación PUT actualiza un recurso existente reemplazándolo completamente con los nuevos datos. Para actualizar necesitamos tanto el identificador del recurso (que viene en la URL) como los nuevos datos (que vienen en el body). Si el recurso no existe, devolvemos un error 404:

```

app.put('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const { nombre, asunto, comentario } = req.body;

  const anuncio = anuncios.find(a => a.id === parseInt(id));

  if (!anuncio) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  anuncio.nombre = nombre;
  anuncio.asunto = asunto;
  anuncio.comentario = comentario;

  res.json(anuncio);
});

```

La operación DELETE elimina un recurso del servidor. Podemos opcionalmente devolver el recurso eliminado en la respuesta, o simplemente devolver un código 204 (No Content) que indica éxito sin contenido. El método `findIndex()` nos permite localizar la posición del elemento en el array para poder eliminarlo con `splice()`:

```

app.delete('/anuncios/:id', (req, res) => {
  const { id } = req.params;
  const index = anuncios.findIndex(a => a.id === parseInt(id));

  if (index === -1) {
    return res.status(404).json({ error: 'No encontrado' });
  }

  const anuncioEliminado = anuncios.splice(index, 1)[0];
  res.json(anuncioEliminado);
});

```

## 12.4. Middlewares en Express

Los middlewares son funciones que se ejecutan durante el ciclo de vida de una petición, antes de que llegue al controlador final. Tienen acceso a los objetos de petición `req` y respuesta `res`, y a la función `next()` que permite pasar el control al siguiente middleware o ruta. Los middlewares son útiles para tareas transversales como logging, autenticación, validación o manejo de errores.

Un middleware de logging básico registra cada petición que llega al servidor. Es importante llamar a `next()` al final para que la petición continúe su flujo normal hacia el controlador apropiado:

```
app.use((req, res, next) => {
  console.log(`${req.method} ${req.path}`);
  next();
});
```

Los middlewares de manejo de errores son especiales porque tienen cuatro parámetros en lugar de tres. Express los identifica automáticamente por esta firma y los invoca cuando ocurre un error. Deben colocarse después de todas las rutas para capturar cualquier error que no haya sido manejado:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Error interno del servidor' });
});
```

## 12.5. Servir archivos estáticos

Además de servir una API REST, Express puede servir archivos estáticos como HTML, CSS, JavaScript del cliente e imágenes. El middleware `express.static()` configura un directorio desde el cual se servirán estos archivos automáticamente. Cuando el cliente solicita un archivo, Express lo busca en ese directorio y lo envía si existe:

```
app.use(express.static('public'));
```

Con esta configuración, si creamos la siguiente estructura de carpetas, los archivos serán accesibles directamente desde la raíz del servidor:

```
mi-api-juego/
├── public/
│   ├── index.html
│   ├── script.js
│   └── style.css
└── src/
    └── app.js
```

El archivo `public/index.html` estará disponible en `http://localhost:8080/index.html` y `public/script.js` en `http://localhost:8080/script.js`

## Comunicación asíncrona cliente - servidor

---

Mientras que las APIs REST son excelentes para operaciones síncronas y peticiones puntuales, los juegos multijugador en tiempo real requieren una comunicación bidireccional constante y de baja latencia. Aquí es donde entran en juego los WebSockets, una tecnología que permite establecer conexiones persistentes entre el cliente y el servidor para intercambiar mensajes en tiempo real. En esta parte final del curso dominaremos WebSockets y Socket.IO para implementar comunicación bidireccional, gestionar conexiones de múltiples jugadores, trabajar con salas y broadcast, y sincronizar el estado del juego entre todos los clientes conectados. Aprenderemos las diferencias fundamentales entre HTTP y WebSockets, cuándo utilizar cada tecnología, y cómo implementar sistemas de comunicación en tiempo real que permitan crear experiencias multijugador fluidas y responsivas, culminando así con un dominio completo del stack tecnológico para juegos en red.

# 13. Introducción a WebSockets

---

## 13.1. Introducción a los Sockets

Antes de adentrarnos en WebSockets, es fundamental comprender el concepto básico de sockets en las comunicaciones de red. En el contexto de las comunicaciones fiables, cuando conocemos la dirección IP y el puerto de dos aplicaciones que se ejecutan en máquinas distintas, los **sockets** permiten establecer una comunicación bidireccional entre ellas.

Un **socket** se identifica mediante una dirección IP y un puerto. Podemos pensar en él como uno de los extremos de un canal de comunicación bidireccional entre dos programas. Cuando establecemos una conexión, tanto el cliente como el servidor tienen un socket asociado a través del cual pueden enviar y recibir información simultáneamente.

Imaginemos un escenario típico donde un servidor tiene la IP `212.128.240.50` y está escuchando en el puerto `10000`, mientras que un cliente con IP `1XX.XXX.XXX.XX` usa un puerto dinámico (por ejemplo, `YYYY`). El proceso de comunicación funciona así: primero, el servidor se encuentra escuchando en su puerto específico; luego, el cliente inicia una petición de conexión hacia el servidor; una vez aceptada, se establece una conexión bidireccional donde ambas partes pueden enviar y recibir datos simultáneamente.

Esta comunicación bidireccional es clave: no se trata solo de que el cliente solicite y el servidor responda, sino que ambos pueden iniciar el envío de información en cualquier momento una vez establecida la conexión. Un socket es cada uno de los extremos de esta comunicación bidireccional entre dos programas.

## 13.2. ¿Qué son los WebSockets?

**WebSockets** es un protocolo de comunicaciones que permite abrir una sesión interactiva entre un cliente (normalmente un navegador web) y un servidor. La principal innovación de WebSockets es que permite establecer un canal **full-duplex**, lo que significa que tanto el cliente como el servidor pueden enviar mensajes de forma independiente sin necesidad de abrir una nueva conexión cada vez que se quiere comunicar. La comunicación es bidireccional y simultánea.

El funcionamiento de WebSockets sigue estos pasos: inicialmente, el cliente solicita la conexión al servidor; la conexión es aceptada por el servidor; se genera una conexión permanente que permanece abierta; y deberá haber procesos en ambos lados que escuchen para recibir mensajes. Esta persistencia de la conexión es fundamental para aplicaciones en tiempo real como juegos multijugador, donde necesitamos que el servidor pueda enviar actualizaciones al cliente sin que este tenga que estar constantemente preguntando “¿hay algo nuevo?”.

### 13.3. WebSockets como Estándar

WebSockets está definido como un estándar de comunicación en el RFC 6455 que funciona sobre un único socket TCP. Una de las grandes ventajas de WebSockets es que utiliza el puerto 80 (el mismo que HTTP) de forma multiplexada, lo que significa que puede soportar varias comunicaciones por el mismo puerto, evita problemas con firewalls y proxies que bloquean puertos no estándar, y está pensado principalmente para navegadores y servidores web.

Aunque WebSockets comienza con HTTP, no está limitado a este protocolo. Un cliente puede abrir un socket con el servidor y comunicarse con él a través de HTTP (el protocolo web tradicional) o un protocolo propio diseñado específicamente para la aplicación. Esta flexibilidad es especialmente útil en juegos en red, donde podemos diseñar un protocolo de comunicación optimizado para nuestras necesidades específicas.

### 13.4. Negociación de Conexión WebSocket

Aunque los desarrolladores raramente necesitan implementar manualmente este proceso, entender cómo se negocia una conexión WebSocket nos ayuda a comprender mejor su funcionamiento. La negociación para obtener un WebSocket se inicia sobre HTTP. El cliente envía una petición especial:

```
GET /demo HTTP/1.1
Host: example.com
Connection: Upgrade
Upgrade: WebSocket
Origin: http://example.com
```

Esta petición comienza como una petición HTTP normal ( `GET /demo HTTP/1.1` ), pero el header `Connection: Upgrade` indica que queremos cambiar de protocolo, el header `Upgrade: WebSocket` especifica que queremos usar WebSocket, y el header `Origin` indica desde dónde se origina la petición (importante para seguridad). En esencia, le estamos diciendo al servidor: "Hola, quiero establecer una conexión WebSocket contigo en la ruta `/demo`".

Si el servidor acepta la conexión WebSocket, responde con:

```
HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Location: ws://example.com/demo
```

El código 101 indica que se acepta el cambio de protocolo, `Upgrade: WebSocket` confirma que vamos a usar WebSocket, `Connection: Upgrade` confirma el cambio de protocolo, `Sec-WebSocket-Origin` confirma el origen de la conexión, y `Sec-WebSocket-Location` indica la ubicación del WebSocket con el nuevo protocolo.

Fíjate en la última línea: `ws://example.com/demo`. WebSocket introduce dos nuevos esquemas de protocolo: `ws://` para WebSocket en claro (equivalente a `http://`) y `wss://` para WebSocket seguro sobre TLS (equivalente a `https://`). Una vez completado este "handshake" (apretón de manos), la conexión HTTP se transforma en una conexión WebSocket persistente, y ambas partes pueden comenzar a intercambiar mensajes libremente.

## 13.5. Uso Normal de WebSockets

En la práctica, normalmente no tenemos que preocuparnos por la negociación HTTP que acabamos de ver. Las conexiones se realizan automáticamente utilizando los protocolos `ws://` para WebSocket normal y `wss://` para WebSocket seguro. Cuando creamos una conexión WebSocket desde JavaScript (como veremos en el siguiente tema), simplemente usamos:

```
new WebSocket('ws://example.com/demo');
```

Y todo el proceso de negociación que hemos explicado ocurre automáticamente entre bastidores. El navegador y el servidor se encargan de todo el intercambio HTTP inicial y del cambio al protocolo WebSocket.

Para concluir este tema, es importante destacar por qué WebSockets es especialmente útil en el contexto de juegos en red: proporciona comunicación en tiempo real donde los jugadores reciben actualizaciones instantáneas sin demoras; es bidireccional, permitiendo que el servidor envíe eventos a los clientes sin que estos pregunten constantemente; es eficiente, ya que una sola conexión persistente consume menos recursos que múltiples peticiones HTTP; ofrece baja latencia, ideal para juegos que requieren respuestas rápidas; y es un estándar compatible que funciona en todos los navegadores modernos sin necesidad de plugins.

# 14. Cliente WebSockets con JavaScript

---

## 14.1. La API WebSocket en JavaScript

JavaScript proporciona una API nativa para crear y administrar conexiones WebSocket desde el navegador. Esta API nos permite establecer conexiones con servidores WebSocket de forma sencilla y manejar toda la comunicación bidireccional. La gran ventaja de esta API es que está integrada directamente en el navegador, por lo que no necesitamos instalar librerías adicionales para usar WebSockets en nuestras aplicaciones web.

La API está implementada en el objeto **WebSocket**. Para crear una conexión, simplemente instanciamos este objeto pasándole como parámetro la URL del servidor al que queremos conectarnos:

```
WebSocket('ws://IP:PUERTO/RUTA');
```

Donde **ws://** es el protocolo WebSocket (o **wss://** para conexiones seguras), **IP** es la dirección IP o dominio del servidor, **PUERTO** es el puerto donde escucha el servidor WebSocket, y **RUTA** es el path específico del endpoint WebSocket en el servidor.

Imagina que tenemos un servidor WebSocket ejecutándose localmente en el puerto 8080, con un endpoint llamado **/echo**:

```
var connection = new WebSocket('ws://127.0.0.1:8080/echo');
```

Con esta única línea hemos creado la conexión. El navegador automáticamente inicia el handshake HTTP que vimos en el tema anterior, negocia el cambio al protocolo WebSocket, y establece la conexión persistente. Ahora tenemos un objeto **connection** que representa nuestro extremo de la comunicación bidireccional.

## 14.2. Métodos del Objeto WebSocket

El objeto WebSocket tiene dos métodos principales que utilizaremos para interactuar con el servidor. El método **send()** transmite datos al servidor a través de la conexión WebSocket:

```
connection.send('Hi');
```

Este método es muy simple: le pasamos los datos que queremos enviar y el objeto WebSocket se encarga de transmitirlos al servidor. Podemos enviar texto simple como 'Hola servidor', o datos más complejos (típicamente JSON convertido a string):

```
var data = {
  action: 'move',
  x: 100,
  y: 250
};
connection.send(JSON.stringify(data));
```

En juegos en red, es común enviar objetos JSON que describen las acciones del jugador. Por ejemplo, si el jugador se mueve, podríamos enviar:

```
connection.send(JSON.stringify({
  type: 'player_move',
  x: 150,
  y: 200
}));
```

El método **close()** cierra la conexión WebSocket de forma ordenada:

```
connection.close();
```

Cuando llamamos a este método, se inicia el proceso de cierre de la conexión, se envía una señal al servidor indicando que vamos a cerrar, y se libera la conexión. Esto es útil cuando el jugador sale del juego o cuando queremos limpiar recursos.

## 14.3. Event Listeners - La Clave de WebSockets

Aquí viene la parte más importante: los **event listeners**. Un objeto WebSocket tiene diferentes atributos que actúan como listeners, es decir, funciones que se ejecutan automáticamente cuando ocurren ciertos eventos. Estos listeners son la forma en que nuestro código "escucha" lo que está pasando con la conexión WebSocket.

El listener **onopen** es llamado cuando la conexión del WebSocket cambia a un estado abierto (OPEN):

```
connection.onopen = function () {
  connection.send('Hi');
}
```

Este evento se dispara justo después de que la conexión se establece exitosamente. Es el momento perfecto para enviar un mensaje inicial al servidor, notificar al usuario que está conectado, o inicializar el estado del juego. En el ejemplo anterior, tan pronto como se abre la conexión, enviamos el mensaje 'Hi' al servidor. Podríamos hacer algo más elaborado:

```
connection.onopen = function() {
  console.log('Conectado al servidor');

  // Enviar información inicial del jugador
  connection.send(JSON.stringify({
    type: 'player_join',
    username: 'Jugador1'
  }));
};


```

El listener **onerror** es llamado cuando se produce un error en la conexión:

```
connection.onerror = function(e) {
  console.log("WS error: " + e);
}
```

Este evento se dispara cuando algo sale mal: no se puede establecer la conexión con el servidor, hay un problema de red, o el servidor rechaza la conexión. Es importante manejar estos errores para informar al usuario de que algo no funciona correctamente:

```
connection.onerror = function(error) {
  console.log("WS error: " + error);
  alert('No se pudo conectar al servidor. Por favor, verifica tu conexión.');
};
```

El listener **onmessage** es llamado cuando se recibe un mensaje del servidor. Este es probablemente el listener más importante, ya que aquí es donde procesamos toda la información que nos envía el servidor:

```
connection.onmessage = function(msg) {
  console.log("WS message: " + msg.data);
}
```

El parámetro `msg` es un objeto evento que contiene la información del mensaje. El dato real que envió el servidor está en `msg.data`. Veamos un ejemplo más completo:

```
connection.onmessage = function(msg) {
  console.log("Mensaje recibido: " + msg.data);

  // Si el servidor envía JSON, lo parseamos
  var data = JSON.parse(msg.data);

  // Procesamos según el tipo de mensaje
  if (data.type === 'player_position') {
    actualizarPosicionJugador(data.playerId, data.x, data.y);
  } else if (data.type === 'game_over') {
    mostrarPantallaFinJuego(data.winner);
  }
};
```

En un juego multijugador, este listener recibiría constantemente actualizaciones del servidor sobre posiciones de otros jugadores, estado del juego, y eventos importantes (un jugador ganó, apareció un enemigo, etc.).

El listener `onclose` es llamado cuando la conexión del WebSocket cambia a un estado cerrado (CLOSED):

```
connection.onclose = function(event) {
  console.log('Conexión cerrada');
};
```

Este evento se dispara cuando llamamos a `connection.close()` nosotros mismos, el servidor cierra la conexión, o se pierde la conexión de red. Es un buen lugar para limpiar recursos o intentar reconectar:

```
connection.onclose = function(event) {
    console.log('Desconectado del servidor');
    console.log('Código de cierre:', event.code);

    // Notificar al usuario
    alert('Se ha perdido la conexión con el servidor');

    // Intentar reconnectar después de 3 segundos
    setTimeout(function() {
        connection = new WebSocket('ws://127.0.0.1:8080/echo');
        configurarListeners(connection);
    }, 3000);
};
```

# 15. Servidor WebSockets con JavaScript

---

## 15.1. Servidor WebSocket con Node.js

Para implementar un servidor WebSocket en Node.js, utilizaremos la librería `ws`, que es una implementación robusta y eficiente del protocolo WebSocket. Esta librería nos permite crear servidores WebSocket que pueden manejar múltiples conexiones simultáneas de clientes.

Primero necesitamos inicializar un proyecto Node.js e instalar las dependencias necesarias:

```
npm init -y  
npm install ws express
```

Instalamos tanto `ws` para el servidor WebSocket como `express` para servir archivos estáticos (la interfaz del cliente). Para usar la sintaxis moderna de módulos ES6, debemos configurar el `package.json`:

```
{  
  "name": "websocket-server",  
  "version": "1.0.0",  
  "type": "module",  
  "dependencies": {  
    "ws": "^8.0.0",  
    "express": "^4.18.0"  
  }  
}
```

La propiedad `"type": "module"` nos permite usar `import` y `export` en lugar de `require` y `module.exports`.

Un servidor WebSocket completo que también sirve archivos estáticos tiene la siguiente estructura:

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

// Configurar Express para servir archivos estáticos
app.use(express.static('public'));

// Crear el servidor WebSocket
const wss = new WebSocketServer({ server });

// Iniciar el servidor HTTP
const PORT = 8080;
server.listen(PORT, () => {
  console.log(`Servidor ejecutándose en http://localhost:${PORT}`);
});
```

En este código estamos creando un servidor HTTP con `createServer(app)` que servirá tanto las peticiones HTTP normales (archivos estáticos) como las conexiones WebSocket; configurando Express para servir archivos estáticos desde la carpeta `public`, donde estará nuestra interfaz HTML/CSS/JS del cliente; y creando el servidor WebSocket asociándolo al mismo servidor HTTP, lo que permite que ambos protocolos comparten el mismo puerto.

El servidor WebSocket funciona mediante eventos. El evento más importante es `connection`, que se dispara cada vez que un cliente establece una conexión:

```
wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  // Aquí manejamos los eventos de este cliente específico
});
```

El parámetro `ws` representa la conexión individual con ese cliente. Cada cliente que se conecta tiene su propio objeto `ws`, lo que nos permite comunicarnos de forma independiente con cada uno.

Para recibir mensajes que envían los clientes, configuraremos un listener para el evento `message` en cada conexión:

```
wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  ws.on('message', (message) => {
    console.log('Mensaje recibido:', message.toString());

    // Procesar el mensaje aquí
    const data = message.toString();
  });
});
```

Los mensajes llegan como objetos `Buffer`, por lo que usamos `.toString()` para convertirlos a strings. Si esperamos recibir JSON, podemos parsearlo:

```
ws.on('message', (message) => {
  const data = JSON.parse(message.toString());
  console.log('Nombre:', data.nombre);
  console.log('Mensaje:', data.mensaje);
});
```

Para enviar un mensaje a un cliente específico, usamos el método `send()` del objeto de conexión:

```
ws.on('message', (message) => {
  const received = message.toString();

  // Enviar respuesta de vuelta al mismo cliente
  ws.send(`Echo: ${received}`);
});
```

Si queremos enviar objetos JavaScript (como JSON), debemos convertirlos a string primero:

```
ws.on('message', (message) => {
  const respuesta = {
    tipo: 'confirmacion',
    timestamp: new Date().toISOString(),
    mensaje: 'Mensaje recibido correctamente'
  };

  ws.send(JSON.stringify(respuesta));
});
```

Es fundamental limpiar recursos cuando un cliente se desconecta. Para esto escuchamos el evento `close`:

```
wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  ws.on('close', () => {
    console.log('Cliente desconectado');
    // Aquí podríamos limpiar cualquier dato asociado a este cliente
  });

  ws.on('message', (message) => {
    // ... manejar mensajes
  });
});
```

También podemos manejar errores de conexión con el evento `error`:

```
ws.on('error', (error) => {
  console.error('Error en la conexión:', error);
});
```

## 15.2. Ejemplo completo: Servidor Echo

Un servidor echo completo que devuelve cualquier mensaje que recibe se vería así:

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

app.use(express.static('public'));

const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  ws.on('message', (message) => {
    const data = message.toString();
    console.log('Mensaje recibido:', data);

    // Enviar el mensaje de vuelta al cliente
    ws.send(`Echo: ${data}`);
  });

  ws.on('close', () => {
    console.log('Cliente desconectado');
  });

  ws.on('error', (error) => {
    console.error('Error:', error);
  });
});

const PORT = 8080;
server.listen(PORT, () => {
  console.log(`Servidor ejecutándose en http://localhost:${PORT}`);
});
```

### 15.3. Ejemplo completo: Servidor de Chat con JSON

Un servidor de chat más sofisticado que maneja mensajes en formato JSON:

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

app.use(express.static('public'));

const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
    console.log('Nuevo cliente conectado al chat');

    ws.on('message', (message) => {
        try {
            // Parsear el mensaje JSON del cliente
            const datos = JSON.parse(message.toString());
            console.log(` ${datos.nombre}: ${datos.mensaje}`);

            // Crear respuesta en formato JSON
            const respuesta = {
                nombre: datos.nombre,
                mensaje: datos.mensaje,
                timestamp: new Date().toISOString()
            };

            // Broadcast: enviar a todos los clientes
            wss.clients.forEach((client) => {
                if (client.readyState === ws.OPEN) {
                    client.send(JSON.stringify(respuesta));
                }
            });
        } catch (error) {
            console.error('Error al parsear mensaje:', error);
            ws.send(JSON.stringify({
                error: 'Formato de mensaje inválido'
            }));
        }
    });

    ws.on('close', () => {
        console.log('Cliente desconectado del chat');
    });

    ws.on('error', (error) => {
        console.error('Error en la conexión:', error);
    });
});
```

```

const PORT = 8080;
server.listen(PORT, () => {
  console.log(`Servidor de chat ejecutándose en http://localhost:${PORT}`);
});

```

En este ejemplo de chat, recibimos mensajes JSON que contienen el nombre del usuario y su mensaje; validamos el formato usando un bloque try-catch para manejar errores de parsing; enriquecemos el mensaje añadiendo un timestamp del servidor; y hacemos broadcast enviando el mensaje a todos los clientes conectados en formato JSON.

A menudo necesitamos mantener información sobre los clientes conectados. Podemos usar estructuras de datos como `Map` para asociar datos adicionales a cada conexión:

```

const clientes = new Map();

wss.on('connection', (ws) => {
  // Asignar un ID único al cliente
  const clientId = Date.now();
  clientes.set(ws, { id: clientId, nombre: null });

  console.log(`Cliente ${clientId} conectado`);

  ws.on('message', (message) => {
    const datos = JSON.parse(message.toString());

    // Guardar el nombre del cliente
    const clienteInfo = clientes.get(ws);
    clienteInfo.nombre = datos.nombre;

    // Procesar el mensaje...
  });

  ws.on('close', () => {
    const clienteInfo = clientes.get(ws);
    console.log(`Cliente ${clienteInfo.id} desconectado`);
    clientes.delete(ws);
  });
});

```

## 15.4. Integración con Express

Podemos combinar rutas REST de Express con WebSockets en el mismo servidor:

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

app.use(express.json());
app.use(express.static('public'));

// Rutas REST
app.get('/api/status', (req, res) => {
  res.json({
    clientesConectados: wss.clients.size,
    estado: 'activo'
  });
});

// Servidor WebSocket
const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
  // ... manejar conexiones WebSocket
});

server.listen(8080, () => {
  console.log('Servidor ejecutándose en http://localhost:8080');
});
```

Esta arquitectura híbrida es muy útil porque HTTP/REST es ideal para operaciones puntuales como autenticación, consultas y subida de archivos, mientras que WebSocket es perfecto para comunicación en tiempo real como notificaciones, actualizaciones en vivo y chat. Ambos protocolos comparten el mismo servidor HTTP y puerto, simplificando el despliegue y la configuración del cliente.

# Referencias

---

- Aldridge, David. 2011. «I Shot You First: Networking the Gameplay of Halo: Reach». En Game Developers Conference.
- 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>.
- Bernier, Yahn W. 2001. «Latency compensating methods in client/server in-game protocol design and optimization». En Game Developers Conference.
- 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>.
- Doglio, Fernando. 2015. Pro REST API Development with Node.js. Berkeley, CA: Apress.
- Fette, Ian, y Alexey Melnikov. 2011. «The WebSocket Protocol». RFC 6455. Internet Engineering Task Force. <https://tools.ietf.org/html/rfc6455>.
- Fiedler, Glenn. 2024. «Gaffer on Games: Networking for Game Programmers». <https://gafferongames.com/>.
- Fielding, Roy Thomas. 2000. «Architectural Styles and the Design of Network-based Software Architectures». Tesis doctoral, University of California, Irvine.
- Flanagan, David. 2020. JavaScript: The Definitive Guide. 7.<sup>a</sup> ed. Sebastopol, CA: O'Reilly Media.
- Gambetta, Gabriel. 2014. «Fast-Paced Multiplayer». <https://www.gabrielgambetta.com/client-server-game-architecture.html>.
- Glazer, Joshua, y Sanjay Madhav. 2015. Multiplayer Game Programming: Architecting Networked Games. Boston, MA: Addison-Wesley Professional.
- Hahn, Evan. 2014. Express in Action: Writing, building, and testing Node.js applications. Shelter Island, NY: Manning Publications.
- Haverbeke, Marijn. 2018. Eloquent JavaScript: A Modern Introduction to Programming. 3.<sup>a</sup> ed. San Francisco, CA: No Starch Press.
- Kurose, James F., y Keith W. Ross. 2017. Computer Networks: A Top-Down Approach. 7.<sup>a</sup> ed. Boston, MA: Pearson.
- . 2021. Computer Networks: A Top-Down Approach. 8.<sup>a</sup> ed. Boston, MA: Pearson.

- Massé, Mark. 2011. REST API Design Rulebook. Sebastopol, CA: O'Reilly Media.
- Mozilla Developer Network. 2024. «JavaScript Guide». <https://developer.mozilla.org/>.
- Photonstorm. 2024. «Phaser 3 Documentation». <https://phaser.io/>.
- Richardson, Leonard, y Sam Ruby. 2013. RESTful Web APIs. Sebastopol, CA: O'Reilly Media.
- Ritchie, Hannah, Edouard Mathieu, Max Roser, y Esteban Ortiz-Ospina. 2023. «Internet». Our World in Data. <https://ourworldindata.org/internet>.
- Sturgeon, Phil. 2016. Build APIs You Won't Hate. LeanPub.
- Tanenbaum, Andrew S., y David J. Wetherall. 2021. Computer Networks. 6.<sup>a</sup> ed. Harlow, England: Pearson.
- Wang, Vanessa, Frank Salim, y Peter Moskovits. 2013. The Definitive Guide to HTML5 WebSocket. Berkeley, CA: Apress.
- Wilson, Jim. 2018. Node.js 8 the Right Way: Practical, Server-Side JavaScript That Scales. Raleigh, NC: Pragmatic Bookshelf.
- Young, Alex, Bradley Meck, y Mike Cantelon. 2017. Node.js in Action. 2.<sup>a</sup> ed. Shelter Island, NY: Manning Publications.

# 16. Diapositivas de la asignatura

## 16.1. Introducción a la asignatura

1

### Presentación de Juegos en Red

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño  
ruben.rodriguez@urjc.es natalia.madrueño@urjc.es  
URJC URJC

2025-09-09

 Universidad  
Rey Juan Carlos  DATA SCIENCE LABORATORY



2

### Tabla de contenidos

- [Información de la asignatura](#)
- [Motivación y objetivos](#)
- [Temario](#)
- [Metodología y Material Docente](#)
- [Evaluación](#)
- [Evaluación extraordinaria](#)
- [Calendario aproximado de entregas](#)
- [Bibliografía](#)



## Información de la asignatura



### Datos de la asignatura

- Tipo: Obligatoria.
- Período de impartición: Primer cuatrimestre.
- Nº de créditos: 6
  - Sesiones: 28 clases de 2h.
  - Horas de trabajo: 180h (56h clase / 124h fuera).
- Departamento: Departamento de Informática y Estadística.
- Web: Aula Virtual (<https://www.aulavirtual.urjc.es>).



## Prof. Rubén Rodríguez

- E-mail: [ruben.rodriguez@urjc.es](mailto:ruben.rodriguez@urjc.es) . (Escribir por aula virtual).
- Despacho: 131, Ed. Departamental II (Campus de Móstoles).
- Tutorías: Concertar por e-mail (aula virtual).



## Prof. Natalia Madrueño

- Email: [natalia.madrueno@urjc.es](mailto:natalia.madrueno@urjc.es) (Escribir por aula virtual).
- Despacho: 044, Ed. Departamental II (Campus de Móstoles).
- Tutorías: Concertar por e-mail (aula virtual).



## Motivación y objetivos



### Aplicaciones en red

La gran mayoría de las aplicaciones actuales son **aplicaciones basadas en red**:

- Aula Virtual
- Gmail
- Twitter
- Facebook
- La web de la Liga de Fútbol Profesional (LFP)
- ...



## Muchos juegos también

- PlayerUnknown's Battlegrounds (PUBG) / Fortnite / Destiny 2 / Overwatch / Quake Champions / Counter-Strike: Global Offensive (CS:GO) / League of Legends (LoL) / DOTA 2
- World of Warcraft
- FIFA / PES
- Mario Kart / Project Cars / Rocket League
- Juegos de Facebook
- ...



## Objetivos

- Introducir al alumno a las **redes de computadores**.
- Dar a conocer y permitir comprender el modelo de **arquitectura de red cliente-servidor**.
- Ofrecer al alumno una visión general de los **problemas y soluciones** al **desarrollar juegos en red**.
- Fomentar el **trabajo en equipo** para la creación de videojuegos.
- Guiar al alumno en la creación de un **producto software**.



## Temario



### Tema 1: Introducción a los juegos en red y a las redes de comunicaciones

Se presentan los retos de las aplicaciones en red y se introducen los principales conceptos necesarios para entender su funcionamiento:

- Qué son y como funcionan las aplicaciones en red.
- Problemas propios de aplicaciones en red.
- Conceptos fundamentales de comunicación en red entre aplicaciones.
- Consideraciones al desarrollar juegos en red.



## Tema 2: Desarrollo en el lado del cliente.

Se estudia el lenguaje JavaScript para construir aplicaciones en el lado del cliente (browser). API REST.

- Desarrollo web en JavaScript, HTML y CSS
- Uso de las API REST



## Tema 3: Desarrollo de juegos con tecnología web.

Se presenta el framework JavaScript Phaser para desarrollo de juegos en el cliente:

- Phaser 3 framework: <https://phaser.io>.
- Amplia galería de ejemplos: <https://phaser.io/examples/>



## Tema 4: Desarrollo en el lado del servidor.

Se estudia el framework express.js con JavaScript para desarrollar el lado del servidor. Implementación de API REST:

- Introducción a express.js
- Desarrollo de comunicación cliente servidor a través de API REST.



## Tema 5: Comunicación asíncrona cliente - servidor

Se estudia el uso de WebSockets para comunicación asíncrona entre nodos:

- Introducción a la comunicación asíncrona con WebSockets.
- Creación de protocolos de mensajes.
- Desarrollo de comunicación asíncrona entre cliente y servidor con WebSockets.



## Metodología y Material Docente



### Metodología

El curso estará conducido por una **metodología basada en proyectos**, en la que los alumnos deberán realizar un proyecto en grupo a lo largo de toda la asignatura.

Se combinarán las **lecciones magistrales síncronas** con la metodología de **clase invertida** para temas concretos.

Se incluirán diferentes **gamificaciones** a lo largo del curso.

Habrá **sesiones de prácticas** que se utilizarán para realizar y corregir la práctica, donde el profesor podrá realizar un seguimiento de cada grupo de prácticas.

El profesor dejará el **material** necesario para el desempeño de la asignatura en el **Aula Virtual**, que incluirá vídeos, presentaciones, ejercicios propuestos y resueltos, etc., para que el alumno estudie de manera autónoma.



## Material teórico

- El material se irá publicando en el Aula Virtual a medida que avance el cuatrimestre.
- Se publicará como asignatura en abierto, todos los materiales son públicos.
- Contará con diapositivas, apuntes, problemas y una guía de estudio.



## Enunciado de la práctica

- Los enunciados de las prácticas se colgarán en el aula virtual.



## Herramientas

Utilizaremos varias **herramientas software y bibliotecas**:

- Visual Studio Code / Tu IDE de confianza.
- Google Chrome/Firefox/Safari
- JavaScript
- Phaser 3
- Node



## Inteligencia Artificial

El uso de ChatGPT, Gemini, Claude o alternativas está permitido a excepción de los exámenes. La única condición que se impone es que seáis capaces de explicar vuestras prácticas, y en caso negativo, se considerarán suspensas.



## Evaluación



### Pruebas de evaluación:

- **40% Pruebas teórico-prácticas**
  - Nota mínima: cada prueba  $\geq 5$
- **40% Práctica**
  - Nota mínima: cada prueba  $\geq 5$
- **15% Resolución de problemas y casos prácticos**
  - No tiene nota mínima
- **5% Participación en clase y otras actividades**
  - No tiene nota mínima

Hay que obtener al menos un 5 de media



## Pruebas teórico-prácticas (40 %)

- 3 exámenes:
  - Examen 1 (Tema 1): 35%.
  - Examen 2 (Tema 2 y 3): 32.5%.
  - Examen 3 (Tema 4): 32.5%.
- Ejercicios online tipo test / preguntas cortas / ejercicios
- Nota mínima en cada prueba 5
- Reevaluable en convocatoria extraordinaria



## Práctica (40 %)

- Se realiza en grupos de 4 personas
- Cada alumno será evaluado individualmente
- Se desarrolla en parte en las sesiones prácticas
- Consta de 4 entregas (fases) obligatorias con ponderaciones 10%, 35%, 25%, 30%.
- Nota mínima en cada fase: 5 puntos
- Cada fase es reevaluable en convocatoria extraordinaria



## Resolución de problemas (15 %)

- Consiste en una fase adicional al juego desarrollado (fase 5).
- Consistirá en una mejora sobre la práctica entregada y la publicación de la misma en plataformas online.
- Nota mínima de 3.
- Reevaluable en convocatoria extraordinaria.



## Participación en clase (5 %)

- Es opcional. No tiene nota mínima.
- Consiste en una serie de actividades que se irán realizando a lo largo del cuatrimestre.
- También se podrán obtener mejoras o correcciones a los apuntes y diapositivas de la asignatura.
- No es reevaluable en convocatoria extraordinaria



## Evaluación extraordinaria



### Evaluación extraordinaria

Para aprobar la asignatura en la convocatoria extraordinaria se deberán realizar las pruebas no superadas durante la evaluación ordinaria.

**Se guardan las notas de:**

- La prueba teórico-práctica
- Cada una de las fases por separado
- La resolución de problemas (fase 5)
- Las actividades de participación en clase



## Dispensa académica

Un alumno con dispensa académica debe realizar los exámenes teóricos-prácticos.

Está obligado a hacer las 4 entregas presenciales de la parte práctica ya que son como exámenes.

Las fechas de las pruebas serán anunciadas con anterioridad.



## No presentado

Un alumno obtendrá la calificación de no presentado únicamente si no se presenta a ninguna prueba evaluable (ya sea de teoría o práctica).



## Calendario aproximado de entregas



## Calendario aproximado de entregas

- **Septiembre:** Registro grupos de prácticas
- **Octubre:** Entrega fase 1 (mes de octubre)
- **Noviembre:** Entrega fase 2 (mes de noviembre)
- **Diciembre:** Entrega fase 3 (mes de diciembre)
- **Enero:** Entrega fase 4 (mes de enero)
- **Enero:** Entrega fase 5 (mes de enero)



## Bibliografía



## Bibliografía

- Material de la asignatura disponible en el [aula virtual](#)
- [Biblioteca virtual](#) para alumnos



## Internet

- Páginas oficiales de los estándares y tecnologías
- Tutoriales, ejemplos y blogs realizados por la comunidad
- Aplicaciones web de código abierto
- ... y todo lo que Google encuentre :)



## Ejemplos de proyectos de otros años

- [Head and Balls](#)
- [Ink Cation](#)
- [Bombs 'R' Us](#)



## 16.2. Introducción a las Redes de Ordenadores

1

# Introducción a las Redes de Ordenadores

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño

ruben.rodriguez@urjc.es

natalia.madrueño@urjc.es

URJC

URJC

2025-09-09



2

## Tabla de contenidos

- [Historia de Internet](#)
- [Infraestructura de Red](#)
- [Modelos de Referencia](#)
- [Rendimiento en Redes](#)



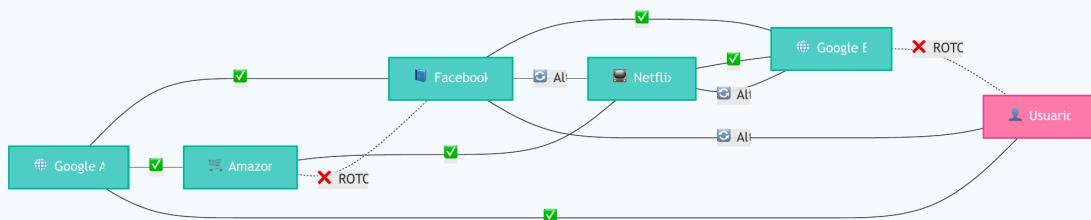
## ¿Qué es Internet?

**Etimología:** “Interconnected Networks”

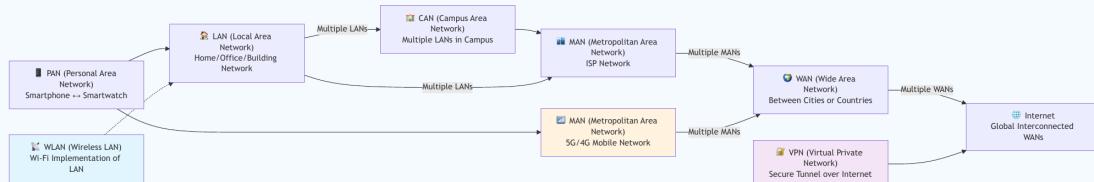
- Red global de redes interconectadas
- Sistema descentralizado
- Múltiples capas jerárquicas

**Características principales:**

- Arquitectura distribuida
- Resiliencia a fallos
- Escalabilidad natural
- Sin control centralizado



## Jerarquía de Redes



- **PAN:** Red personal entre dispositivos cercanos.
- **LAN:** Red local de casa/oficina/edificio.
- **WLAN:** LAN inalámbrica (Wi-Fi).
- **CAN:** Red de campus - conecta múltiples LANs.
- **MAN:** Red metropolitana - cubre una ciudad, incluye redes de ISP y móviles (4G/5G)
- **WAN:** Red de área amplia - conecta ciudades o países.
- **Internet:** Red global - interconexión de todas las WANs del mundo

## Ejemplo: Mensaje Madrid → Tokio

Smartphone María (WiFi) en Madrid -> Takeshi LAN en la Universidad de Tokio

1. **Origen LAN Madrid:** Smartphone → Router WiFi
2. **Router local → MAN:** ISP local → MAN Madrid
3. **MAN → WAN nacional:** MAN Madrid → WAN España
4. **WAN → Internet global:** España → Backbone internacional
5. **Llegada a Japón:** WAN Japón → MAN Tokio
6. **MAN → CAN:** MAN Tokio → Universidad
7. **CAN → LAN:** Campus → LAN específica
8. **Destino final:** LAN → Dispositivo de Takeshi

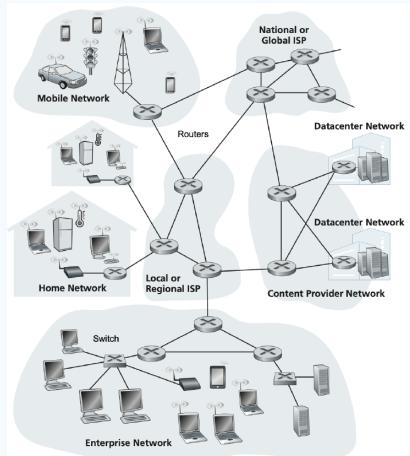


## Un caso un poco más real

- Probad a ejecutar en vuestras terminales `traceroute www.google.es` (`tracert www.google.es` en Windows)
- ¿Qué información estáis obteniendo?
- Comparadla con vuestros compañer@s. ¿Es la misma?



## Internet simplificado



## Componentes Clave

### Router

- Conecta **diferentes redes**
- Usa direcciones **IP**
- Enrutamiento “hop by hop”
- Opera entre redes distantes

### Switch

- Conecta dispositivos en **misma red**
- Usa direcciones **MAC**
- Entrega local inteligente
- Opera dentro de la LAN



## En nuestras casas

Entonces... ¿Esto que es?



## Identificadores en Red

### Dirección IP

- “Dirección postal”
- Localiza en la red
- Ejemplo: 192.168.1.100
- Puede cambiar

### Dirección MAC

- “DNI del dispositivo”
- Única y permanente
- Asignada por fabricante
- No cambia nunca

### Protocolo ARP

- “Directorio telefónico”
- Traduce IP ↔ MAC
- Permite entrega final
- Opera localmente

**Ejercicio**

Prueba a ejecutar `ifconfig` en tu terminal MacOS/Linux o `ipconfig` en Windows. ¿Qué ves?.



## Protocolos de Red

**Protocolo:** Serie de pasos bien definidos que especifican cómo intercambiar información entre dispositivos

### Analogía del tráfico urbano

- **Sin protocolos:** Caos total, pérdida de información
- **Con protocolos:** Flujo ordenado, comunicación efectiva

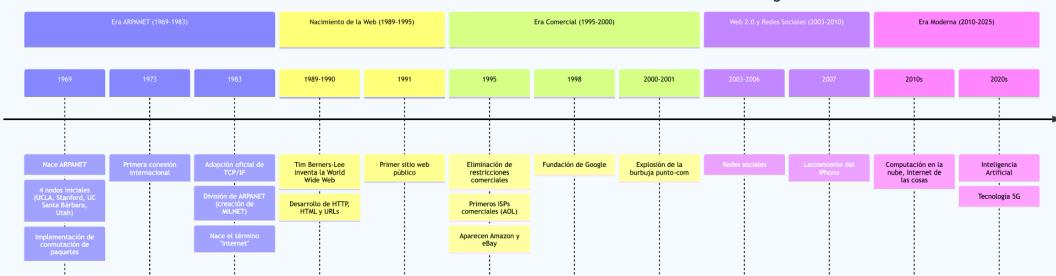


## Historia de Internet



## Linea temporal de Internet.

Evolución de Internet: De ARPANET a la Era Digital



## Infraestructura de Red

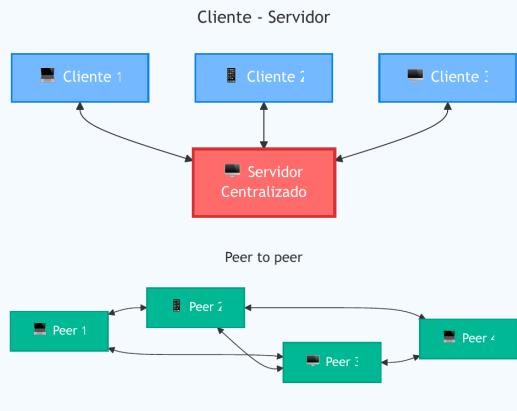


## Sistemas Terminales (End Systems)

Hosts (End systems): Son los dispositivos que **usan** Internet como PCs, smartphones, IoT, servidores. Ejecutan aplicaciones de red.

### Clasificación

- **Clientes:** Solicitan servicios
- **Servidores:** Proporcionan servicios
- Roles dinámicos (P2P)



## Redes de Acceso

Redes de acceso: Es la red en la que se conectan los host con el router de borde.

### Tecnologías host → router

- WiFi 6: 200-400 Mb/s
- Ethernet: 10 Gb/s
- 4G LTE: 50/15 Mb/s
- 5G: 300/50 Mb/s

### Características

- Alcance limitado
- Velocidades variables
- Medios compartidos vs dedicados

## Tecnologías WAN

Router de borde: Router que conecta la red de acceso con el núcleo de la red.

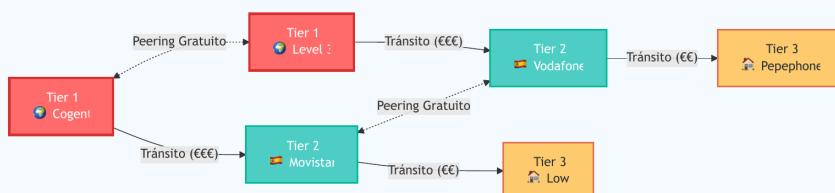
### Tecnologías comunes:

Tecnología	Velocidad típica	Estado 2025
DSL/VDSL	50/15 Mb/s	En declive
Cable HFC	300/30 Mb/s	Estable
FTTH PON	1000/1000 Mb/s	En expansión
FTTH P2P	10000/10000 Mb/s	Premium
Satelital	100/20 Mb/s	Nicho



## Núcleo de la Red: ISPs

ISP (Internet Service Providers): Son los componentes del núcleo de la red y proporcionan interconexión entre diferentes redes.



**Tier 1**

**Tier 2**

**Tier 3**

- Cobertura regional/nacional
- Pagan tránsito a Tier 1
- Peering selectivo

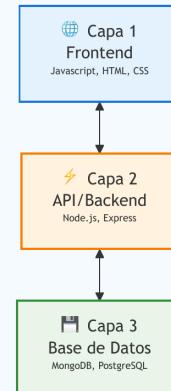


## Modelos de Referencia



### Arquitecturas por Capas

- Cada capa = responsabilidad específica
- Servicios a capa superior
- Usa servicios de capa inferior
- Desarrollo independiente



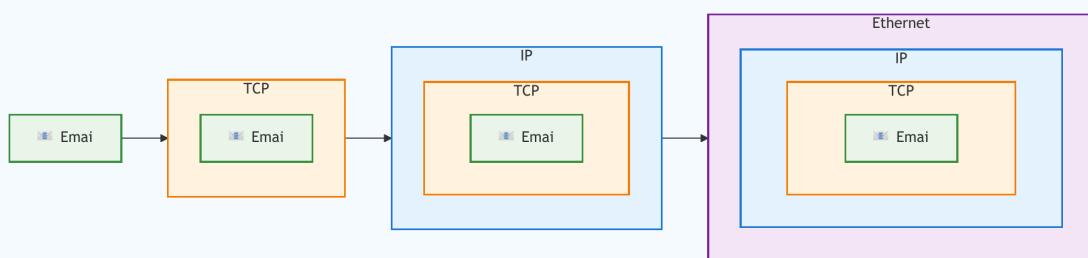
## Encapsulación

- Cada capa añade headers
- Datos superiores = payload
- No modifica contenido interno

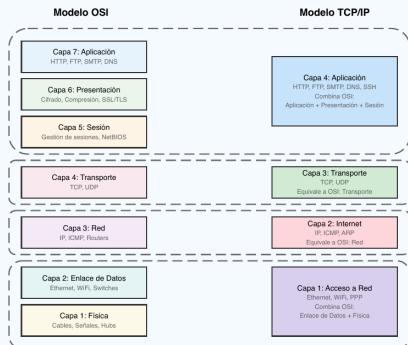


## Encapsulación + Arquitectura por capas

Encapsulación



## Modelos OSI vs TCP/IP



- **OSI:** 7 capas, modelo teórico
- **TCP/IP:** 4 capas, usado en Internet

⚠ Nota: TCP/IP no es un protocolo, hace referencia a una pila de protocolos. Además, no tiene porque utilizar necesariamente TCP, podría ser UDP.



## Nivel de Aplicación

Es el nivel en que desarrollamos aplicaciones.

### OSI (Capas 7, 6, 5)

- **Aplicación:** HTTP, FTP, DNS
- **Presentación:** Cifrado, compresión
- **Sesión:** Control de diálogos

### TCP/IP

- Una sola capa integrada
- Protocolos: HTTP/HTTPS, SMTP, FTP, DNS
- Más práctico



## Nivel de Transporte

Gestiona la comunicación extremo a extremo entre aplicaciones.

### Capa 4 (ambos modelos)

#### TCP

- Comunicación confiable
- Control de flujo
- Entrega ordenada
- Corrección de errores

#### UDP

- Comunicación rápida
- Sin garantías
- Ideal para tiempo real
- Menor overhead



## Nivel de Red/Internet

Se encarga de encontrar el mejor camino para enviar datos a través de múltiples redes. En otras palabras, se encarga del **enrutamiento** de paquetes.

### Capa 3 OSI / Capa Internet TCP/IP

#### Protocolos:

- IP: Protocolo principal
- ICMP: Control y errores
- ARP: Resolución de direcciones
- OSPF, BGP: Protocolos de enrutamiento

#### Ejercicio

Prueba a ejecutar `ping www.google.es` en tu terminal. ¿Qué ves?.



## Nivel de Acceso Físico

Controla cómo los datos se transmiten físicamente a través del medio de comunicación.

### OSI (Capas 2 y 1)

- **Enlace:** Control de errores, MAC
- **Física:** Señales, voltajes

### TCP/IP

- Capa de Acceso a Red
- Combina ambas funciones
- Ethernet, WiFi, etc.



## Rendimiento en Redes



## Métricas Principales

### Latencia

- Tiempo que tarda un paquete en llegar a su destino.
- “Velocidad del vehículo”
- Medida en ms

 **Nota:** 1 MB/s = 8 Mb/s

### Throughput (Tasa de Transferencia Efectiva)

- Datos enviados por cantidad de tiempo.
- “Número de carriles”
- Medido en Mb/s o Gb/s



## Throughput vs Bandwidth

### Bandwidth

- Capacidad **máxima teórica**
- Límite físico del canal
- Condiciones ideales

### Throughput

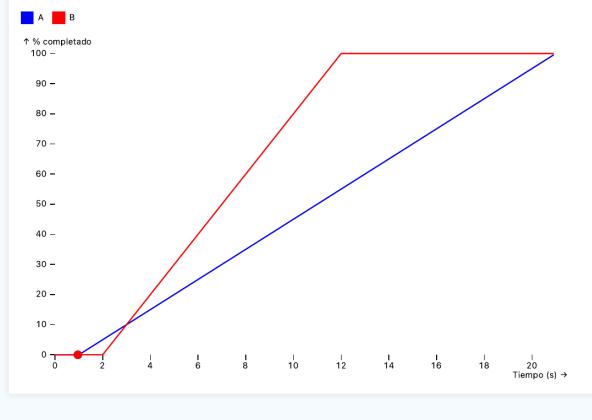
- Transferencia **real**
- Limitado por el componente más lento
- Condiciones reales



## Latencia vs Throughput

Comparativa del efecto de la latencia y throughput en el tiempo para enviar una cantidad de datos.

Tamaño total	<input type="text" value="200"/>
Latencia (A)	<input type="text" value="1"/>
Throughput (A)	<input type="text" value="10"/>
Latencia (B)	<input type="text" value="2"/>
Throughput B	



## Componentes de la Latencia

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

- **dproc:** Procesamiento en router (microsegundos)
- **dqueue:** Espera en buffer (variable con tráfico)
- **dprop:** Propagación por el medio (d/s)
- **dtrans:** Transmisión de datos (L/R)



## RTT

RTT (Round trip time): Tiempo total que tarda un paquete en ir desde el origen hasta el destino y volver de vuelta (ida + vuelta).

- La latencia no tiene porque ser simétrica.
- Generalmente la descarga es más rápida que la subida.
- Por lo tanto, el RTT es un valor muy importante en aplicaciones interactivas.



## Comparación: Fibra vs 5G

Factor	Fibra Óptica	5G
<b>Propagación</b>	67% velocidad luz	100% velocidad luz
<b>Procesamiento</b>	~0.1ms/salto	~4ms (estación radio)
<b>Cola</b>	Baja congestión	Alta congestión
<b>Transmisión</b>	Hasta 10 Gb/s	< 1 Gb/s

**Resultado:** Fibra generalmente más rápida y estable



## Jitter: Variabilidad de Latencia

Jitter: Variación en el tiempo de llegada de los paquetes que causa inconsistencia en la comunicación.

### Ejemplo comparativo

**Escenario 1** (Bajo jitter):

- Paquetes: 50, 52, 48, 51 ms
- Promedio: 50.25 ms
- Variación: 1.48 ms

**Escenario 2** (Alto jitter):

- Paquetes: 28, 68, 43, 62 ms
- Promedio: 50.25 ms
- Variación: 15.82 ms

**Impacto:** Voz entrecortada, saltos en video, degradación en juegos



## Requisitos para Videojuegos

### RTT máximo tolerado

Género	Tolerancia	Ejemplo
Fighting	16-50ms	Street Fighter
FPS Competitivo	20-50ms	Counter-Strike
Racing	50-100ms	Gran Turismo
RTS	100-200ms	StarCraft
MMORPG	Variable	World of Warcraft
Turn-based	500ms+	Civilization



## Pérdida de Paquetes

### Causas principales

- **Congestión:** Buffers llenos en routers
- **Corrupción:** Interferencias electromagnéticas
- **Radiación cósmica:** ~1 error/256MB/día

### Soluciones

- Protocolos de capas superiores (TCP)
- Retransmisión automática
- Códigos de corrección de errores
- Interpolación de información



## Resumen

- Internet es un **sistema distribuido y descentralizado**
- Evolución desde 4 hosts (1969) a >100B dispositivos (2025)
- **Infraestructura jerárquica:** PAN → LAN → MAN → WAN → Internet
- **Modelos de capas:** OSI (teórico) vs TCP/IP (práctico)
- **Rendimiento:** Balance entre latencia y throughput
- **Aplicaciones críticas:** Videojuegos requieren <50ms para competitivo



## 16.3. Capa de Acceso

1

### Capa de Acceso a la Red

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño  
ruben.rodriguez@urjc.es natalia.madrueño@urjc.es  
URJC URJC

2025-09-09



 2

### Tabla de contenidos

- [Introducción](#)
- [Funciones Principales](#)
- [Dispositivos de Capa 2](#)
- [Protocolos Principales](#)
- [Consideraciones Prácticas](#)
- [Resumen](#)

 2

# Introducción



## ¿Qué es la 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

### ¿Qué hace?

- Maneja los aspectos físicos de la transmisión
- Controla el acceso al medio compartido
- Se ejecuta en el host y en el núcleo de la red
- Garantiza transmisión confiable entre nodos adyacentes

### Posición en los modelos:

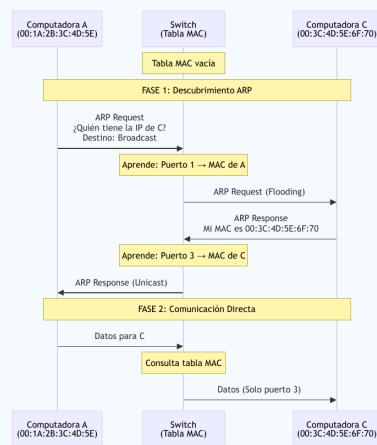


### ! Importante

**Responsabilidad principal:** Garantizar que los datos puedan transmitirse de manera confiable entre nodos adyacentes en la red



## Ejemplo: Comunicación en Red Local



## Funciones Principales



## 1. Control de Acceso al Medio (MAC)

Coordina cómo múltiples dispositivos comparten un medio de transmisión común

### Ethernet Half-Duplex: CSMA/CD

**Carrier Sense Multiple Access with Collision Detection**

1. Escuchar el medio antes de transmitir
2. Si está libre → transmitir
3. Si hay colisión → detectar
4. Aplicar backoff exponencial
5. Reintentar transmisión

*Solo puede transmitir en una dirección a la vez. En Full-Duplex no necesitaríamos realizar controles.*

### Redes Inalámbricas: CSMA/CA

**Carrier Sense Multiple Access with Collision Avoidance**

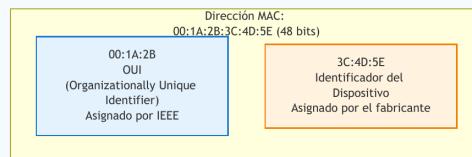
1. Esperar tiempo aleatorio antes de transmitir
2. Usar acknowledgments para confirmar recepción
3. Protocolo RTS/CTS para problema del nodo oculto

*La detección de colisiones es impráctica en radio*



## 2. Direccionamiento Físico

Opera a nivel de hardware, independiente de protocolos superiores, usando direcciones MAC únicas



### Unicast

- Un único destinatario
- Dirección específica del dispositivo

### Broadcast

- Todos los dispositivos
- FF:FF:FF:FF:FF:FF

### Multicast

- Grupo específico
- Primer bit = 1

#### Tip

Las direcciones MAC son como el DNI del dispositivo: únicas, estáticas y cada dispositivo tiene una



### 3. Detección y Corrección de Errores

Garantiza la integridad de los datos transmitidos a través del medio físico

#### Códigos de Redundancia Cíclica (CRC)

##### Proceso CRC:

1. Generar polinomio matemático sobre datos
2. Agregar Frame Check Sequence (FCS) al final
3. Receptor recalcula el CRC
4. Comparar con el recibido
5. Detectar errores de 1 bit y múltiples bits

##### Forward Error Correction (FEC):

- No solo detecta, también **corrige** errores
- Códigos Hamming: errores de 1 bit
- Reed-Solomon: errores en ráfagas
- Especialmente importante en medios inalámbricos

##### Checksums simples:



### 3. Detección y Corrección de Errores (Ejemplo)

Introducir un mensaje de 4 bits (1011) y os saldrá a la derecha el mensaje codificado

""

#### Mensaje (4 bits)

e.g. 1011

Aquí podréis meter el mensaje codificado y saber si ha habido cambios en un bit y corregirlos (Hamming).

"Enter exactly 7 bits"

#### Codeword (7 bits)

e.g. 0110011



## 4. Control de Tamaño

Maneja las limitaciones de tamaño impuestas por diferentes tecnologías de red

### Maximum Transmission Unit (MTU)

Tecnología	MTU (bytes)	Características
Ethernet	1500	Estándar más común en LAN
Token Ring	4464	Tecnología legacy
FDDI	4352	Fiber Distributed Data Interface
PPP	Variable (~1500)	Para compatibilidad con Ethernet

#### ⚠️ Advertencia

Si datos > MTU → La Capa de Acceso a la Red **descarta automáticamente** el paquete

Nota: El MTU determina el tamaño máximo de datos que puede transportar una sola trama



## 5. Sincronización y Temporización

Coordina el timing entre dispositivos para asegurar la correcta interpretación de señales digitales

### Niveles de sincronización

#### Tipos de sincronización:

- **Sincronización de bit:** Determina límites temporales de cada bit
- **Sincronización de trama:** Identifica inicio y fin de cada trama
- **Sincronización de símbolo:** Para modulaciones complejas (QAM)

#### ¿Por qué es crítica?

- En redes de alta velocidad, pequeñas diferencias causan errores
- Establece marcos de tiempo comunes
- Permite interpretación correcta de señales
- Esencial para comunicación digital confiable



## 6. Gestión de Topología

Descubre y mantiene información sobre la estructura física de la red

### Componentes principales

#### Mantenimiento de enlaces:

- Keepalive messages
- Detección proactiva de fallos
- Antes de afectar tráfico de usuarios

#### Protocolos de detección:

#### Prevención de bucles:

- Spanning Tree Protocol (STP)
- Previene bucles en topologías redundantes
- Evita tormentas de broadcast

#### Adaptación automática:

- Detecta cambios en la topología
- Responde a fallos de enlaces
- Incorpora nuevos dispositivos



## 7. Control de Calidad de Servicio (QoS)

Prioriza diferentes tipos de tráfico según su importancia y requisitos de rendimiento

### Mecanismos de gestión

#### Gestión de buffers:

- **Weighted Fair Queuing:** Recursos proporcionales según importancia
- **Priority Queuing:** Tráfico crítico tiene precedencia
- **Random Early Detection:** Descarta proactivamente antes de saturación

#### Aplicaciones beneficiadas:

- 📺 Video en tiempo real
- 📞 VoIP (Voz sobre IP)
- 🎮 Gaming online
- 💼 Aplicaciones críticas de negocio

*Fundamental para aplicaciones sensibles al tiempo*



QoS garantiza que aplicaciones críticas reciban el ancho de banda necesario incluso en momentos de congestión



## Dispositivos de Capa 2



### Switches: Evolución y Tipos

#### Switches No Gestionados

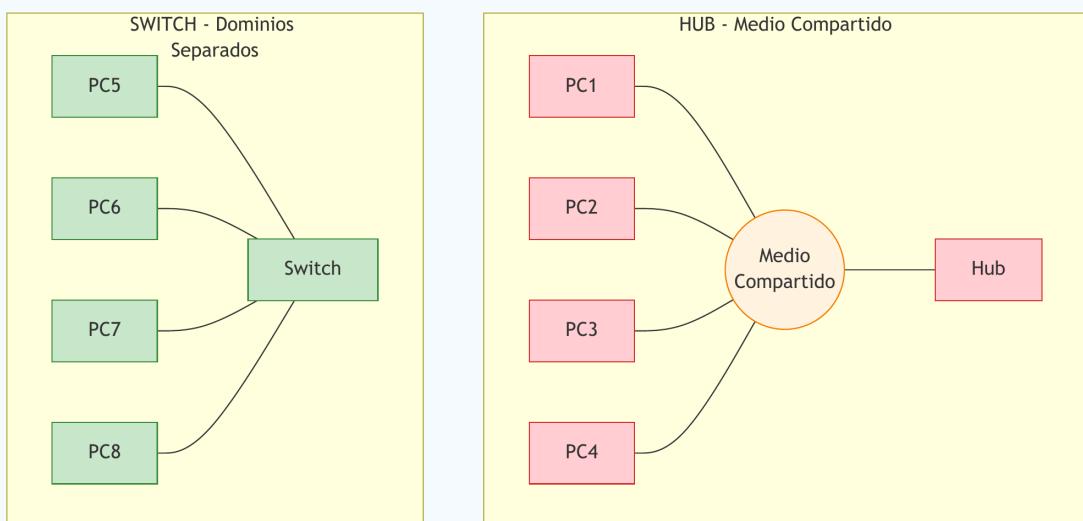
- Plug-and-play
- Aprendizaje MAC automático
- Redes pequeñas/domésticas
- Sin configuración

#### Switches Gestionados

- VLANs y segmentación
- QoS y priorización
- SNMP para monitorización
- Seguridad 802.1X



## Dominios de Colisión: Switch vs Hub



## Otros Dispositivos de Acceso

Dispositivo	Función	Características	Aplicación
<b>Access Points</b>	WiFi ↔ Cableado	CSMA/CA, Beamforming	Redes inalámbricas
<b>Repetidores</b>	Extensión alcance	Regeneración señal	Superar distancia
<b>Media Converters</b>	Cambio de medio	Fibra ↔ Cobre	Migración gradual
<b>Transceivers</b>	Modular	SFP/SFP+/QSFP	Flexibilidad

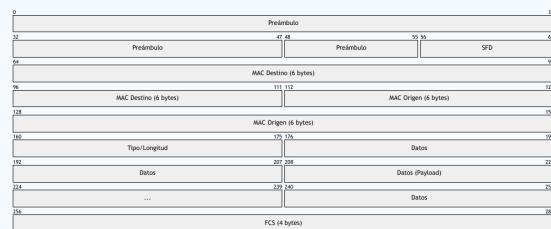


# Protocolos Principales



## Ethernet (IEEE 802.3)

### Estructura de Trama Ethernet



#### Campos principales:

- Preámbulo: Sincronización
- MACs: Identificación única
- Tipo: Protocolo superior (IPv4: 0x0800)
- Payload: Datos + padding si < 46 bytes

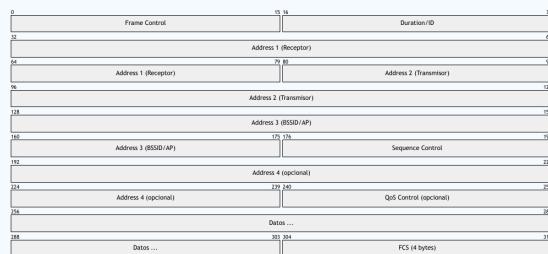
#### Evolución de velocidades:

- 10Base-T: 10 Mbps
- Fast Ethernet: 100 Mbps
- Gigabit: 1 Gbps
- 10G/40G/100G Ethernet



## WiFi (IEEE 802.11)

### Trama WiFi: Mayor complejidad



La trama es considerablemente más compleja debido a que estamos en un medio compartido y puede haber repetidores, así como comunicaciones directas.

## Evolución de Estándares WiFi

Generación	Estándar	Velocidad Max	Bandas	Año
WiFi 4	802.11n	600 Mbps	2.4/5 GHz	2009
WiFi 5	802.11ac	3.5 Gbps	5 GHz	2014
WiFi 6	802.11ax	9.6 Gbps	2.4/5 GHz	2019
WiFi 6E	802.11ax	9.6 Gbps	+ 6 GHz	2020
WiFi 7	802.11be	46 Gbps	2.4/5/6 GHz	2024

### Mejoras clave:

- MIMO (múltiples antenas)
- OFDMA (mejor uso espectro)
- Beamforming direccional (dirigir hacia un punto en concreto)

### Trade-offs de bandas:

- 2.4 GHz: Mayor alcance, menor velocidad
- 5 GHz: Mayor velocidad, menor alcance
- 6 GHz: Máxima velocidad, mínimo alcance

## PPP y Frame Relay

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

#### Características:

- Enlaces punto a punto
- Detección de errores
- Autenticación (PAP/CHAP)
- Configuración IP automática

#### Uso actual:

- Enlaces de respaldo
- Conexiones satelitales
- Algunas VPNs

### Frame Relay

#### Características:

- WAN con circuitos virtuales
- Comutación de tramas
- Control de congestión

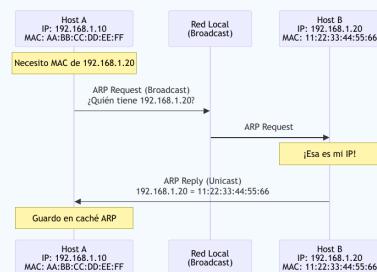
#### Estado:

- Reemplazado por MPLS
- Legacy en empresas antiguas
- Conceptos aún relevantes



## ARP: Address Resolution Protocol

### Traducción IP → MAC



#### Proceso ARP:

#### Tipos de ARP:



## Consideraciones Prácticas



### Límites Físicos y Distancias

#### Cable de Cobre (UTP/STP)

##### Causas de la limitación:

- Atenuación de señal
- Interferencia electromagnética
- Crosstalk entre pares
- Degradación con distancia

##### Soluciones:

- Switches cada 100m
- Repetidores/Extensores
- Fibra óptica (kilómetros)
- Enlaces inalámbricos

#### Comparación de Medios

Medio	Distancia Max	Velocidad	Interferencia
UTP Cat5e	100m	1 Gbps	Alta
UTP Cat6a	100m	10 Gbps	Media
Fibra MM	2 km	10 Gbps	Nula
Fibra SM	100+ km	100 Gbps	Nula



## Ejemplo Práctico: Verificación de Configuración

### Comandos útiles para verificar la capa de acceso

Ver información de red:

```
1 # Linux/Mac - Ver dirección MAC
2 ifconfig
3
4 # Windows - Ver dirección MAC
5 ipconfig /all
6
7 # Ver tabla ARP
8 arp -a
```

Ejemplo de salida ARP:

```
192.168.1.1 00:1a:2b:3c:4d:5e
192.168.1.10 00:2b:3c:4d:5e:6f
192.168.1.20 00:3c:4d:5e:6f:70
```

Muestra las asociaciones IP-MAC en la caché local



## Resumen



## Puntos Clave

- La **Capa de Acceso a la Red** maneja la transmisión física y el control de acceso al medio compartido
- Combina las funciones de las **capas física y de enlace del modelo OSI**
- **Control de acceso al medio**: CSMA/CD (Ethernet) vs CSMA/CA (WiFi)
- **Direcciones MAC**: 48 bits, únicas por dispositivo (OUI + ID dispositivo)
- **Switches** evolucionaron desde hubs, creando dominios de colisión independientes
- **Detección de errores** mediante CRC y técnicas FEC
- **MTU** define el tamaño máximo de trama (Ethernet: 1500 bytes)
- **ARP** resuelve la traducción entre direcciones IP y MAC



## 16.4. Capa de Red

### Capa de Red

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez    Natalia Madrueño  
ruben.rodriguez@urjc.es    natalia.madrueño@urjc.es  
URJC                          URJC

2025-09-09



## Tabla de contenidos

- [Introducción](#)
- [Funciones Fundamentales](#)
- [Modelos de Servicio](#)
- [Dispositivos de Capa de Red](#)
- [Protocolo](#)
- [Protocolos Complementarios](#)
- [Resumen](#)



## Introducción

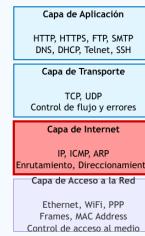


## ¿Qué es la Capa de Red?

La Capa de Red es el **tercer nivel del modelo TCP/IP** y forma el núcleo del sistema de comunicaciones de Internet

### Función principal:

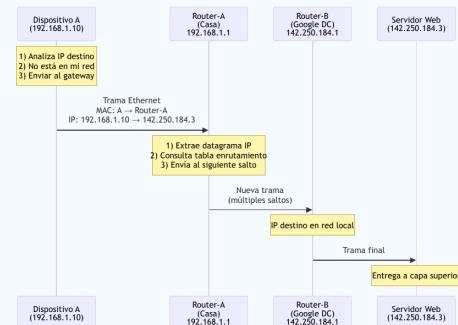
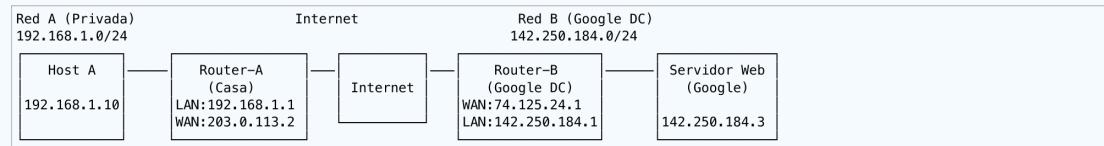
- Proporcionar comunicación end-to-end entre dispositivos
- Potencialmente separados por múltiples redes intermedias
- Independiente de la tecnología subyacente



#### ⚠ Importante

La comunicación funciona de igual forma independientemente del medio físico utilizado

## Ejemplo Simplificado: Host A → Servidor Google



# Funciones Fundamentales



## Enrutamiento vs Reenvío

### Enrutamiento

Proceso global que determina rutas óptimas

- Considera toda la topología de red
- Tiempo: segundos a minutos
- Algoritmos: RIP, OSPF, BGP
- Genera tabla de enrutamiento completa

### Reenvío

Proceso local de mover paquetes

- Puerto entrada → puerto salida
- Tiempo: microsegundos
- Implementado en hardware
- Usa tabla de reenvío optimizada



Los algoritmos de enrutamiento generan la tabla de enrutamiento → se traduce en tabla de reenvío con next-hop



## Responsabilidades por Dispositivo

### Host Emisor

- Recibe segmentos de TCP/UDP
- Encapsula en datagramas IP
- Fragmenta si excede MTU
- Determina si destino es local o remoto

### Routers Intermedios

- Examinan cabecera IP (dirección destino)
- Consultan tabla de enrutamiento
- Determinan siguiente salto
- Reenvían por interfaz correspondiente

### Host Receptor

- Reensambla fragmentos
- Verifica integridad (checksum)
- Extrae segmentos
- Entrega a capa de transporte



## Modelos de Servicio



## Redes de Circuitos Virtuales

**Funcionamiento en 3 fases:**

1. **Establecimiento:** SETUP, reserva recursos
2. **Transferencia:** Usa VC ID, ruta fija
3. **Terminación:** TEARDOWN, libera recursos

**Ventajas:**

- QoS predecible
- Overhead reducido (solo VC ID)
- Orden garantizado

**Desventajas:**

- Complejidad alta
- Mantiene estado por conexión
- Rigidez ante cambios

**Tecnologías:** ATM, Frame Relay, X.25, MPLS



## Redes de Datagramas

**Características:**

- Cada paquete tratado independientemente
- Sin estado de conexión en routers
- Dirección destino completa en cada paquete
- Diferentes rutas posibles por paquete

**Ventajas:**

- Simplicidad de diseño
- Robustez ante fallos
- Flexibilidad y balanceo de carga
- Escalabilidad superior

**Limitaciones:**

- Sin garantías QoS
- Posible desorden de paquetes
- Servicio best-effort

**Importante**

Fundamento de Internet por su adaptabilidad a condiciones cambiantes



## Circuitos Virtuales vs Datagramas

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

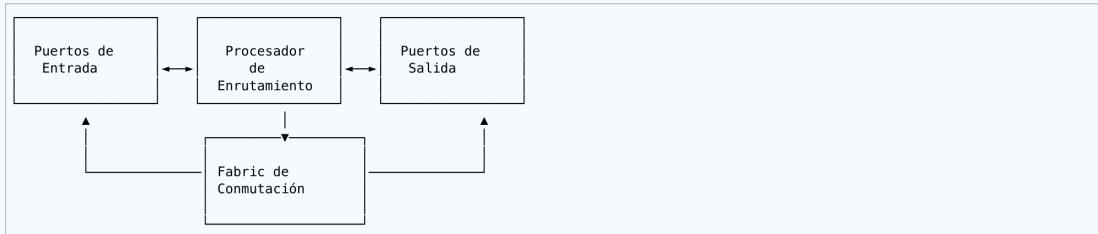
Internet usa el modelo de **datagramas** por su simplicidad, robustez y escalabilidad



## Dispositivos de Capa de Red



## Arquitectura del Router



### Plano de Control:

- Ejecuta enrutamiento (software)
- Genera tablas de enrutamiento

### Plano de Datos:

- Ejecuta reenvío (hardware)
- Puertos entrada/salida + fabric



## Proceso de Reenvío de Paquetes

- Recepción:** Llega paquete, se procesa capa enlace, se extrae datagrama IP
- Verificación:** Checksum de cabecera, TTL > 0
- Decisión:** Extrae IP destino, aplica longest prefix matching
- Modificación:** Decrementa TTL, recalcula checksum
- Resolución:** ARP si necesario para MAC siguiente salto
- Encapsulación:** Nueva trama según protocolo salida
- Transmisión:** Envío por interfaz física

### ⚠️ Advertencia

Si TTL llega a 0 → descarta paquete y envía ICMP "Time Exceeded"



## Switches Layer 3

Un switch Ethernet (L3) que combina switching de alta velocidad por hardware (ASICs) con capacidades básicas de enrutamiento IP para redes LAN.

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

Los switches L3 combinan la velocidad del switching con las capacidades del routing. Útiles en redes locales.



## Protocolo



## Protocolo IP

### Características fundamentales:

- **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

### Responsabilidades principales:

- Define estructura de datagramas
- Establece sistema de direccionamiento
- Mecanismos básicos de entrega
- Fragmentación y reensamblado
- Control de vida del paquete (TTL)

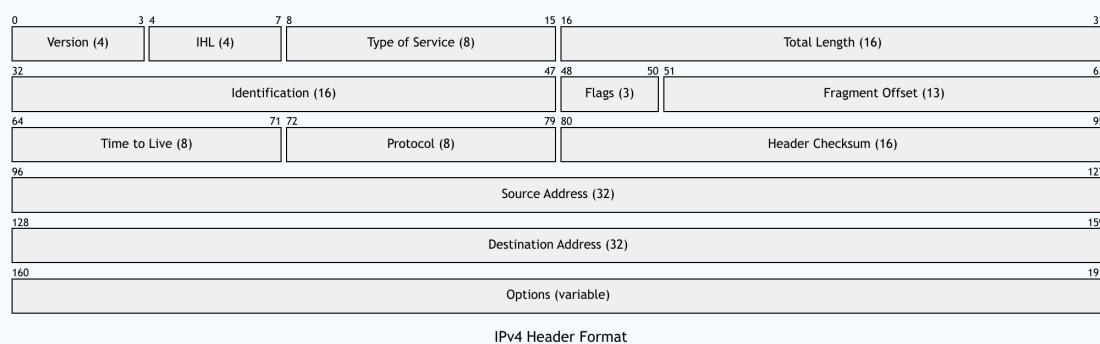
 **Importante**

**IP NO garantiza:** Entrega, orden entre datagramas diferentes, ni detección de duplicados. Estas funciones se delegan a capas superiores (TCP).

**Versiones:** IPv4 (32 bits, diseñado años 70) e IPv6 (128 bits, soluciona limitaciones IPv4)



## IPv4: Estructura Básica



- Source address y Destination address para identificación de hosts
- Protocol para identificar el protocolo de la capa superior
- Checksum para integridad de datos.



## IPv4: Direccionamiento

### Formato: 32 bits (4 octetos)

Ejemplo: 192.168.1.1 con máscara 255.255.255.0 (/24)

- Parte azul: **Red**
- Parte roja: **Host**
- Total direcciones:  $2^{32} \approx 4.3$  mil millones

### Obtención dirección de red:

192.168.1.1 AND 255.255.255.0 = 192.168.1.0



La división red/host permite enruteamiento jerárquico eficiente



## Sistema de Clases (Histórico)

Clase	Rango	Bits Red	Bits Host	Redes	Hosts/Red	Uso
A	0.0.0.0 - 127.255.255.255	7	24	126	16,777,214	ISPs, gobiernos
B	128.0.0.0 - 191.255.255.255	14	16	16,384	65,534	Universidades
C	192.0.0.0 - 223.255.255.255	21	8	2,097,152	254	Empresas pequeñas



Problema: Organización con 1,000 hosts

- Clase B: desperdicia 64,534 direcciones (98.5%)
- Clase C: insuficiente



## CIDR: Solución Moderna

### Classless Inter-Domain Routing

Notación: 192.168.1.0/24 → 24 bits para red

Ventajas:

- Asignación flexible (cualquier potencia de 2)
- Utilización: 20-30% → 95-98%
- Agregación de rutas eficiente

### Longest Prefix Matching

Seleccionamos en nuestra tabla de rutas aquella con la coincidencia **mas grande**.

Tabla con rutas:

- 192.168.0.0/16
- 192.168.1.0/24 ← **Seleccionada**
- 192.168.1.128/25



## Direcciones Especiales

### Direcciones Reservadas

Dirección	Propósito	Descripción
0.0.0.0/32	Este host	Sin IP configurada (DHCP)
127.0.0.0/8	Loopback	Pruebas locales (127.0.0.1)
255.255.255.255/32	Limited broadcast	Solo red local
x.x.x.0	Dirección de red	Identifica la red
x.x.x.255	Directed broadcast	Broadcast a red específica

### Rangos Privados (RFC 1918)

- **10.0.0.0/8** → 16.7 millones hosts (grandes organizaciones)
- **172.16.0.0/12** → 1 millón hosts (empresas medianas)
- **192.168.0.0/16** → 65,000 hosts (hogares/oficinas)

No enrutables en Internet público → Requieren NAT



## Direcciones Especiales (Práctica)

- Broadcast: Cuando queremos enviar un paquete a todos los dispositivos de la red local. Ejemplo: 192.168.1.255 (para red 192.168.1.0/24)

```
1 ifconfig
2 ipconfig /all en windows
```

- Gateway: Dirección del router que conecta nuestra red local con otras redes/Internet. Ejemplo: 192.168.1.1 (típicamente la primera IP utilizable de la red)

```
1 route -n get default (macOS/Linux)
2 ip route show default (Linux)
3 ipconfig /all (Windows)
```



## Fragmentación

### MTU (Maximum Transmission Unit)

Tecnología	MTU (bytes)
Ethernet	1500
Token Ring	4464
FDDI	4352
PPP	Variable (~1500)

#### Proceso:

1. Si datagrama > MTU → fragmentar
2. Enviar fragmentos por separado
3. Reensamblar en destino
4. IP preserva integridad del datagrama original

**! Importante**

IP garantiza orden dentro del datagrama, NO entre datagramas diferentes



## IPv6: La Evolución

### Motivación

#### Limitaciones IPv4:

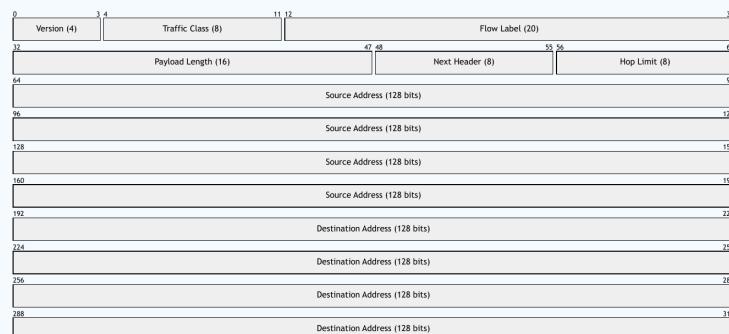
- Agotamiento de direcciones ( $4.3 \times 10^9$ )
- Fragmentación ineficiente en routers
- Sin autoconfiguración
- Seguridad opcional (IPSec)
- QoS limitado

#### Características IPv6

- **Direcciones:** 128 bits ( $3.4 \times 10^{38}$  direcciones)
- **Cabecera:** Fija 40 bytes
- **Sin checksum** en cabecera
- **IPSec obligatorio**
- **Autoconfiguración SLAAC**
- **Mejor QoS** (Traffic Class, Flow Label)



## Cabecera IPv6



- Migración gradual IPv4 → IPv6 mediante mecanismos de interoperabilidad
- Las direcciones IP ahora ocupan el doble de tamaño
- Se elimina el checksum
- El siguiente protocolo es ahora “Next Header”



## Protocolos Complementarios



### ICMP: Control y Diagnóstico

#### Internet Control Message Protocol

##### Características:

- Complementario a IP
- Usa IP para transporte
- No orientado a conexión
- Implementación obligatoria (en IPv6)

#### Tipos de Mensajes

##### Mensajes de Error:

##### Mensajes de Consulta:

- Echo Request/Reply (Type 8/0)
- Timestamp Request/Reply (Type 13/14)



## ICMP: Herramientas de Diagnóstico

### Ping - Verificación de Conectividad

```

1 $ ping 8.8.8.8
2 PING 8.8.8.8 (8.8.8.8): 56 data bytes
3 64 bytes from 8.8.8.8: icmp_seq=0 ttl=55 time=15.1 ms
4 64 bytes from 8.8.8.8: icmp_seq=1 ttl=55 time=14.9 ms

```

Echo Request (Type 8) → Echo Reply (Type 0)

### Traceroute - Descubrimiento de Ruta

```

1 $ traceroute google.com
2 1 192.168.1.1 (192.168.1.1) 3.414 ms
3 2 100.70.0.1 (100.70.0.1) 5.245 ms
4 3 10.14.0.53 (10.14.0.53) 7.091 ms
5 4 * * *
6 5 72.14.195.182 (72.14.195.182) 4.665 ms

```

Incrementa TTL progresivamente → Time Exceeded (Type 11)



## NAT: Network Address Translation

### Problema y Solución

#### Problema:

- Agotamiento direcciones IPv4
- Múltiples dispositivos, una IP pública

#### Solución NAT:

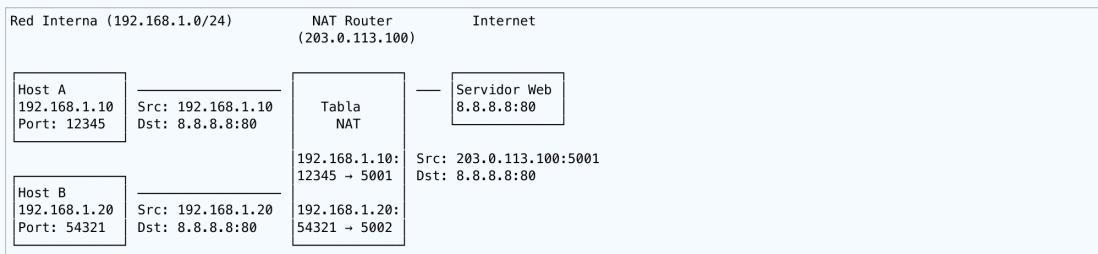
- Usa direcciones privadas internamente
- Traduce a IP pública en router
- Mantiene tabla de traducción

#### Funcionamiento:

1. Host interno inicia conexión
2. Router reemplaza IP:puerto origen
3. Registra en tabla NAT
4. Respuesta llega a router
5. Consulta tabla y reenvía internamente



## NAT: Ejemplo Práctico



- Traducción de direcciones: El router NAT convierte las IP privadas a su IP pública
- Mapeo de puertos: Asigna puertos únicos externos (5001, 5002) a cada host interno para distinguir las conexiones simultáneas en la tabla NAT
- Enmascaramiento de red interna: Permite que múltiples dispositivos privados comparten una sola IP pública



## NAT: Limitaciones y Soluciones

### Limitaciones

- No permite conexiones entrantes directas
- Complicaciones con protocolos que embeben IPs
- Pérdida del principio end-to-end

### Técnicas para Atravesar NAT

#### Hole Punching:

- Ambos conectan simultáneamente
- Crea “agueros” temporales

#### STUN:

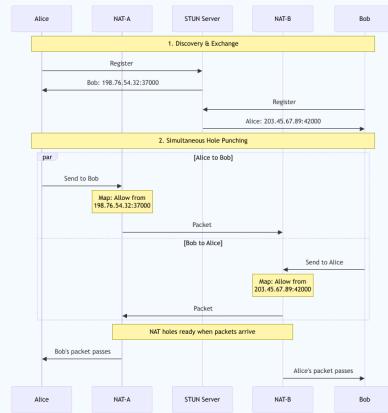
#### TURN:

- Servidor relay intermedio
- Más confiable pero más recursos

#### UPnP:



## Hole punching



- Descubrimiento: servidor STUN para obtener sus IPs/puertos
- Envío simultáneo: Envían paquetes UDP al **mismo tiempo** -> mappings en sus NATs
- Agujeros listos: Los mappings NAT se crean **ANTES** de recibir



## Resumen



## Puntos Clave

- La **Capa de Red** proporciona comunicación end-to-end entre dispositivos en diferentes redes
- **Dos funciones principales:** Enrutamiento (global) y Reenvío (local)
- **Modelos de servicio:** Circuitos Virtuales vs Datagramas (Internet usa datagramas)
- **IPv4:** 32 bits, sistema de clases → CIDR para eficiencia
- **IPv6:** 128 bits, soluciona limitaciones de IPv4
- **ICMP:** Herramientas de diagnóstico (ping, traceroute)
- **NAT:** Permite compartir IP pública, pero limita conectividad directa
- **MTU:** Define tamaño máximo, fragmentación si se excede
- Los **routers** operan con plano de control (enrutamiento) y plano de datos (reenvío)



## 16.5. Capa de Transporte

## Capa de Transporte

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez    Natalia Madrueño  
ruben.rodriguez@urjc.es    natalia.madrueño@urjc.es  
URJC                          URJC

2025-09-09



## Tabla de contenidos

- [Introducción](#)
- [Funciones Principales](#)
- [UDP \(User Datagram Protocol\)](#)
- [TCP \(Transmission Control Protocol\)](#)
- [Comparativa TCP vs UDP para Videojuegos](#)
- [Resumen](#)



## Introducción



## ¿Qué es la Capa de Transporte?

La capa de transporte proporciona comunicación lógica entre procesos de aplicación que se ejecutan en diferentes hosts

### Función principal:

- Se ejecuta en hosts finales
- No en el núcleo de la red
- Divide mensajes en segmentos
- Recomponen segmentos en el receptor

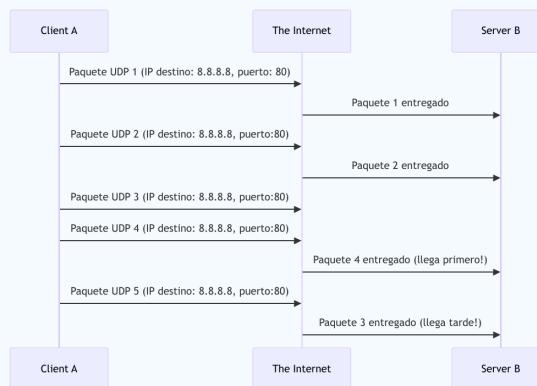


### Protocolos principales:

- **UDP**: Protocolo minimalista
- **TCP**: Protocolo complejo con garantías



## Ejemplo: Cliente-Servidor con UDP



# Funciones Principales



## 1. Multiplexación y Demultiplexación

**Multiplexación:** Recoger información de diferentes sockets y enviarla por un único medio.

**Demultiplexación:** Recibir segmentos y enviarlos a los sockets correspondientes.

### Identificación de sockets:

- TCP: (IP y Puerto origen, IP y puerto destino)
- UDP: (IP y puerto origen, IP y puerto destino)

### Puertos: Identificadores numéricos (1-65535)

- Servidores: asignación manual y fija
- Clientes: asignación aleatoria



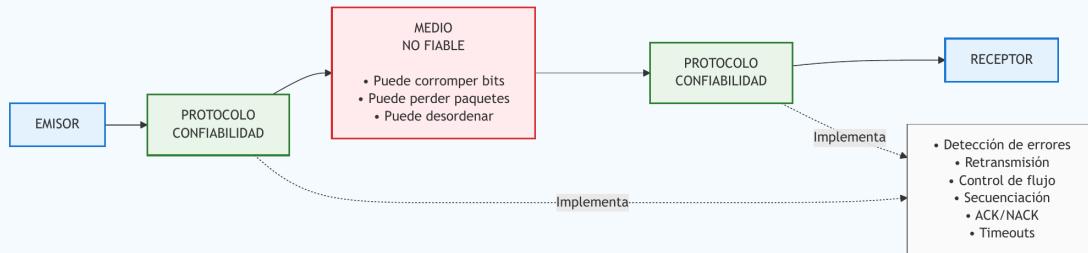
## 2. Transferencia Fiable

### Características de una transferencia fiable:

- No se corrompe ningún bit
- No se pierde información (paquetes)
- La información se entrega en orden correcto

### Opciones de implementación:

1. Usar protocolos fiables existentes (TCP)
2. Implementar características propias sobre protocolo no fiable (UDP + lógica aplicación)



## Otros conceptos

- **Control de flujo:** Evita que el emisor sature al receptor limitando la velocidad de envío según la capacidad del destinatario
- **Control de congestión:** Ajusta la velocidad de transmisión para evitar saturar la red cuando detecta congestión
- **Temporización:** Proporciona el tiempo mínimo de entrega de datos entre emisor y receptor a través de la red
- **Tasa de transferencia mínima:** Garantiza una velocidad mínima de transmisión de datos para aplicaciones que requieren ancho de banda constante



## UDP (User Datagram Protocol)



### Características de UDP

#### Protocolo minimalista [RFC 768]:

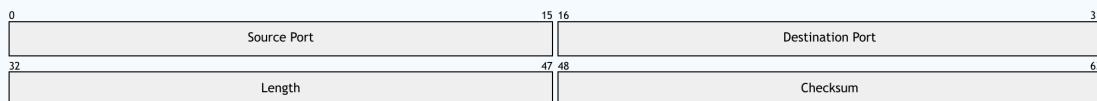
- Basado en best-effort (Fire-and-forget)
- No orientado a conexión
- Entrega no fiable y sin orden
- Integridad básica (checksum)
- Multiplexación y demultiplexación

#### Lo que NO proporciona:

- Control de flujo
- Control de congestión
- Temporización
- Tasa de transferencia mínima
- Seguridad



## Estructura del Paquete UDP



- **Longitud:** Hasta 65535 bytes (limitado por MTU)
- **Checsum:** Verificación de integridad
- Estructura simple comparada con otros protocolos
- 8 bytes de cabecera fijos.



## Checksum UDP

### Proceso de cálculo:

1. Preparación: pseudo-cabecera IP + cabecera UDP + datos
2. División en palabras de 16 bits
3. Suma usando aritmética de complemento a uno
4. Complemento del resultado → campo checksum

### Verificación en receptor:

- Mismo algoritmo incluyendo checksum recibido
- Resultado esperado: 0xFFFF
- Si difiere: datagrama descartado silenciosamente



## Casos de Uso de UDP

Aplicaciones ideales para UDP:

- **Multimedia streaming:** Tolerante a pérdidas, sensible a interrupciones
- **DNS:** Respuestas rápidas necesarias
- **SNMP:** Administración de red
- **Gaming online:** Latencia baja crítica
- **QUIC/HTTP3:** Base para protocolos modernos optimizados

Ejemplo: Implementar protocolo propio sobre UDP para juegos

- Añadir número de paquete
- Descartar paquetes fuera de orden
- Ignorar duplicados
- Sobrecarga mínima



## TCP (Transmission Control Protocol)



## Características de TCP

### Protocolo confiable [RFC 793]:

- Orientado a conexión
- Entrega fiable y ordenada (confiabilidad)
- Control de flujo
- Control de congestión
- Multiplexación y demultiplexación

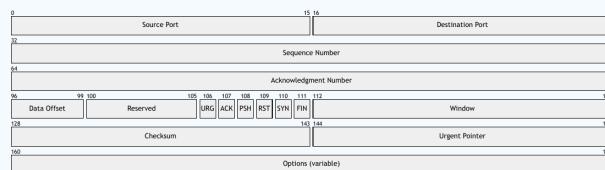
### Lo que NO proporciona:

- Temporización específica
- Tasa mínima garantizada
- Seguridad nativa (necesita TLS/SSL)

**Trade-off:** Confiabilidad y orden sobre velocidad pura



## Estructura del Paquete TCP



- **Sequence number:** Número que identifica la posición del primer byte de datos en el segmento dentro del flujo de datos.
- **Ack number:** Indica el próximo número de secuencia que el receptor espera recibir. Confirma la recepción correcta de datos anteriores.
- **Window:** Implementa control de flujo (bytes que receptor acepta)
- **checksum:** Muy similar a UDP.
- Tamaño variable de 20 a 60 bytes.

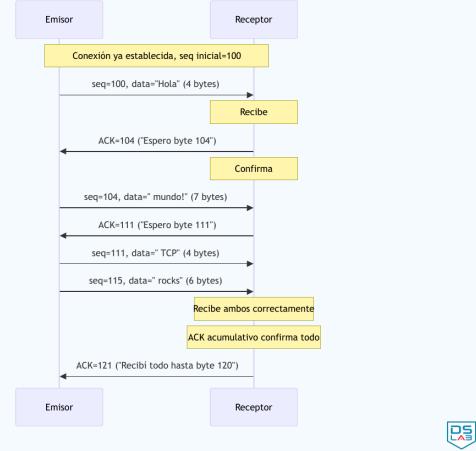


## Mecanismos de Confiabilidad

Los mecanismos de confiabilidad en TCP garantizan que los datos lleguen correctamente y en orden.

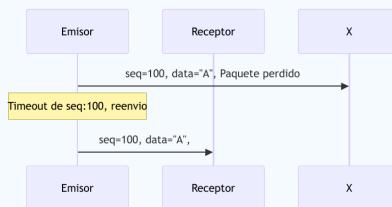
### Números de secuencia y ACKs:

- Cada byte tiene número único
- Cuando enviamos información, está identificada por un número de secuencia (SEQ)
- Además, esperamos confirmación de que se ha recibido correctamente (ACK)
- ACKs acumulativos (ACK para byte N confirma hasta N-1)
- Los ACKs y SEQs permiten detectar datos perdidos, duplicados o desordenados



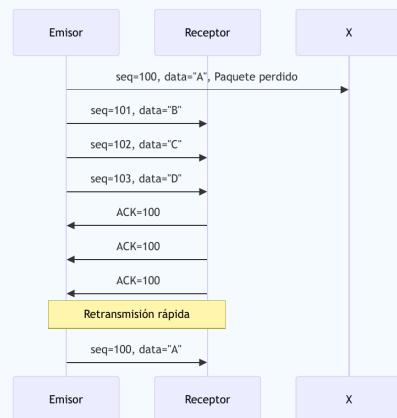
## Detección de pérdidas

### Pérdidas por timeout



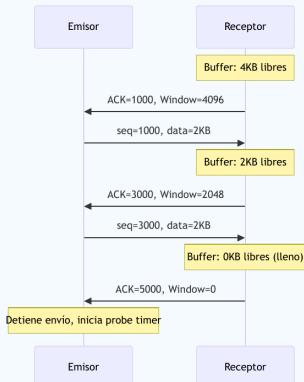
- Si no recibe ACK en tiempo determinado → asume pérdida y retransmite

### Pérdidas por ACKs duplicados



## Control de Flujo

El control de flujo en TCP es un mecanismo que evita que el emisor envíe más datos de los que el receptor puede procesar.



- Receptor informa de su capacidad disponible, y se define:
- $\text{VentanaRecepcion} = \text{BufferRecepcion} - (\text{UltimoByteRecibido} - \text{UltimoByteLeido})$
- El emisor se limita a esta ventana
- Si la capacidad es 0, se espera un tiempo y se vuelve a probar.



## Control de Congestión

El control de congestión en TCP es un mecanismo que ajusta automáticamente la velocidad de envío (ventana de congestión) para evitar saturar la red cuando detecta pérdida de paquetes o retardos.

### Ventana de congestión

- Variable del emisor
- Bytes máximos en “el aire” (enviados sin ACK)
- Tasa efectiva =  $\min(\text{VentanaCongestion}, \text{VentanaRecepcion})$

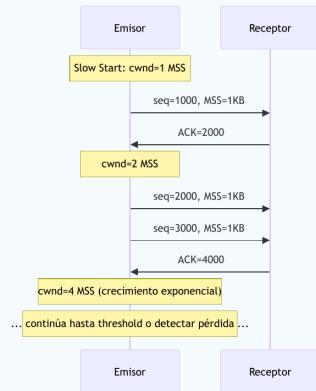
### Eventos de Congestión (Pérdidas)

- **Timeout -> Modo slow start**
  - Pérdida severa
  - Ventana → 1 MSS
- **3 ACKs duplicados -> Modo congestion avoidance**
  - Pérdida moderada
  - Ventana → mitad



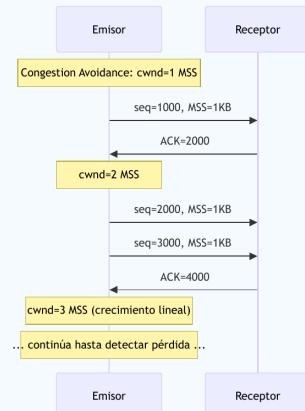
## Mecanismos de control de congestión

### Slow start



Duplicar ventana por cada RTT

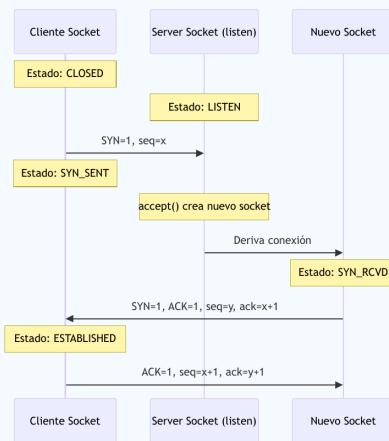
### Congestion Avoidance



Incrementar en 1 la ventana en cada RTT



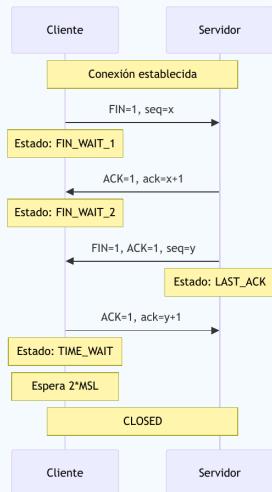
## Handshake de Tres Fases



- Se negocian: MSS, opciones de ventana, extensiones TCP
- Los flags de las cabeceras también consumen bits.
- El serverSocket está en el servidor,
- una vez se establece la comunicación ambos sockets son iguales.



## Terminación de Conexión



- Cada extremo debe enviar su propio FIN y recibir confirmación,
- permitiendo cierre unidireccional (half-close).
- El cliente espera  $2^*MSL$  para asegurar que su último ACK llegó,
- manejando retransmisiones tardías antes del cierre definitivo.



## Equidad y Coexistencia

TCP es “fair”:

- N conexiones TCP comparten enlace equitativamente
- Cada una obtiene  $\sim R/N$  del ancho de banda R
- Ver ejemplos en [jergames](#) (Bandwidth distribution)

Limitaciones:

- UDP no implementa control → puede monopolizar
- Aplicaciones con múltiples conexiones TCP
- Conexiones con menor RTT tienen ventaja



## Comparativa TCP vs UDP para Videojuegos



### Cuándo usar...

#### UDP

##### Requisitos para UDP:

- Latencias < 50ms
- Actualizaciones frecuentes
- Información nueva más valiosa que la vieja

##### Ventajas:

- Cabeceras pequeñas
- Sin tráfico de control
- Servidor necesita menos recursos
- No mantiene estado

**Ejemplos:** Shooters (Counter Strike), juegos de lucha

#### TCP

##### Requisitos para TCP:

- Tolerancia 100-200ms latencia
- Entrega ordenada garantizada
- Detección y corrección de errores

##### Consideraciones:

- Bloqueo cabeza de línea
- Latencias variables por retransmisiones
- Mayor tráfico de red

**Ejemplos:** MMORPGs (World of Warcraft), juegos



## Ejemplos Concretos

### World of Warcraft (TCP):

- Hechizos necesitan entrega garantizada
- Actualizaciones de inventario críticas
- Estado de misiones consistente
- MMORPGs toleran 100-200ms

### Counter Strike (UDP):

- Retroalimentación inmediata crítica
- Actualizaciones posición/disparos
- Técnicas de interpolación en cliente
- Mitiga efecto paquetes perdidos



## Resumen



## Puntos Clave

- La capa de transporte proporciona comunicación lógica entre **procesos**
  - **UDP:** Minimalista, best-effort, no orientado a conexión
  - **TCP:** Confiable, ordenado, orientado a conexión
  - **Multiplexación:** Múltiples sockets por un medio
  - **Control de flujo:** Evita saturar al receptor
  - **Control de congestión:** Responde a condiciones de red
  - **Trade-off fundamental:** Confiabilidad vs velocidad
  - Elección protocolo depende de requisitos aplicación



## 16.6. Capa de Aplicación

# Capa de Aplicación

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez      ruben.rodriguez@urjc.es  
Natalia Madrueño      natalia.madrueno@urjc.es

2025-09-09



## Tabla de contenidos

- [Introducción](#)
- [Sockets](#)
- [Arquitecturas de Aplicaciones Distribuidas](#)
- [Protocolos de Aplicación](#)
- [Servicios](#)
- [Resumen](#)



## Introducción



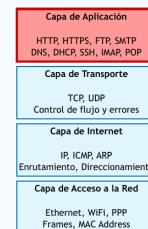
## ¿Qué es la Capa de Aplicación?

La capa de aplicación define los **protocolos que utilizarán las aplicaciones para intercambiar datos**

### ¿Qué hace?

- Define protocolos para intercambio de datos
- Se centra en comunicación entre procesos
- Permite crear protocolos propios
- Opera sobre la capa de transporte

### Posición en el modelo TCP/IP:



**Concepto clave:** Podemos crear nuestros propios protocolos que se ejecuten a nivel de capa de aplicación



## Ejemplo: Protocolo Echo

### Servidor Echo (JavaScript)

```

1 const net = require('net');
2
3 function echoServer() {
4     const server = net.createServer();
5
6     server.on('connection', (socket) => {
7         const clientAddress = `${socket.remoteAddress}:${socket.remotePort}`;
8         console.log(`Client connected: ${clientAddress}`);
9         handleClient(socket, clientAddress);
10    });
11
12    server.listen(8888, () => {
13        console.log('Echo server listening on localhost:8888');
14    });
15 }
16
17 function handleClient(socket, clientAddress) {
18     socket.on('data', (data) => {
19         const message = data.toString('utf-8').trim();
20         if (message.toLowerCase() === 'quit') {
21             socket.end();
22             return;
23         }
24         socket.write(`Echo: ${message}`);
25     });
  
```



## Ejemplo: Cliente Echo

### Cliente Echo (Python)

```

1 import socket
2
3 def echo_client():
4     """Interactive echo client"""
5
6     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7     client_socket.connect(('localhost', 8888))
8
9     while True:
10         message = input("Enter message: ")
11
12         if message.lower() == 'quit':
13             client_socket.send(message.encode('utf-8'))
14             break
15
16         client_socket.send(message.encode('utf-8'))
17         response = client_socket.recv(1024).decode('utf-8')
18         print(f"Server response: {response}")
19
20     client_socket.close()

```



La comunicación puede ser entre procesos en diferentes máquinas e independiente del lenguaje de programación



## Conceptos Clave

- **Protocolos de capa de aplicación:** Definen cómo las aplicaciones intercambian datos
- **Arquitectura de aplicaciones en red:** Cliente-servidor, P2P, híbrida
- **Sockets:** Interfaz entre capa de aplicación y capa de transporte



# Sockets



## ¿Qué son los Sockets?

Los sockets son la **interfaz de programación** que permite a las aplicaciones comunicarse con la capa de transporte

### Características:

- Punto de conexión bidireccional
- Abstracción de detalles de bajo nivel
- API introducida en BSD4.1 UNIX (1981)
- Basada en paradigma cliente/servidor

### Identificación de procesos:

Para identificar un proceso necesitamos:

- **IP del host:** Dirección única (32 bits IPv4)
- **Número de puerto:** Asociado al proceso

Ejemplos de puertos:

- HTTP: 80
- HTTPS: 443
- DNS: 53



## Sockets TCP

### Características principales

#### Propiedades:

- Orientado a conexión
- Confiabilidad garantizada
- Control de flujo
- Control de congestión
- Full-duplex

#### Proceso:

1. Establecer conexión
2. Intercambiar datos
3. Cerrar conexión

*Requiere conexión explícita antes del intercambio*



## Creación de Servidor TCP

### Paso 1: Crear y escuchar

```

1 const net = require('net');
2
3 // Crear servidor TCP
4 const server = net.createServer();
5
6 // Configurar el servidor para escuchar en puerto 8888
7 server.listen(8888, 'localhost', () => {
8   console.log('Servidor TCP escuchando en localhost:8888');
9 });

```

### Paso 2: Manejar conexiones

```

1 // Manejar nuevas conexiones
2 server.on('connection', (socket) => {
3   console.log('Cliente conectado:', socket.remoteAddress);
4
5   // Manejar datos recibidos
6   socket.on('data', (data) => {
7     // Procesar datos
8   });
9
10  // Manejar cierre de conexión
11  socket.on('close', () => {
12    console.log('Cliente desconectado');
13  });
14 });

```



## Cliente TCP

### Establecer conexión y comunicar

```

1 const net = require('net');
2
3 // Crear socket TCP
4 const socket = new net.Socket();
5
6 // Conectar al servidor
7 socket.connect(8888, 'localhost', () => {
8   console.log('Conectado al servidor TCP');
9 });
10
11 // Enviar datos
12 socket.write('Hola servidor');
13
14 // Recibir respuesta
15 socket.on('data', (data) => {
16   console.log('Respuesta:', data.toString());
17 });
18
19 // Cerrar conexión
20 socket.close();

```



## Sockets UDP

### Características principales

#### Propiedades:

- Sin conexión
- Mejor esfuerzo
- Baja latencia
- Simplicidad
- Broadcast/Multicast nativo

#### Ventajas:

- Menor overhead que TCP
- Ideal para tiempo real
- No requiere establecer conexión

#### Desventajas:

- No garantiza entrega
- No garantiza orden



## Servidor y Cliente UDP

### Servidor UDP

```

1 const dgram = require('dgram');
2 const server = dgram.createSocket('udp4');
3
4 server.bind(8888, 'localhost', () => {
5   console.log('Servidor UDP escuchando en localhost:8888');
6 });
7
8 server.on('message', (msg, rinfo) => {
9   console.log(`Mensaje de ${rinfo.address}:${rinfo.port}`);
10  // Responder al cliente
11  server.send('Respuesta', rinfo.port, rinfo.address);
12 });

```

### Cliente UDP

```

1 const dgram = require('dgram');
2 const client = dgram.createSocket('udp4');
3
4 client.send('Hola servidor UDP', 8888, 'localhost', (err) => {
5   if (err) throw err;
6   console.log('Mensaje enviado');
7 });
8
9 client.on('message', (msg, rinfo) => {
10  console.log(`Respuesta recibida: ${msg.toString()}`);
11 });

```



## Servicios Requeridos por Aplicaciones

### Requisitos de las aplicaciones de red

Aplicación	Confiabilidad	Temporización	Ancho de Banda	Seguridad	Protocolo
Transferencia archivos	Sí	No crítica	Elástica	Según contenido	TCP
Correo electrónico	Sí	No crítica	Elástica	Sí	TCP
Navegación web	Sí	Moderada	Elástica	Sí (HTTPS)	UDP / TCP
Streaming video	Tolerante	Crítica	Mínima garantizada	Según contenido	UDP/TCP
Juegos tiempo real	Tolerante	Muy crítica	Moderada	Sí	UDP
Videoconferencia	Tolerante	Crítica	Mínima garantizada	Sí	UDP/TCP



HTTP/3 utiliza QUIC sobre UDP, añadiendo confiabilidad en la capa de aplicación



# Arquitecturas de Aplicaciones Distribuidas



## Tipos de Arquitecturas

Las arquitecturas indican **cómo se conectan los nodos** y **cuál es el rol de cada uno**

### Cliente-Servidor

- Servidor siempre activo
- IP fija conocida
- Clientes no se comunican entre sí
- Centralización de recursos

### Peer-to-Peer

- Nodos se conectan entre sí
- Sin servidor central
- Funcionalidad distribuida
- Ejemplo: BitTorrent

### Híbrida

- Mezcla de ambas
- Autoridades centrales
- Funcionalidades distribuidas
- Más común que P2P puro



## Arquitectura Cliente-Servidor

### Características fundamentales

#### Modelo de funcionamiento:

1. Cliente inicia comunicación
2. Servidor procesa petición
3. Servidor envía respuesta
4. Cliente procesa respuesta

#### Ventajas:

- Centralización de recursos
- Facilita mantenimiento
- Mayor seguridad
- Consistencia del sistema

#### Requerimientos del servidor:

- Dirección IP fija
- Alta disponibilidad
- Capacidad de múltiples conexiones
- Centros de datos
- Balanceamiento de carga
- Redundancia

#### Ejemplos:



## Cliente-Servidor en Videojuegos

### Implementación en juegos multijugador

#### Arquitectura típica:

- Servidor mantiene estado autoritativo
- Clientes manejan presentación visual
- Servidor valida todas las acciones
- Prevención de trampas centralizada

#### Ejemplos:

- World of Warcraft
- Counter-Strike: GO
- League of Legends
- Fortnite Battle Royale

#### Problemas comunes:

- **Latencia/Lag:** Tiempo de procesamiento
- **Sincronización:** Orden de acciones
- **Servidores sobrecargados:** Lanzamientos
- **Pérdida de conexión:** Penalizaciones
- **Costos de infraestructura:** Millones en servidores

*Soluciones: Predicción cliente, interpolación, CDNs*



## Arquitectura Peer-to-Peer

### Funcionamiento y características

#### Principios:

- Cada peer es cliente y servidor
- Sin entidad central
- Autoescalable
- Recursos compartidos
- Unión/salida libre

#### Clasificación por pureza:

- Centralizados (Napster, BitTorrent)
- Descentralizados (Freenet, Gnutella)

#### Aplicaciones comunes:

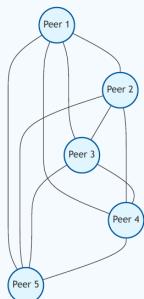
- BitTorrent (archivos)
- Bitcoin (criptomonedas)
- IPFS (contenido distribuido)
- Skype original (VoIP)
- Tox, Briar (mensajería)

#### En videojuegos:

- Juegos de lucha (Street Fighter 6)
- Cooperativos (Portal 2, It Takes Two)



## Topologías P2P



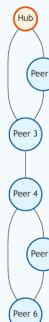
**Full Mesh:** Máxima redundancia, no escalable



**Ring:** Eficiente, vulnerable a fallos



**Star:** Pseudo-P2P, punto único de falla



**Hybrid:** Combina ventajas de diferentes topologías



# Protocolos de Aplicación



## HTTP - HyperText Transfer Protocol

### Fundamentos

#### Características:

- Protocolo para transferencia en WWW
- Texto legible en comandos y respuestas
- Puerto 80 (HTTP) / 443 (HTTPS)
- Modelo cliente-servidor
- Sin estado (stateless)

#### URL estructura:

<https://www.ejemplo.com/pagina.html>

#### Verbos HTTP:

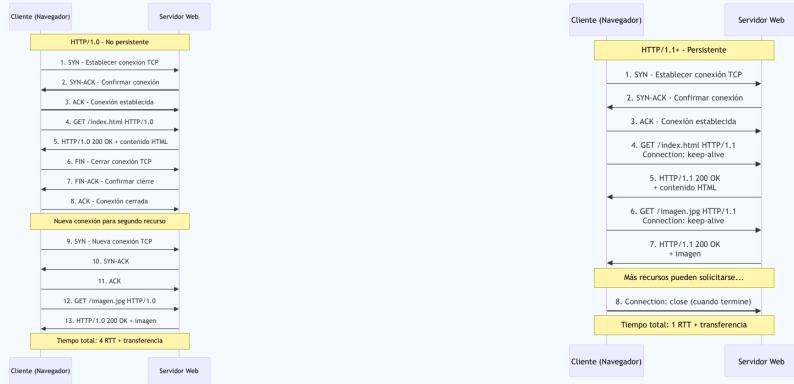
- **GET**: Obtener recurso (idempotente)
- **POST**: Enviar datos (cambia estado)
- **HEAD**: Como GET sin cuerpo
- **PUT**: Cargar objeto (idempotente)
- **DELETE**: Borrar recurso

#### Códigos de respuesta:



## Evolución de HTTP

### Conexiones persistentes vs no persistentes



## Cookies HTTP

### Mecanismo de estado en protocolo sin estado

#### Funcionamiento:

- Pares clave-valor en cliente
- Se configuran en respuesta HTTP
- Fecha de expiración
- Dominio del servidor

#### Usos principales:

- Mantener sesiones
- Personalización
- Análisis de uso
- Publicidad dirigida

#### Tipos:

- **Propias:** De la web navegada
- **Terceros:** Servicios externos
- **Permanentes:** Sin expiración
- **Sesión:** Expiran al cerrar

#### Seguridad:

- Dominio específico
- Evitar suplantaciones
- HTTPS only cookies



## DNS - Domain Name System

### Sistema de nombres de dominio

#### Objetivo:

Traducir nombres a direcciones IP - www.google.es  
→ 142.250.200.67

#### Jerarquía de servidores:

1. **Servidores raíz:** 13 lógicos (A-M)
2. **Servidores TLD:** .com, .org, .es
3. **Servidores autoritativos:** Info definitiva
4. **Servidores locales:** Recursivos/resolvers

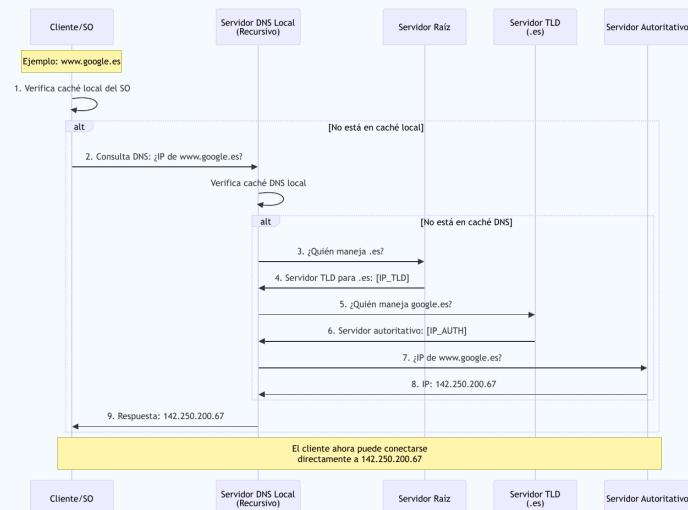
#### Proceso de resolución:

1. Verificar caché local
2. Consulta a DNS local
3. DNS local → Servidor raíz
4. Raíz → Servidor TLD
5. TLD → Servidor autoritativo
6. Autoritativo → IP final
7. Respuesta al cliente

*Sistema distribuido sin servidor central*



## Proceso DNS - Ejemplo



## Protocolos de Correo

### SMTP, IMAP y POP

#### SMTP

**Función:** Envío de correos

- Protocolo “push”
- Transporta mensajes
- No maneja recepción
- Puerto 25/587

#### POP3

**Función:** Descarga de correos

- Descarga completa
- Elimina del servidor
- Un solo dispositivo
- Puerto 110/995

#### IMAP

**Función:** Acceso sincronizado

- Mensajes en servidor
- Multi-dispositivo
- Carpetas y etiquetas
- Puerto 143/993



## QUIC

### Protocolo moderno sobre UDP

#### Ventajas principales:

- Multiplexado sin head-of-line blocking
- Establecimiento 0-RTT
- Migración de conexión transparente
- Control de congestión mejorado
- Forward Error Correction

#### Desarrollado por:

- Google (2012)
- Estandarizado IETF (2021)
- RFC 9000

#### Adopción 2025:

- 8.2% de sitios web usan QUIC
- 31.1% usan HTTP/3
- YouTube reduce 30% tiempo de carga

#### Casos de uso:

- Streaming
- Videoconferencia
- Juegos en línea
- Plataformas de contenido



# Servicios



## CDNs - Content Delivery Networks

### Funcionamiento y beneficios

#### ¿Cómo funcionan?

- Red distribuida de servidores edge
- Copias de contenido cerca del usuario
- Enrutamiento inteligente automático
- Reduce latencia: 200-500ms → <50ms

#### Estrategias de caché:

- **Estático:** Días/semanas (imágenes, videos)
- **Dinámico:** Minutos/horas (APIs)
- **Personalizado:** Cache parcial
- **Streaming:** Segmentos individuales

#### Servicios adicionales:

- Compresión automática
- Conversión de formatos
- Balanceo de carga
- Protección DDoS
- Pre-carga predictiva
- Ejecución edge computing

#### En videojuegos:



## Servidores Proxy

### Intermediarios inteligentes

#### Funcionamiento:

1. Cliente envía petición a proxy
2. Proxy analiza petición
3. Si puede resolver → responde
4. Si no → consulta servidor origen
5. Cachea respuesta
6. Envía al cliente

#### Ventajas:

- Navegación más rápida
- Reduce tráfico de red
- Seguridad adicional
- Anonimato
- Control de acceso

#### GET condicional:

- Solo devuelve si hay cambios
- Ahorra ancho de banda
- Reduce tiempo de respuesta



#### Ubicación típica:

## Resumen



## Puntos Clave

- La **Capa de Aplicación** define protocolos para intercambio de datos entre procesos
- **Sockets**: Interfaz entre aplicación y transporte (TCP confiable vs UDP rápido)
- **Arquitecturas**: Cliente-servidor (centralizado), P2P (distribuido), Híbrida
- **HTTP**: Protocolo web sin estado, evolución de 1.0 a HTTP/3 sobre QUIC
- **DNS**: Sistema distribuido jerárquico para traducir nombres a IPs
- **Correo**: SMTP (envío), POP (descarga), IMAP (sincronización)
- **QUIC**: Protocolo moderno sobre UDP con ventajas de TCP + TLS
- **CDNs**: Redes de distribución que acercan contenido a usuarios
- Podemos crear **protocolos propios** en esta capa



## 16.7. JavaScript

### Desarrollo en el cliente

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez    Natalia Madrueño  
ruben.rodriguez@urjc.es    natalia.madrueño@urjc.es  
URJC                            URJC

2025-09-09



## Tabla de contenidos

- [Introducción a JavaScript](#)
- [Configuración del Entorno](#)
- [El Lenguaje JavaScript](#)
- [Arrays](#)
- [Control de Flujo](#)
- [Funciones](#)
- [Manejo de Excepciones](#)
- [Almacenamiento de Datos](#)



# Introducción a JavaScript



## Introducción

JavaScript es el lenguaje fundamental para desarrollo web interactivo y videojuegos web.

### Características principales:

- Scripting (no necesita compilador)
- Tipado dinámico
- Funcional
- Orientado a objetos (prototipos)

### Aplicaciones:

- Interactividad en páginas web
- Modificación del DOM
- Peticiones AJAX
- Videojuegos web



## Versiones de ECMAScript

- Primera versión en 10 días (1995).
- **ES5 (2011)**: Base sólida que estableció JavaScript moderno en todos los navegadores
- **ES6/ES2015**: Revolución del lenguaje - introdujo sintaxis moderna que cambió cómo programamos
- **Actualizaciones anuales**: Cada año se añaden mejoras sin romper código existente (compatibilidad hacia atrás)
- **ES2015 marcó un antes y después**: La mayoría del código moderno usa características de ES6+

Versión	Año	Características
ES5	2011	Popularizó JavaScript
ES2015 (ES6)	2015	Clases, módulos, arrow functions, promises
ES2016-2024	2016-2024	Actualizaciones anuales con compatibilidad



## JavaScript vs Java

### ¡Importante!

Aunque la sintaxis recuerda a Java, son lenguajes **completamente diferentes**.

- **El nombre “JavaScript” fue puro marketing:** Se eligió para aprovechar la popularidad de Java
- **Inicialmente se llamó LiveScript:** Cambió de nombre antes de su lanzamiento oficial
- **Java estaba en auge** cuando se publicó JavaScript (1995)
- **Diferentes propósitos:** Java para aplicaciones robustas, JavaScript para web interactivo



## DOM y BOM

### DOM (Document Object Model)

- **Representa HTML como árbol de objetos:** Cada etiqueta HTML es un objeto manipulable
- **Modificación dinámica:** Cambia contenido sin recargar la página completa
- **Gestión de eventos:** Captura clicks, teclas, movimientos del ratón, etc.
- **Ejemplo:** `document.getElementById('boton')` accede a un elemento

### BOM (Browser Object Model)

- **Controla el navegador entero:** No solo el documento, sino ventanas, historial, URL
- **window.location:** Navega a otras URLs o recarga la página
- **window.history:** Retrocede/avanza en el historial del navegador
- **window.localStorage:** Guarda datos en el navegador
- **El BOM contiene al DOM:** `window.document` es el DOM



## Librerías JavaScript

### Para videojuegos:

- Phaser: Framework completo para juegos 2D - maneja física, colisiones, animaciones

### Herramientas de desarrollo:

- Webpack: Empaqueador de módulos - combina y optimiza tu código JavaScript y assets
- Babel: Transpilador - convierte código JavaScript moderno a versiones compatibles con navegadores antiguos

### Backend y comunicación:

- Express.js: Framework minimalista para crear servidores web y APIs en Node.js
- WS (ws.js): Biblioteca para WebSockets - permite comunicación en tiempo real bidireccional



## Configuración del Entorno



## Node.js

- **Ejecuta JavaScript fuera del navegador:** En tu ordenador, servidores, etc.
- **npm (Node Package Manager):** Gestor de paquetes para instalar librerías
- **Entorno de desarrollo moderno:** Necesario para usar herramientas como Webpack

Node.js permite ejecutar JavaScript fuera del navegador.

### Instalación:

1. Descargar desde [nodejs.org](https://nodejs.org)
2. Instalar versión LTS (Long Term Support - más estable)

### Verificar instalación:

```
1 # Muestra la versión de Node.js instalada
2 node --version
3
4 # Muestra la versión de npm (viene incluido con Node)
5 npm --version
```



## Iniciar un Proyecto

- **npm init -y:** Crea `package.json` con configuración por defecto (sin preguntas)
- **package.json:** Archivo que describe tu proyecto y sus dependencias
- **src/:** Carpeta con tu código fuente JavaScript
- **dist/:** Carpeta con archivos finales optimizados listos para producción
- **public/:** Recursos estáticos como HTML, imágenes, CSS que no se procesan

```
1 # Inicializar proyecto npm (crea package.json) dentro de una carpeta
2 npm init -y
```



### Estructura recomendada:

```
mi-juego-web/
├── package.json      # Configuración del proyecto
├── webpack.config.js # Configuración de Webpack
└── src/               # Tu código fuente
    └── dist/           # Archivos compilados (generado)
        └── public/       # HTML y recursos estáticos
            └── index.html
```

## Instalación de Dependencias

- **Dependencias de producción:** Librerías que necesita tu juego para funcionar
- **Dependencias de desarrollo:** Herramientas solo para programar (no van al juego final)
- **npm install:** Descarga e instala paquetes desde el registro de npm
- **-save-dev:** Marca como dependencia de desarrollo

### Dependencias de producción:

```
1 # Phaser: motor de juegos 2D
2 # Lodash: utilidades para datos
3 # Axios: peticiones HTTP
4 npm install phaser lodash axios
```

### Dependencias de desarrollo:

```
1 # Webpack: empaqueta y optimiza código
2 npm install --save-dev webpack webpack-cli webpack-dev-server
3
4 # Plugins de Webpack para HTML y CSS
5 npm install --save-dev html-webpack-plugin css-loader style-loader
6
7 # Herramientas de calidad de código
8 npm install --save-dev eslint prettier
```



## Configuración de Webpack

- **Webpack:** Empaquetador que une todos tus archivos JS en uno solo optimizado
- **entry:** Punto de entrada - primer archivo que se ejecuta
- **output:** Dónde guardar el resultado final
- **plugins:** Extensiones que añaden funcionalidades (ej: generar HTML)
- **devServer:** Servidor de desarrollo con recarga automática

```
1 const path = require('path');
2 const HtmlWebpackPlugin = require('html-webpack-plugin');
3
4 module.exports = {
5   // **entry**: Punto de entrada de la aplicación
6   entry: './src/index.js',
7
8   // **output**: Dónde se guarda el bundle generado
9   output: {
10     path: path.resolve(__dirname, 'dist'),
11     filename: 'bundle.js',
12     clean: true // Limpia la carpeta 'dist' antes de cada build
13   },
14
15   // **plugins**: Añade funcionalidades extra (como generar el HTML automáticamente)
16   plugins: [
17     new HtmlWebpackPlugin({
18       template: './public/index.html', // Usa esta plantilla HTML
19     })
20   ]
21 }
```



## Scripts de Desarrollo

- **Scripts npm:** Comandos personalizados para automatizar tareas comunes
- **npm run dev:** Inicia servidor de desarrollo con recarga automática
- **npm run build:** Crea versión optimizada para producción
- **npm run lint:** Revisa errores de código con ESLint
- **npm run format:** Formatea código con Prettier

**package.json:**

```
1 {
2   "scripts": {
3     // Construye el paquete para producción usando Webpack
4     "build": "webpack --mode production",
5
6     // Inicia el servidor de desarrollo con Webpack Dev Server y habilita hot-reload
7     "dev": "webpack serve --mode development",
8
9     // Inicia el servidor en modo producción (requiere que server.js esté correctamente configurado)
10    "start": "node server.js",
11
12    // Observa los cambios en los archivos y recompila automáticamente (sin servidor)
13    "watch": "webpack --watch",
```



## El Lenguaje JavaScript



## Características del Lenguaje

- **Imperativo y Estructurado:** Escribe instrucciones paso a paso, como Java o C
- **Lenguaje de Script:** El navegador ejecuta el código directamente, sin compilar primero
- **Tipado Dinámico:** Las variables pueden cambiar de tipo (`let x = 5; x = "texto";`)
- **Orientado a Objetos:** Usa prototipos en lugar de clases tradicionales (hasta ES6)
- **Funcional:** Las funciones son valores - pueden pasarse como argumentos



## Modo Estricto

- **Detecta errores que normalmente JavaScript ignora:** Convierte errores silenciosos en excepciones
- **Prohibe sintaxis peligrosa:** Variables sin declarar, duplicar parámetros, etc.
- **Mejor rendimiento:** Permite optimizaciones del motor JavaScript
- **En módulos ES2015+ ya está activo:** No necesitas añadirlo manualmente
- También se puede utilizar `node --use_strict` en node.

```

1 // Activar modo estricto (poner al inicio del archivo o función)
2 "use strict";
3
4 // Ahora esto genera error (sin strict mode, crea variable global)
5 playerName = "Juan"; // Error: playerName is not defined

```



## Integración con HTML

- **Scripts en `<head>`**: Se ejecutan antes de cargar el contenido (puede bloquear renderizado)
- **Scripts al final de `<body>`** : Mejor práctica - el HTML ya está cargado
- **`async`**: Descarga el script en paralelo, ejecuta en cuanto esté listo (orden no garantizado)
- **`defer`**: Descarga en paralelo, pero ejecuta en orden después del HTML

```

1 <html>
2 <head>
3   <!-- Script en head - se ejecuta inmediatamente -->
4   <script src="js/config.js"></script>
5 </head>
6 <body>
7   <!-- Contenido HTML -->
8   <!-- Scripts al final - mejor rendimiento (HTML ya cargado) -->
9   <script src="js/game.js"></script>
10  <!-- async: descarga paralela, ejecuta inmediatamente -->
11  <script src="js/game.js" async></script>
12
13  <!-- defer: descarga paralela, ejecuta después del HTML -->
14  <script src="js/game.js" defer></script>
15
16 </body>
17 </html>

```



## Mostrar Información

- **`document.write()`**: Escribe directamente en el HTML (evitar - obsoleto)
- **`console.log()`**: Muestra información en consola del navegador (F12)
- **`console.error()`**: Muestra errores en rojo - útil para debugging
- **`console.warn()`**: Muestra advertencias en amarillo

```

1 // Escribir en el documento HTML (no recomendado)
2 document.write('Texto');
3
4 // Consola del navegador (debugging) - Pulsa F12 para verla
5 console.log('Información de debug');
6 console.error('Error crítico');
7 console.warn('Advertencia');
8
9 // También puedes mostrar objetos
10 console.log('Jugador:', { nombre: 'Juan', vida: 100 });

```



## Comentarios

- Igual que en Java.

```

1 // Comentario de una línea - se ignora al ejecutar
2
3 /*
4  * Comentario
5  * multilinea
6  * útil para documentar funciones
7 */
8
9 // Los comentarios explican el "por qué", no el "qué"
10 // Malo: let x = 5; // Asigna 5 a x
11 // Bueno: let maxRetries = 5; // Límite de reintentos antes de fallar

```



## Variables

- **const**: Usa por defecto - evita modificaciones accidentales de valores que no deben cambiar
- **let**: Solo cuando necesites reasignar el valor (contadores, acumuladores)
- **var**: Obsoleto - tiene problemas de scope que causan bugs difíciles de encontrar
- **Block scope**: Variables solo existen dentro del bloque `{}` donde se declaran

```

1 // const - valor inmutable (no se puede reasignar), block scope
2 const MAX_LIVES = 3;
3 const GAME_CONFIG = {
4   width: 800,
5   height: 600
6 };
7
8 // let - valor mutable (se puede reasignar), block scope
9 let currentLives = MAX_LIVES;
10 let playerPosition = { x: 0, y: 0 };
11
12 // var - evitar en código nuevo (function scope - problemas)
13 var oldStyle = "no recomendado";
14
15 // Ejemplos de uso
16 currentLives = 2; // OK - let permite reasignar
17 MAX_LIVES = 5; // ERROR - const no permite reasignar

```



## Ámbito de Variables

- **Block scope** (`let`, `const`): Variable solo existe dentro de las llaves `{}` donde se declaró
- **Function scope** (`var`): Variable existe en toda la función, ignorando bloques `{}`
- **Variables sin declarar**: Se vuelven globales (accesibles desde cualquier lugar) - PELIGROSO
- **Modo estricto**: Convierte variables sin declarar en error

```

1 function ejemploScope() {
2   if (true) {
3     let bloqueVariable = "solo aquí";
4     var funcionVariable = "toda la función";
5   }
6
7   // console.log(bloqueVariable); // ERROR - no existe fuera del if
8   console.log(funcionVariable); // OK - var tiene function scope
9 }
10
11 // Sin modo estricto
12 function peligro() {
13   sinDeclarar = "ups"; // Crea variable global - BAD!
14 }
```



## Tipos de Datos Primitivos

- **Number**: Un solo tipo para enteros y decimales (64 bits de precisión)
- **String**: Cadenas de texto - pueden usar `""`, `''` o ```` (template literals)
- **Boolean**: Solo dos valores: `true` o `false`
- **null**: Ausencia intencional de valor - “esto está vacío a propósito”
- **undefined**: Variable declarada pero sin valor asignado - “aún no tiene valor”

```

1 // Number - enteros y decimales (punto flotante de 64 bits)
2 let score = 1000;
3 let health = 75.5;
4 let infinity = Infinity; // Valor especial
5
6 // String - cadenas de caracteres
7 let playerName = "Jugador1";
8
9 // Boolean - verdadero o falso
10 let isGameRunning = true;
11 let isPaused = false;
12
13 // Tipos especiales
14 let powerUp = null;           // Ausencia intencional - "no hay powerup"
15 let specialAbility;          // undefined - no inicializada aún
```



## Template Literals (ES2015+)

- **Backticks ( ` )**: En lugar de comillas normales
- **Interpolación \${}** : Inserta variables o expresiones dentro del string
- **Multilínea**: Puedes escribir strings en varias líneas sin concatenar
- **Expresiones**: Dentro de \${} puedes poner cualquier código JavaScript

```

1 let level = 5;
2 let experience = 1250;
3
4 // Interpolación de strings - inserta valores de variables
5 let status = `Nivel ${level} - EXP: ${experience}`;
6
7 // También puedes usar expresiones dentro de ${}
8 let progress = `Progreso: ${experience / 2000} * 100%`;
9
10 // Strings multilínea - se respetan los saltos de línea
11 let gameInfo = `
12 Jugador: ${playerName}
13 Nivel: ${level}
14 Puntuación: ${score}
15 Estado: ${score > 1000 ? 'Pro' : 'Novato'}
16 `;

```



## Operadores Aritméticos

- **Operadores matemáticos básicos**: + , - , \* , / , % (módulo/resto)
- **Comparación**: > , < , >= , <= comparan valores
- **Lógicos**: && (y), || (o), ! (no) para combinar condiciones
- **Precedencia**: Multiplicación y división antes que suma y resta

```

1 // Operadores aritméticos - similares a Java
2 let damage = baseDamage + bonus;
3 let remaining = total - used;
4 let area = width * height;
5 let average = sum / count;
6 let remainder = value % modulo; // Resto de división
7
8 // Lógicos - combinan condiciones -> Devuelven true or false
9 let canAct = isAlive && !isStunned; // Y lógico, NO lógico
10 let shouldRespawn = isDead || health <= 0; // O lógico

```



## Operadores de Comparación

- **== (estricto)**: Compara valor Y tipo - **siempre recomendado**
- **== (débil)**: Convierte tipos antes de comparar - **evitar** (causa bugs)
- **!= (desigualdad estricta)**: Diferente valor O tipo
- **Ejemplo**: `5 == "5"` es `true`, pero `5 === "5"` es `false`

```

1 // Igualdad estricta (recomendado) – compara valor Y tipo
2 if (playerID === targetID) {
3     // Solo true si ambos tienen el mismo valor Y tipo
4 }
5
6 // Desigualdad estricta – diferente valor O tipo
7 if (level !== previousLevel) {
8     // true si son diferentes
9 }
10
11 // Igualdad débil (EVITAR) – hace conversión de tipos
12 if (score == "100") {
13     // true – convierte "100" a número 100. Puede causar bugs difíciles de encontrar
14 }
15
16 // Comparación de tipos diferentes
17 console.log(5 === "5"); // false – número vs string
18 console.log(5 == "5"); // true – convierte a mismo tipo

```



## Operadores Modernos (ES2020+)

- **Nullish coalescing ( ?? )**: Valor por defecto solo si `null` o `undefined`
- **Optional chaining ( ?. )**: Accede a propiedades sin error si no existen
- **Logical assignment**: Asigna solo si cumple condición
- **Diferencia con || : ||** también considera `0`, `""`, `false` como “falsy”

```

1 // Nullish coalescing – valor por defecto solo para null/undefined
2 let playerName = savedName ?? "Jugador Anónimo";
3 // Si savedName es 0 o "" NO usa el default (solo null/undefined)
4
5 // Optional chaining – evita errores si propiedades no existen
6 let weapon = player.inventory?.equipment?.weapon;
7 // Si inventory o equipment es null/undefined, retorna undefined sin error
8
9 // Logical assignment
10 playerName ||= "Jugador por defecto"; // Asigna solo si falsy
11 playerName ??= "Valor por defecto"; // Asigna solo si null/undefined
12
13 // Diferencia entre || y ???
14 let count = 0;
15 let result1 = count || 10; // 10 (0 es falsy)
16 let result2 = count ?? 10; // 0 (0 no es null/undefined)

```



## Valores Falsy

- JavaScript evalúa algunos valores como `false` en condiciones
- **6 valores falsy:** `false`, `0`, `""`, `null`, `undefined`, `NaN`
- **Todos los demás son truthy:** Incluso `"0"`, `[]`, `{}`
- **Útil para:** Validaciones y valores por defecto

JavaScript considera **falso** (falsy):

- `false` - booleano falso
- `null` - sin valor
- `undefined` - no definido (verificación de existencia)
- `""` - string vacío
- `0` - cero numérico
- `NaN` - “Not a Number”



## Utiliza siempre ===

	<code>true</code>	<code>false</code>	<code>1</code>	<code>0</code>	<code>-1</code>	<code>"true"</code>	<code>"false"</code>	<code>"1"</code>	<code>"0"</code>	<code>"-1"</code>	<code>""</code>	<code>null</code>	<code>undefined</code>	<code>Infinity</code>	<code>-Infinity</code>	<code>[]</code>	<code>{}</code>	<code>[1]</code>	<code>[0]</code>	<code>[1]</code>	<code>NaN</code>
<code>true</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>false</code>	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	
<code>1</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>0</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>-1</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>"true"</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>"false"</code>	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	
<code>"1"</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>"0"</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>"-1"</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>""</code>	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>null</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>undefined</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>Infinity</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>-Infinity</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>[]</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>{}</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>[1]</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>[0]</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>[1]</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	
<code>NaN</code>	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	



# Arrays



## Creación de Arrays

- **Arrays dinámicos:** Pueden crecer o reducirse automáticamente
- **Tipos mezclados:** Puedes guardar diferentes tipos en el mismo array
- **Índice desde 0:** El primer elemento está en posición 0
- **length:** Propiedad que indica cuántos elementos tiene

```

1 // Creación de arrays - literal (forma más común)
2 let empty = [];
3 let numbers = [1, 2, 3, 4, 5];
4 let mixed = ["texto", 42, true, null]; // Tipos mezclados OK
5
6 // Constructor Array (menos común)
7 let inventory = new Array(10); // Crea array con 10 espacios vacíos
8
9 // Acceso y modificación por índice
10 console.log(numbers[0]); // 1 - primer elemento
11 numbers[2] = 999; // Modifica tercer elemento
12
13 // Arrays dinámicos - se expanden automáticamente
14 numbers[10] = 100; // Salta del índice 4 al 10
15 console.log(numbers.length); // 11 - ahora tiene 11 elementos
16 // Los índices 5-9 quedan vacíos (undefined)

```



## Métodos de Array Modernos

- **filter()**: Crea nuevo array con elementos que cumplen condición
- **map()**: Crea nuevo array transformando cada elemento
- **No modifican el original**: Retornan nuevo array (inmutabilidad)
- **Callback**: Función que se ejecuta por cada elemento

```

1 // Array de ejemplo
2 let enemies = [
3   { id: 1, health: 100, type: "orc" },
4   { id: 2, health: 50, type: "goblin" },
5   { id: 3, health: 0, type: "orc" }
6 ];
7
8 // filter() - selecciona elementos que cumplen condición
9 // Retorna nuevo array con enemigos vivos
10 let aliveEnemies = enemies.filter(enemy => enemy.health > 0);
11 // [{ id: 1, ... }, { id: 2, ... }]
12
13 // Solo los orcos
14 let orcs = enemies.filter(enemy => enemy.type === "orc");
15
16 // map() - transforma cada elemento
17 // Extrae solo la salud de cada enemigo
18 let healthValues = enemies.map(enemy => enemy.health);
19 // [100, 50, 0]

```



## Más Métodos de Arrays

- **find()**: Retorna el **primer** elemento que cumple condición
- **findIndex()**: Retorna el **índice** del primer elemento que cumple condición
- **every()**: Retorna **true** si **todos** los elementos cumplen condición
- **some()**: Retorna **true** si **algun** elemento cumple condición
- **reduce()**: Reduce array a un único valor (suma, acumulación, etc.)

```

1 // find() - retorna PRIMER elemento que cumple condición
2 let firstOrc = enemies.find(enemy => enemy.type === "orc"); // { id: 1, health: 100, type: "orc" }
3
4 // findIndex() - retorna índice del primer match
5 let orcIndex = enemies.findIndex(enemy => enemy.type === "orc"); // 0
6
7 // every() - ¿TODOS cumplen la condición?
8 let allDead = enemies.every(enemy => enemy.health === 0); // false -> no todos muerto
9
10 // some() - ¿ALGUNO cumple la condición?
11 let someDead = enemies.some(enemy => enemy.health === 0); // true -> al menos un muerto
12
13 // reduce() - reduce array a un solo valor
14 // Suma total de la salud de todos los enemigos
15 let totalHealth = enemies.reduce((sum, enemy) =>
16   sum + enemy.health, 0); // 0 es valor inicial
17 // 150 (100 + 50 + 0)

```



## Modificar Arrays

- **push()**: Añade elemento al final - modifica el array original
- **pop()**: Elimina y retorna último elemento
- **shift()**: Elimina y retorna primer elemento
- **unshift()**: Añade elemento al principio
- **splice()**: Elimina/inserta elementos en cualquier posición

```

1 // push() - añadir al final (modifica original)
2 enemies.push({ id: 4, health: 75, type: "troll" });
3
4 // shift() - quitar del principio
5 let firstEnemy = enemies.shift();
6
7 // pop() - quitar del final
8 let lastEnemy = enemies.pop();
9
10 // splice(inicio, cantidad) - eliminar elementos
11 enemies.splice(1, 2); // Eliminar 2 elementos desde índice 1
12
13 // splice(inicio, 0, elemento) - insertar sin eliminar
14 enemies.splice(1, 0, newEnemy); // Insertar en índice 1
15
16 // unshift() - añadir al principio
17 enemies.unshift({ id: 0, health: 50, type: "scout" });

```



## Destructuring de Arrays

- **Destructuring**: Extraer valores de arrays a variables individuales
- **Rest operator ( ... )**: Captura “el resto” de elementos
- **Sintaxis limpia**: Evita acceder por índice repetidamente

```

1 // Extraer valores de array a variables
2 let coordinates = [100, 200];
3 let [x, y] = coordinates; // x = 100, y = 200
4
5 // Ignorar elementos
6 let [first, , third] = [1, 2, 3]; // first = 1, third = 3
7
8 // Rest operator - captura el resto de elementos
9 let [first, second, ...rest] = inventory;
10 // first = primer elemento
11 // second = segundo elemento
12 // rest = array con todos los demás
13
14 // En parámetros de función
15 function showCoords([x, y]) {
16   console.log(`X: ${x}, Y: ${y}`);
17 }

```



# Control de Flujo



## Sentencias Básicas

- **if-else:** Ejecuta código según condición verdadera o falsa
- **else if:** Permite múltiples condiciones en cadena
- **Condiciones:** Expresiones que evalúan a `true` o `false`
- **Bloques {}:** Agrupan varias instrucciones

```
1 // if - else - estructura condicional básica
2 if (health > 50) { // Siempre recomendado con llaves
3     // Se ejecuta si la condición es verdadera
4     statusColor = "green";
5 } else if (health > 20) {
6     // Se ejecuta si la primera es falsa y esta es verdadera
7     statusColor = "yellow";
8 } else {
9     // Se ejecuta si todas las anteriores son falsas
10    statusColor = "red";
11 }
12
13 // Sin llaves para una sola instrucción (no recomendado)
14 if (isDead) gameOver();
```



## Switch

- **switch:** Compara una expresión contra múltiples valores
- **case:** Cada posible valor a comparar
- **break:** Sale del switch (sin él, continúa al siguiente case)
- **default:** Se ejecuta si ningún case coincide

```

1 // switch - útil cuando comparas misma variable con muchos valores
2 switch (gameState) {
3     case "menu":
4         // Se ejecuta si gameState === "menu"
5         showMenu();
6         break; // Sale del switch.
7
8     case "rollback": // Como no hay break se ejecuta el siguiente
9         rollback();
10
11    case "playing":
12        updateGame();
13        break;
14
15    default:
16        // Se ejecuta si ningún case coincide
17        handleUnknownState();
18 }
```



## Loops

- **for tradicional:** Cuando sabes cuántas iteraciones necesitas
- **for...of:** Itera sobre valores de un array (ES2015+) - más limpio
- **for...in:** Itera sobre claves/propiedades de un objeto
- **break:** Sale del loop inmediatamente
- **continue:** Salta a la siguiente iteración

```

1 // for tradicional - control total sobre índice
2 for (let i = 0; i < enemies.length; i++) {
3     updateEnemy(enemies[i]);
4     // i = 0, 1, 2, ... hasta length-1
5 }
6
7 // for...of - itera valores (ES2015+) - MÁS LIMPIO
8 for (let enemy of enemies) {
9     updateEnemy(enemy); // enemy es el valor directamente
10 }
11
12 // for...in - itera propiedades/claves (para objetos)
13 for (let key in gameConfig) {
14     console.log(key, gameConfig[key]);
15     // key = "width", "height", etc.
16 }
```



## While

- **while:** Repite mientras la condición sea verdadera
- **Cuidado con loops infinitos:** Asegúrate que la condición eventualmente sea falsa
- **do...while:** Ejecuta al menos una vez, luego verifica condición

```
1 // while - repite mientras la condición sea true
2 while (isGameRunning && playerLives > 0) {
3     processGameFrame();
4     // IMPORTANTE: algo dentro debe cambiar la condición
5     // o será un loop infinito
6 }
7
8 // do...while - ejecuta AL MENOS una vez
9 let input;
10 do {
11     input = prompt("Ingresa comando:");
12 } while (input !== "quit");
```



## Funciones



## Declaración de Funciones

- **function:** Palabra clave para declarar funciones
- **Parámetros:** Valores que recibe la función (entre paréntesis)
- **return:** Devuelve un valor y termina la función
- **Expresión de función:** Asignar función a una variable
- **Hoisting:** Las declaraciones se mueven al inicio (se pueden llamar antes de declarar)

```

1 // Declaración tradicional - se puede llamar antes de declarar
2 function calculateDamage(baseDamage, criticalHit) {
3     // Verifica si hay golpe crítico
4     if (criticalHit) {
5         return baseDamage * 2; // Retorna el doble
6     }
7     return baseDamage; // Retorna daño normal
8 }
9
10 // Uso
11 let damage = calculateDamage(50, true); // 100

```

```

1 // Expresión de función - asignada a variable
2 let heal = function(amount) {
3     player.health += amount; // Suma salud
4     // Limita la salud al máximo
5     if (player.health > player.maxHealth) {
6         player.health = player.maxHealth;
7     }
8 };
9
10 // Uso
11 heal(20);

```



## Arrow Functions (ES2015+)

- **Sintaxis más concisa** que funciones tradicionales
- **No tienen su propio this**: Heredan **this** del contexto (útil en callbacks)
- **Retorno implícito**: Si es una sola expresión, retorna automáticamente (sin **return**)
- **Ideal para**: Callbacks, funciones cortas, map/filter/reduce

```

1 // Arrow function completa con llaves
2 let movePlayer = (deltaX, deltaY) => {
3     player.x += deltaX; // Mueve en X
4     player.y += deltaY; // Mueve en Y
5 };
6
7 // Arrow function con una expresión - retorno implícito
8 let isAlive = (entity) => entity.health > 0;
9 // Equivalente a: function(entity) { return entity.health > 0; }
10
11 // Sin parámetros - paréntesis vacíos
12 let generateRandomID = () => Math.random().toString(36);
13
14 // Un parámetro - paréntesis opcionales
15 let double = x => x * 2;

```



## Parámetros de Función

- **Parámetros por defecto:** Valor asignado si no se pasa argumento
- **Rest parameters ( ... ):** Captura argumentos restantes en un array
- **Destructuring:** Extrae propiedades de objetos directamente en parámetros
- **Orden:** Parámetros normales, luego con default, luego rest

```

1 // Parámetros por defecto (ES2015+)
2 function createEnemy(health = 100, damage = 10) {
3   return { health, damage };
4 }
5 // Si no pasas argumentos, usa los valores por defecto
6 createEnemy(); // { health: 100, damage: 10 }
7 createEnemy(50); // { health: 50, damage: 10 }
8
9 // Rest parameters - captura argumentos restantes en array
10 function logMessage(level, ...messages) {
11   console.log(`[${level}]`, ...messages);
12 }
13 logMessage("ERROR", "Falló", "al cargar", "recurso");
14 // messages = ["Falló", "al cargar", "recurso"]

```



## Closures

- **Closure:** Función que “recuerda” variables de su contexto exterior
- **Encapsulación:** Variables privadas que solo la función puede modificar
- **Estado persistente:** Mantiene estado entre llamadas sin variables globales
- **Útil para:** Contadores, factory functions, datos privados

```

1 // Closure - función que retorna objeto con métodos
2 function createCounter(initialValue = 0) {
3   // Variable privada - sólo accesible dentro
4   let count = initialValue;
5
6   // Retorna objeto con métodos que acceden
7   // a 'count'
8   return {
9     // Incrementa y retorna nuevo valor
10    increment: () => ++count,
11
12    // Decrementa y retorna nuevo valor
13    decrement: () => --count,
14
15    // Retorna valor actual
16    getValue: () => count
17  };
18 }

```

```

1 // Cada contador tiene su propio estado
2 let scoreCounter = createCounter(0);
3 scoreCounter.increment(); // 1
4 scoreCounter.increment(); // 2
5 console.log(scoreCounter.getValue()); // 2
6
7 // 'count' NO es accesible desde fuera
8 // console.log(count);
9 // ERROR - no existe aquí

```



# Manejo de Excepciones



## Try-Catch-Finally

- **try:** Bloque donde puede ocurrir un error
- **catch:** Captura el error si ocurre y maneja la situación
- **finally:** Se ejecuta SIEMPRE (haya error o no) - útil para limpieza
- **error.message:** Descripción del error

```

1 // Manejo de errores al cargar partida guardada
2 try {
3     // Intenta parsear JSON - puede fallar si está corrupto
4     let gameData = JSON.parse(savedGameString);
5     loadGame(gameData); // Carga el juego
6
7 } catch (error) {
8     // Se ejecuta SOLO si hay error en try
9     console.error('Error loading game:', error.message);
10    showErrorDialog('No se pudo cargar la partida');
11
12 } finally { // Opcional
13     // Se ejecuta SIEMPRE (con o sin error)
14     hideLoadingSpinner();
15 }
```



## Lanzar Excepciones

- **throw:** Lanza un error manualmente
- **new Error():** Crea objeto de error con mensaje
- **Validaciones:** Úsalo para validar inputs antes de procesar
- **El error sube:** Si no hay catch, el error se propaga hacia arriba

```
1 // Función que valida input y lanza errores si es inválido
2 function validatePlayerInput(input) {
3     // Verifica que no esté vacío
4     if (!input || input.trim() === '') {
5         throw new Error('El nombre no puede estar vacío');
6     }
7
8     // Verifica longitud máxima
9     if (input.length > 20) {
10        throw new Error('El nombre es demasiado largo');
11    }
12
13    // Si pasa las validaciones, retorna input limpio
14    return input.trim();
15 }
```



## Almacenamiento de Datos



## Local Storage

- **localStorage:** Almacena datos en el navegador de forma permanente
- **Capacidad:** ~5-10MB (depende del navegador)
- **Solo strings:** Debes convertir objetos a JSON con `JSON.stringify()`
- **Persiste:** Datos permanecen incluso cerrando navegador/reiniciando PC
- **Por dominio:** Cada sitio web tiene su propio localStorage

```

1 // Guardar datos simples (solo strings)
2 localStorage.setItem('playerName', 'Jugador1');
3 localStorage.setItem('highScore', '15000');
4
5 // Guardar objetos - primero convertir a JSON string
6 const gameConfig = { volume: 0.8, difficulty: 'normal' };
7 localStorage.setItem('gameConfig', JSON.stringify(gameConfig));
8
9 // Leer datos simples
10 const playerName = localStorage.getItem('playerName');
11 // "Jugador1"
12
13 // Eliminar un dato específico
14 localStorage.removeItem('playerName');
15
16 // Eliminar TODOS los datos (usar con cuidado)
17 localStorage.clear();

```



## Session Storage

- **sessionStorage:** API idéntica a localStorage
- **Duración:** Solo durante la sesión actual (cerrar pestaña = se borra)
- **Por pestaña:** Cada pestaña tiene su propio sessionStorage
- **Uso típico:** Datos temporales como estado actual del juego
- **No se comparte:** Entre pestañas ni ventanas

```

1 // API idéntica a localStorage
2 sessionStorage.setItem('tempData', 'valor temporal');
3 const tempData = sessionStorage.getItem('tempData');
4 sessionStorage.removeItem('tempData');
5 sessionStorage.clear();
6
7 // Ideal para estado temporal del juego actual
8 sessionStorage.setItem('currentGameState',
9     JSON.stringify(gameState));

```



## Cookies

- **Cookies:** Método antiguo de almacenamiento
- **Capacidad limitada:** Solo 4KB
- **Se envían al servidor:** En cada petición HTTP (aumenta tráfico)
- **API manual:** Más complicado que localStorage
- **Expiración configurable:** Puedes definir cuándo expiran

```

1 // Escribir Cookie con expiración (30 días desde ahora)
2 // Sin fecha es la sesión
3 const fecha = new Date();
4 fecha.setTime(fecha.getTime() + (30 * 24 * 60 * 60 * 1000));
5 document.cookie = 'highScore=15000; expires=${fecha.toUTCString()}; path=/';
6
7 // Leer cookie (complicado - no hay API directa)
8 function getCookie(nombre) {
9     const value = `; ${document.cookie}`;
10    const parts = value.split(`; ${nombre}=`);
11    if (parts.length === 2) return parts.pop().split(';').shift();
12    return null; // No encontrada
13 }
14
15 // Uso
16 let highScore = getCookie('highScore'); // "15000"
17

```



## Comparación de Métodos

- **localStorage:** Mejor opción para datos persistentes (configuración, progreso)
- **sessionStorage:** Perfecto para datos temporales de la sesión actual
- **Cookies:** Solo si necesitas comunicar con el servidor

Característica	localStorage	sessionStorage	Cookies
<b>Capacidad</b>	~5-10MB	~5-10MB	4KB
<b>Persistencia</b>	Hasta eliminar manualmente	Solo sesión actual	Configurable (expires)
<b>Envío al servidor</b>	No	No	Sí (automático)
<b>API</b>	Síncrona y simple	Síncrona y simple	Manual y compleja
<b>Compartido</b>	Entre pestañas	NO (por pestaña)	Entre pestañas



## 16.8. JavaScript OOP

1

### Desarrollo en el cliente

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño  
ruben.rodriguez@urjc.es natalia.madrueño@urjc.es  
URJC URJC

2025-09-09



2

### Tabla de contenidos

- [Orientación a Objetos](#)



# Orientación a Objetos



## Prototipos vs Clases

JavaScript usa herencia prototípica, no clases tradicionales

- Cualquier objeto puede ser prototipo de otros objetos
- Los objetos heredan directamente de otros objetos
- Cadena de herencia flexible y dinámica
- ES2015+ añadió sintaxis de clases (azúcar sintáctico sobre prototipos)

Nota

A diferencia de Java o C++, JavaScript no tiene clases “reales” en su núcleo



## Objeto Literal Simple

### La forma más directa de crear objetos

```

1 const enemigo = {
2   vida: 100,
3   damage: 15,
4   atacar() {
5     return `Enemigo ataca causando ${this.damage} puntos`;
6   }
7 };

```

- Definimos propiedades y métodos directamente
- Sintaxis clara y concisa
- Ideal para objetos únicos o configuraciones



## Creación Basada en Prototipos

### Usando `Object.create()` para heredar

```

1 const goblin = Object.create(enemigo);
2 goblin.vida = 50;
3 goblin.damage = 8;
4
5 console.log(goblin.atacar());
6 // "Enemigo ataca causando 8 puntos"

```

- `goblin` hereda el método `atacar()` de `enemigo`
- Las propiedades propias sobrescriben las heredadas
- `this` se refiere al objeto que invoca el método



## Herencia con Prototipos - Concepto

### Creando cadenas de herencia

- Cada objeto puede heredar de otro objeto
- Se forma una cadena de prototipos
- JavaScript busca propiedades/métodos en la cadena hasta encontrarlos
- Permite estructurar jerarquías de objetos



## Herencia con Prototipos - Base

### Creando el objeto base

```

1 const personajeBase = {
2   mover(x, y) {
3     this.x += x;
4     this.y += y;
5   },
6   x: 0,
7   y: 0
8 };

```

- Define comportamiento común
- Puede ser heredado por múltiples objetos
- Los métodos usan `this` para acceder a propiedades



## Herencia con Prototipos - Extensión

### Creando un prototipo intermedio

```
1 const prototipoJugador = Object.create(personajeBase);
2 prototipoJugador.atacar = function(objetivo) {
3   return `${this.nombre} ataca a ${objetivo.nombre}`;
4 };
5 prototipoJugador.nombre = "Sin nombre"
```

- Hereda de `personajeBase`
- Añade funcionalidad específica
- Añade un nombre por defecto.
- Forma el segundo eslabón de la cadena



## Herencia con Prototipos - Instancia

### Creando la instancia final

```
1 const jugador = Object.create(prototipoJugador);
2 jugador.nombre = "Aragorn";
3 jugador.x = 10;
4 jugador.y = 10;
5 jugador.puntuacion = 3;
```

- Tiene acceso a `mover()` y `atacar()`
- Define sus propias propiedades
- Propiedades que ocultan las del prototipo (locales) y nuevas.
- Completa la cadena de herencia



## Vision Completa

11

### Base

```
1 let personajeBase = {  
2   mover(x, y) {  
3     this.x += x;  
4     this.y += y;  
5   },  
6   x: 0,  
7   y: 0  
8 };
```

### Prototipo

```
1 let prototipoJugador = Object.create(personajeBase);  
2 prototipoJugador.atacar = function(objetivo) {  
3   return `${this.nombre} ataca a ${objetivo.nombre}`;  
4 };  
5 prototipoJugador.nombre = "Sin nombre"
```

### Instancia

```
1 let jugador = Object.create(prototipoJugador);
```

- Acceder a prototipoJugador.nombre y a jugador.nombre
- añadir los cambios a nivel de instancia y ejecutar de nuevo.
- Comprobar las propiedades con `Object.keys(INSTANCIA)` antes y después.

```
1 jugador.nombre = "Aragorn";  
2 jugador.x = 10;
```



## Acceso a Propiedades

12

### Notación punto

#### La forma más común y legible

```
1 const config = {  
2   sonido: true,  
3   volumen: 0.8,  
4   idioma: "es"  
5 };  
6  
7 console.log(config.sonido);  
8 config.volumen = 0.5;
```

- Sintaxis clara: `objeto.propiedad`
- Recomendada cuando el nombre es conocido
- No funciona con nombres dinámicos

### Notación corchetes

#### Para propiedades dinámicas o especiales

```
1 console.log(config["idioma"]);  
2  
3 const propiedad = "volumen";  
4 console.log(config[propiedad]);
```

- Permite nombres de propiedad dinámicos
- Útil con variables o bucles
- Necesaria para nombres con espacios o caracteres especiales



## Añadir Propiedades Dinámicamente

JavaScript permite modificar objetos en tiempo de ejecución

```
1 config.dificultad = "normal";
2 config["nivel-maximo"] = 10;
```

- Se pueden añadir propiedades después de crear el objeto
- Característica de la naturaleza dinámica de JavaScript
- Útil pero puede complicar el seguimiento del código



## Iteración sobre Propiedades - For...in

Recorriendo todas las propiedades

```
1 const inventario = {
2   espada: 1,
3   pocion: 5,
4   oro: 150
5 };
6
7 for (let item in inventario) {
8   console.log(`$item): ${inventario[item]}`);
9 }
```

- Itera sobre propiedades enumerables
- Incluye propiedades heredadas del prototipo
- La variable toma el nombre de cada propiedad



## Iteración sobre Propiedades - Verificación

### Comprobando existencia de propiedades

```

1 if ("oro" in inventario) {
2   console.log("El jugador tiene oro");
3 }
4
5 if (inventario.hasOwnProperty("espada")) {
6   console.log("Propiedad propia, no heredada");
7 }

```

- `in` verifica si existe la propiedad (propia o heredada)
- `hasOwnProperty()` verifica solo propiedades propias
- No necesitamos acceder al valor para verificar



## Iteración sobre Propiedades - Arrays

### Obteniendo claves y valores como arrays

```

1 const items = Object.keys(inventario);
2 // ["espada", "potion", "oro"]
3
4 const cantidades = Object.values(inventario);
5 // [1, 5, 150]
6
7 const pares = Object.entries(inventario);
8 // [["espada", 1], ["potion", 5], ["oro", 150]]

```

- Útil para trabajar con métodos de array (map, filter, etc.)
- Solo devuelven propiedades propias enumerables



## Función Constructor - Definición

### El patrón clásico pre-ES2015

```

1 function Jugador(nombre, x, y) {
2     this.nombre = nombre;
3     this.x = x;
4     this.y = y;
5     this.vida = 100;
6 }
```

- Se invoca con `new` para crear instancias
- `this` se refiere al nuevo objeto creado
- Inicializa las propiedades de la instancia
- Por convención, nombre en PascalCase



## Función Constructor - Métodos

### Añadiendo métodos al prototipo

```

1 Jugador.prototype.mover = function(deltaX, deltaY) {
2     this.x += deltaX;
3     this.y += deltaY;
4 };
5
6 Jugador.prototype.mostrarPosicion = function() {
7     return `${this.nombre} está en (${this.x}, ${this.y})`;
8 };
```

- Los métodos van en el prototipo, no en el constructor
- Se comparten entre todas las instancias (ahorra memoria)
- Tienen acceso a `this` de la instancia



## Función Constructor - Uso

### Creando instancias

```

1 const player1 = new Jugador("Aragorn", 10, 20);
2 const player2 = new Jugador("Legolas", 15, 25);
3
4 player1.mover(2, 3);
5 console.log(player1.mostrarPosicion());
6 // "Aragorn está en (12, 23)"

```

- Siempre usar `new` para invocar constructores
- Cada instancia tiene sus propias propiedades
- Los métodos son compartidos vía prototipo



## Herencia con Constructor - Clase Base

### Definiendo el constructor padre

```

1 function Personaje(nombre, vida) {
2     this.nombre = nombre;
3     this.vida = vida;
4 }
5
6 Personaje.prototype.saludar = function() {
7     return `Hola, soy ${this.nombre}`;
8 };

```

- Constructor base con propiedades comunes
- Métodos compartidos en el prototipo



## Herencia con Constructor - Clase Hija

### Extendiendo la funcionalidad

```
1 function Guerrero(nombre, vida, fuerza) {
2   Personaje.call(this, nombre, vida);
3   this.fuerza = fuerza;
4 }
```

- `Personaje.call(this, ...)` llama al constructor padre
- Inicializa las propiedades heredadas
- Añade propiedades específicas de `Guerrero`



## Herencia con Constructor - Cadena de Prototipos

### Estableciendo la herencia correctamente

```
1 Guerrero.prototype = Object.create(Personaje.prototype);
2 Guerrero.prototype.constructor = Guerrero;
3
4 Guerrero.prototype.atacar = function() {
5   return `${this.nombre} ataca con fuerza ${this.fuerza}`;
6 };
```

- `Object.create()` establece el prototipo correcto
- Restauramos la propiedad `constructor`
- Ahora podemos añadir métodos específicos



## Herencia con Constructor - Resultado

### Usando la herencia completa

```

1 const conan = new Guerrero("Conan", 150, 25);
2
3 console.log(conan.saludar());
4 // "Hola, soy Conan" (heredado de Personaje)
5
6 console.log(conan.atacar());
7 // "Conan ataca con fuerza 25" (propio de Guerrero)

```

- `conan` tiene acceso a métodos de ambos constructores
- Patrón verboso y propenso a errores



## Clases ES2015+ - Introducción

### Sintaxis moderna y clara

- Introducida en ES2015 (ES6)
- Azúcar sintáctico sobre prototipos
- Más familiar para desarrolladores de otros lenguajes OO
- Sintaxis más estructurada y menos propensa a errores



## Clases ES2015+ - Definición Básica

### Declarando una clase

```

1 class GameObject {
2     constructor(x, y) {
3         this.x = x;
4         this.y = y;
5         this.activo = true;
6     }
7 }
```

- Palabra clave `class` seguida del nombre
- Método `constructor()` se ejecuta al crear instancias
- Inicializa las propiedades del objeto



## Clases ES2015+ - Métodos

### Añadiendo comportamiento

```

1 class GameObject {
2     constructor(x, y) {
3         this.x = x;
4         this.y = y;
5         this.activo = true;
6     }
7
8     actualizar(deltaTime) {
9         if (!this.activo) return;
10        // Lógica de actualización
11    }
12
13     destruir() {
14         this.activo = false;
15     }
16 }
```

- Los métodos se definen dentro de la clase
- No se usa `function` keyword
- Automáticamente se añaden al prototipo



## Herencia con Clases - Extends

### Creando subclases

```

1 class Enemigo extends GameObject {
2     constructor(x, y, tipo) {
3         super(x, y); // Llamada al constructor padre
4         this.tipo = tipo;
5         this.vida = 50;
6         this.velocidad = 2;
7     }
8 }
```

- `extends` establece la herencia
- `super()` llama al constructor de la clase padre
- Debe ser la primera línea del constructor hijo



## Herencia con Clases - Sobrescritura

### Extendiendo métodos heredados

```

1 class Enemigo extends GameObject {
2     // ... constructor ...
3
4     actualizar(deltaTime) {
5         super.actualizar(deltaTime);
6         if (this.vida > 0) {
7             this.x += this.velocidad;
8         }
9     }
10 }
```

- Podemos sobrescribir métodos del padre
- `super.nombreMetodo()` llama a la versión del padre
- Permite extender sin reemplazar completamente



## Getters y Setters - Concepto

### Propiedades calculadas y validación

- Getters: propiedades de solo lectura o calculadas
- Setters: validación al asignar valores
- Se usan como propiedades normales (sin paréntesis)
- Permiten encapsular lógica de acceso



## Getters y Setters - Getter

### Propiedades de solo lectura

```

1 class Enemigo extends GameObject {
2     constructor(x, y, tipo) {
3         super(x, y);
4         this._vida = 50;
5     }
6
7     get estaVivo() {
8         return this._vida > 0;
9     }
10 }
11
12 const orc = new Enemigo(10, 20, "Orc");
13 if (orc.estaVivo) { // Sin paréntesis
14     console.log("El enemigo sigue vivo");
15 }
```

- Palabra clave `get` antes del nombre
- Se accede como propiedad, no como método
- Útil para valores derivados



## Getters y Setters - Setter

### Validación al asignar

```

1 class Enemigo extends GameObject {
2     // ... constructor ...
3
4     set vida(valor) {
5         this._vida = Math.max(0, valor);
6         if (this._vida === 0) {
7             this.destruir();
8         }
9     }
10    get vida() {
11        return this._vida;
12    }
13}
14

```

- Palabra clave `set` antes del nombre
- Permite validar y ejecutar lógica adicional
- Por convención, la propiedad real usa `_` como prefijo



## Getters y Setters - Uso

### Asignación transparente

```

1 const goblin = new Enemigo(5, 10, "Goblin");
2
3 goblin.vida = 30;      // Usa el setter
4 console.log(goblin.vida); // Usa el getter
5
6 goblin.vida = -10;     // Setter lo convierte a 0
7 // El setter también llama a destruir()

```

- Sintaxis de propiedad normal
- El setter ejecuta validación automáticamente
- Transparente para el código que usa la clase



## Métodos Estáticos - Concepto

### Métodos de la clase, no de instancias

- Pertenece a la clase, no a objetos individuales
- Se invocan sobre la clase misma
- No tienen acceso a `this` de instancia
- Útiles para funciones de utilidad o métodos factoría



## Métodos Estáticos - Definición

### Palabra clave `static`

```

1 class Enemigo extends GameObject {
2     // ... resto del código ...
3
4     static crearOrc() {
5         const orc = new Enemigo(0, 0, "Orc");
6         orc.vida = 80;
7         orc.velocidad = 1.5;
8         return orc;
9     }
10
11    static crearGoblin() {
12        const goblin = new Enemigo(0, 0, "Goblin");
13        goblin.vida = 30;
14        goblin.velocidad = 3;
15    }
16
17 }
```

- Prefijo `static` antes del método
- Patrón factoría para crear instancias preconfiguradas



## Métodos Estáticos - Uso

### Invocación sobre la clase

```

1 const orc = Enemigo.crearOrc();
2 const goblin = Enemigo.crearGoblin();
3
4 // NO se puede llamar sobre instancias
5 const enemigo = new Enemigo(5, 5, "Slime");
6 // enemigo.crearOrc(); // ✗ Error

```

- Se llaman con `NombreClase.metodoEstatico()`
- No están disponibles en las instancias
- Similares a métodos de clase en Java/C++



## Uso de Clases - Creación

### Instanciando objetos

```

1 const goblin = new Enemigo(10, 20, "Goblin");
2 goblin.vida = 30;
3
4 const orc = Enemigo.crearOrc(); // Método estático

```

- Constructor normal con `new`
- Podemos usar setters para asignar valores
- Los métodos estáticos crean instancias preconfiguradas



## Uso de Clases - Polimorfismo

### Tratamiento uniforme de objetos diferentes

```

1 const enemigos = [goblin, orcl];
2
3 enemigos.forEach(enemigo => {
4   enemigo.actualizar(16);
5   if (enemigo.estaVivo) {
6     enemigo.mover(1, 0);
7   }
8 });

```

- Todos los enemigos comparten la misma interfaz
- Podemos tratarlos de forma uniforme
- Cada uno puede tener comportamiento específico



## Ventajas de Clases ES2015+

### Por qué usar la sintaxis moderna

- **Sintaxis más clara:** Familiar para otros lenguajes OO
- **Herencia simplificada:** `extends` y `super()` más directos
- **Menos errores:** Estructura más rígida previene problemas comunes
- **Getters y setters integrados:** Encapsulación natural
- **Métodos estáticos:** Organización clara de funciones de utilidad
- **Mejor soporte de herramientas:** IDEs y linters funcionan mejor



La sintaxis de clases es la **recomendada** para proyectos nuevos



## Resumen: Prototipos vs Clases

## Dos formas, mismo resultado

Aspecto	Prototipos	Clases ES2015+
Sintaxis	Verbosa y manual	Clara y estructurada
Herencia	<code>Object.create()</code> y <code>call()</code>	<code>extends</code> y <code>super()</code>
Métodos	<code>Funcion.prototype.metodo</code>	Dentro de la clase
Uso actual	Legacy code	<b>Recomendado</b>

**! Importante**

Las clases son azúcar sintáctico: por debajo siguen usando prototipos



## 16.9. Phaser

# Introducción a Phaser 3

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez      ruben.rodriguez@urjc.es  
Natalia Madrueño      natalia.madrueno@urjc.es  
URJC      URJC

2025-09-09



## Tabla de contenidos

- [Introducción a Phaser 3](#)
- [Estructura Básica](#)
- [Gestión de Escenas](#)
- [Trabajo con Imágenes](#)
- [Motores de Físicas](#)
- [Sistema de Entrada](#)
- [Detección de Colisiones](#)
- [Patrón Command](#)
- [Resumen](#)



## Introducción a Phaser 3



## ¿Qué es Phaser 3?

Framework open source de HTML5 para videojuegos

- Juegos que se ejecutan **directamente en navegadores web**
- Liberado en **2018** - evolución de versiones anteriores
- Enfocado en **juegos 2D** multiplataforma

### Ventajas:

- Sin instalación para usuarios finales
- Multiplataforma (desktop + móvil)
- Distribución simple vía web
- Basado en tecnologías estándar

### Tecnologías:

- HTML5 Canvas
- JavaScript/TypeScript
- Canvas API / WebGL



## Sistemas de Renderizado

Dos opciones disponibles:

- **Canvas (por defecto):** Mayor compatibilidad, menos exigente
- **WebGL:** Gráficos avanzados, mejor rendimiento, requiere soporte GPU

¿Cuándo usar cada uno?

- Canvas: Juegos simples, máxima compatibilidad
- WebGL: Muchos objetos, efectos visuales complejos

```

1 // Configurar tipo de renderizado
2 const config = {
3   type: Phaser.AUTO, // Elige el mejor disponible
4   // type: Phaser.CANVAS, // Forzar Canvas
5   // type: Phaser.WEBGL, // Forzar WebGL
6 };

```



## Requisitos para Empezar

### Navegador moderno:

- Chrome (recomendado - mejores DevTools)
- Firefox, Safari, Edge u Opera

### Obtener Phaser 3:

- [CDN](#) (más rápido)

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

- [Descarga directa](#) [phaser.io/download/stable](https://phaser.io/download/stable)
- [GitHub](#) [github.com/photonstorm/phaser](https://github.com/photonstorm/phaser)
- [Node](#) npm install phaser



## Estructura Básica



## El Elemento Canvas

### ¿Qué es el Canvas?

- Etiqueta HTML5 ( <canvas> ) para dibujar gráficos
- Funciona como un **lienzo en blanco**
- Se manipula mediante JavaScript

### Sistema de coordenadas:

- Origen **(0,0)** en esquina superior izquierda
- Eje X positivo → derecha
- Eje Y positivo → abajo (inverso a matemáticas)
- Dimensiones por defecto: 300x300 píxeles

```
1  <canvas id="game" width="800" height="600"></canvas>
```



## Configuración Inicial

### Objeto de configuración define parámetros fundamentales:

- Tipo de renderizado (AUTO/CANVAS/WEBGL)
- Dimensiones del juego
- Motor de físicas y gravedad
- Funciones principales del juego

```
1 const config = {
2   type: Phaser.AUTO,           // Renderizado automático
3   width: 800,                 // Ancho del canvas
4   height: 600,                // Alto del canvas
5   physics: {
6     default: 'arcade',       // Motor de físicas
7     arcade: {
8       gravity: { y: 300 }    // Gravedad vertical
9     }
10   },
11   scene: {
12     preload: preload,        // Función de carga
13     create: create,          // Función de creación
14     update: update           // Función de actualización
15   }
16};
```



## Las Tres Funciones Principales

### preload() - Cargar recursos

- **Cuándo:** Se ejecuta primero, solo una vez
- **Para qué:** Cargar imágenes, sonidos, sprites
- **Importante:** Phaser espera a que todo se cargue antes de continuar

```
1 function preload() {
2     // Cargar recursos con identificadores únicos
3     this.load.image('cielo', 'assets/cielo.png');
4     this.load.image('suelo', 'assets/plataforma.png');
5     this.load.image('estrella', 'assets/estrella.png');
6 }
```



El primer parámetro es el **identificador** que usaremos después



## Las Tres Funciones Principales (2)

### create() - Crear objetos del juego

- **Cuándo:** Se ejecuta después de preload()
- **Para qué:** Inicializar y posicionar elementos
- **Importante:** El orden de creación determina el orden de renderizado

```
1 function create() {
2     // Añadir imagen de fondo centrada
3     this.add.image(400, 300, 'cielo');
4
5     // Crear plataforma con físicas
6     this.plataforma = this.physics.add.image(400, 500, 'suelo');
7     this.plataforma.setImmovable(true); // No se mueve
8 }
```



## Las Tres Funciones Principales (3)

### update(time, delta) - Bucle del juego

- **Cuándo:** Se ejecuta constantemente (~60 veces/segundo)
- **Para qué:** Lógica que debe actualizarse continuamente
- **Parámetros:** `time` (tiempo total), `delta` (desde último frame)

```
1 function update(time, delta) {  
2     // Mover jugador con teclado  
3     if (this.cursors.left.isDown) {  
4         this.jugador.x -= 5; // Mover izquierda  
5     }  
6     if (this.cursors.right.isDown) {  
7         this.jugador.x += 5; // Mover derecha  
8     }  
9 }
```



Usar `delta` para movimiento consistente en diferentes dispositivos



## Gestión de Escenas



## Concepto de Escena

### ¿Qué es una escena?

- Pantalla o estado independiente del juego
- Tiene su propio flujo de ejecución (preload, create, update)
- Mantiene sus propios recursos y objetos
- Permite organización modular del código

### Ejemplos comunes:

- Menú principal
- Pantalla de gameplay
- Menú de pausa
- Pantalla de game over
- Tutorial
- Pantalla de créditos



## Escena Implícita vs Explícita

### Forma implícita (simple):

```
1 // Crea una escena anónima automáticamente
2 const config = {
3   scene: {
4     preload: preload,
5     create: create,
6     update: update
7   }
8 };
```

### Forma explícita (recomendada):

```
1 class MiEscena extends Phaser.Scene {
2   preload() { /* ... */ }
3   create() { /* ... */ }
4   update() { /* ... */ }
5 }
6
7 const config = {
8   scene: [MiEscena, OtraEscena]
9 };
```

Nota

Las funciones preload/create/update **siempre** están dentro de una escena



## SceneManager - Métodos Disponibles

### Cambiar entre escenas:

- `start()` : Inicia escena y **detiene la actual**
- `launch()` : Inicia escena **en paralelo** (mantiene actual)
- `stop()` : Detiene completamente una escena

### Controlar estado:

- `pause()` / `resume()` : Pausar/reanudar (sigue visible)
- `sleep()` / `wake()` : Dormir/despertar (no visible)

### Gestionar orden:

- `bringToFront()` / `sendToBack()` : Mover al frente/fondo
- `moveAbove()` / `moveBelow()` : Posicionar relativamente



## Configurar Múltiples Escenas

### Definir escenas en el array de configuración:

```
1 // La primera escena del array se inicia automáticamente
2 const config = {
3   scene: [MenuPrincipal, Juego, GameOver]
4 };
```

### Añadir/eliminar dinámicamente:

```
1 // Añadir nueva escena en tiempo de ejecución
2 this.scene.add('clave', ConfigEscena, autoStart, datos);
3
4 // Eliminar una escena
5 this.scene.remove('clave');
```



## Cambiar Entre Escenas

### start() vs launch():

```

1 // start: Detiene la actual y cambia
2 this.scene.start('Juego', { nivel: 1 });
3
4 // launch: Ejecuta en paralelo (útil para HUDs)
5 this.scene.launch('MenuPausa', { origenEscena: 'Juego' });
6
7 // stop: Detiene completamente
8 this.scene.stop('MenuPausa');

```



**Diferencia clave:** `start` reemplaza, `launch` superpone escenas



## Pausar y Reanudar

### pause() / resume():

- Detiene `update()` pero sigue renderizando
- Útil para pausar el juego manteniéndolo visible

```

1 // Pausar (sigue visible, no actualiza)
2 this.scene.pause('Juego');
3
4 // Reanudar
5 this.scene.resume('Juego');

```

### sleep() / wake():

- Detiene `update()` y renderizado
- Más eficiente que pause

```

1 // Dormir (no visible, no actualiza)
2 this.scene.sleep('Fondo');
3
4 // Despertar
5 this.scene.wake('Fondo');

```



## Sistema de Pausa - Ejemplo

### Escena principal del juego:

```

1 class Juego extends Phaser.Scene {
2     constructor() {
3         super('Juego');
4     }
5
6     create() {
7         // Configurar juego...
8
9         // Detectar tecla ESC para pausar
10        this.input.keyboard.on('keydown-ESC', () => {
11            this.scene.pause(); // Pausar juego
12            this.scene.launch('MenuPausa', {
13                escenaOrigen: 'Juego'
14            });
15        });
16    }
17 }
```



## Sistema de Pausa - Ejemplo (2)

### Escena del menú de pausa:

```

1 class MenuPausa extends Phaser.Scene {
2     constructor() {
3         super('MenuPausa');
4     }
5
6     create(datos) {
7         // Recibir datos de la escena que pausó
8         console.log('Pausado desde:', datos.escenaOrigen);
9
10        // Crear fondo semitransparente
11        let overlay = this.add.rectangle(400, 300, 800, 600, 0x000000, 0.7);
12
13        // Botón continuar
14        let botonContinuar = this.add.text(400, 300, 'Continuar', {
15            fontSize: '32px'
16        }).setOrigin(0.5);
17        botonContinuar.setInteractive();
18
19        botonContinuar.on('pointerdown', () => {
20            this.scene.stop(); // Cerrar menú
21            this.scene.resume(datos.escenaOrigen); // Reanudar juego
22        });
23    }
24 }
```



## Pasar Datos Entre Escenas

Enviar datos al cambiar de escena:

```
1 // Pasar datos al iniciar una escena
2 this.scene.start('SiguienteEscena', {
3     puntuacion: 100,
4     nivel: 2,
5     nombre: 'Jugador1'
6});
```

Recibir datos en la escena destino:

```
1 class SiguienteEscena extends Phaser.Scene {
2     create(datos) {
3         // Recibir los datos como parámetro
4         console.log(datos.puntuacion); // 100
5         console.log(datos.nivel); // 2
6         console.log(datos.nombre); // 'Jugador1'
7
8         this.add.text(100, 100, `Puntuación: ${datos.puntuacion}`);
9     }
10 }
```



## Pasar Datos - Otros Métodos

También funciona con launch() y otros:

```
1 // Con launch (escena en paralelo)
2 this.scene.launch('MenuPausa', {
3     juegoActual: 'Nivel1',
4     tiempoTranscurrido: 120
5 });
6
7 // Con transition
8 this.scene.transition({
9     target: 'GameOver',
10    duration: 1000,
11    data: {
12        puntuacionFinal: this.puntuacion,
13        tiempoJugado: this.tiempo
14    }
15});
```



## Transiciones Entre Escenas

### Crear efectos visuales suaves:

- `duration` : Tiempo en milisegundos
- `onUpdate` : Callback durante transición
- `progress` : Valor de 0 a 1
- `data` : Objeto con datos para la escena destino

```

1 this.scene.transition({
2   target: 'SiguienteEscena',
3   duration: 1000,           // 1 segundo
4   moveBelow: true,
5   data: {                  // Datos a pasar
6     puntuacion: this.puntos,
7     nivel: this.nivelActual
8   },
9   onUpdate: (progress) => {
10    // Crear efectos de fundido
11    this.cameras.main.setAlpha(1 - progress);
12  }
13 });

```



## Reordenar Escenas

### Control del orden de renderizado:

- Valores mayores se dibujan encima
- Útil para HUDs y menús superpuestos

```

1 // Operaciones básicas
2 this.scene.bringToFront('HUD');
3 this.scene.sendToBack('Fondo');
4
5 // Movimiento relativo
6 this.scene.moveAbove('A', 'B');    // A encima de B
7 this.scene.moveBelow('A', 'B');    // A debajo de B
8
9 // Movimiento incremental
10 this.scene.moveUp('Escena');
11 this.scene.moveDown('Escena');

```



# Trabajo con Imágenes



## Cargar y Mostrar

Proceso en dos pasos:

1. **Cargar** en `preload()` con identificador único
2. **Mostrar** en `create()` usando el identificador

```
1 function preload() {  
2     // Cargar con identificador único  
3     this.load.image('personaje', 'assets/personaje.png');  
4     this.load.image('fondo', 'assets/fondo.jpg');  
5 }  
6  
7 function create() {  
8     // Añadir al canvas en posición (x, y)  
9     this.add.image(400, 300, 'fondo');  
10    let sprite = this.add.image(200, 150, 'personaje');  
11 }
```

**Advertencia**

Siempre cargar en `preload()` antes de usar en `create()`



## Sistema de Coordenadas y Origen

### Origen por defecto: centro de la imagen

- Valores de 0 a 1
- (0, 0) = esquina superior izquierda
- (0.5, 0.5) = centro (defecto)
- (1, 1) = esquina inferior derecha

```

1 // Cambiar punto de origen
2 let imagen = this.add.image(100, 100, 'personaje')
3   .setOrigin(0, 0); // Esquina superior izquierda
4
5 imagen.setOrigin(0.5, 1); // Centro inferior
6 imagen.setOrigin(1, 0); // Esquina superior derecha

```



Cambiar origen útil para alinear objetos o rotaciones específicas



## Transformaciones Básicas

### Escalar, voltear y rotar:

```

1 let jugador = this.add.image(400, 300, 'personaje');
2
3 // Escalar uniformemente
4 jugador.setScale(1.5); // 150% del tamaño
5
6 // Escalar independientemente
7 jugador.setScale(2, 0.5); // Ancho x2, alto x0.5
8
9 // Voltear
10 jugador.flipX = true; // Espejo horizontal
11 jugador.flipY = true; // Espejo vertical
12
13 // Rotar (en radianes)
14 jugador.rotation = Math.PI / 4; // 45 grados

```



$\pi$  radianes = 180 grados



## Profundidad y Capas

Controlar qué se dibuja encima:

- La propiedad `depth` funciona como capas
- Valores mayores se dibujan sobre valores menores
- Por defecto todas tienen `depth = 0`

```
1 let fondo = this.add.image(400, 300, 'cielo');
2 fondo.depth = 0; // Atrás
3
4 let jugador = this.add.image(400, 300, 'personaje');
5 jugador.depth = 10; // Encima del fondo
6
7 let hud = this.add.image(400, 50, 'interfaz');
8 hud.depth = 100; // Encima de todo
```



## Motores de Físicas



## Tres Motores Disponibles

### Arcade Physics:

- Más simple y rápido
- Solo rectángulos y círculos
- Ideal para juegos arcade/plataformas

### Impact Physics:

- Soporta pendientes en tiles
- Más complejo que Arcade
- Ideal para plataformas con terreno inclinado

### Matter Physics:



## Configurar Arcade Physics

### Configuración en el objeto config:

```

1 const config = {
2   physics: {
3     default: 'arcade',
4     arcade: {
5       gravity: { y: 300 }, // Gravedad en píxeles/s2
6       debug: false        // Mostrar contornos físicos
7     }
8   }
9 };

```



Activar `debug: true` durante desarrollo para ver colisiones



## Añadir Físicas a Objetos

Diferencia entre objetos con y sin físicas:

```

1 // Sin físicas (imagen estática)
2 this.add.image(400, 300, 'fondo');
3
4 // Con físicas (puede moverse, colisionar, etc.)
5 this.jugador = this.physics.add.image(100, 450, 'personaje');
```



Usar `physics.add` en lugar de solo `add` para habilitar físicas



## Propiedades Físicas

Configurar comportamiento físico:

```

1 this.jugador = this.physics.add.image(100, 450, 'personaje');
2
3 // No salir de los límites del canvas
4 this.jugador.setCollideWorldBounds(true);
5
6 // Rebote al chocar (0 = no rebota, 1 = rebote perfecto)
7 this.jugador.setBounce(0.3);
8
9 // Velocidad inicial (pixeles/segundo)
10 this.jugador.setVelocity(100, -50);
11
12 // Aceleración (pixeles/segundo2)
13 this.jugador.setAcceleration(50, 0);
```



# Sistema de Entrada



## Teclado - Eventos

Detectar teclas específicas:

```
1 function create() {  
2     // Detectar cuando se presiona  
3     this.input.keyboard.on('keydown-SPACE', () => {  
4         this.jugador.setVelocityY(-330); // Saltar  
5     });  
6     // Detectar cuando se suelta  
7     this.input.keyboard.on('keyup-SPACE', () => {  
8         console.log('Tecla soltada');  
9     });  
10 }  
11 }
```

Nota

`keydown` se dispara al presionar, `keyup` al soltar



## Teclado - Modificadores

Detectar combinaciones de teclas:

```

1 this.input.keyboard.on('keydown-A', (event) => {
2   if (event.ctrlKey) {
3     console.log('CTRL + A');
4   } else if (event.shiftKey) {
5     console.log('SHIFT + A');
6   } else if (event.altKey) {
7     console.log('ALT + A');
8   } else {
9     console.log('Solo A');
10  }
11 });

```



## Teclado - Cursos

Objeto para flechas, espacio y shift:

```

1 function create() {
2   // Crear objeto de cursos
3   this.cursors = this.input.keyboard.createCursorKeys();
4 }
5
6 function update() {
7   // Resetear velocidad
8   this.jugador.setVelocityX(0);
9
10  // Comprobar teclas presionadas
11  if (this.cursors.left.isDown) {
12    this.jugador.setVelocityX(-160);
13  } else if (this.cursors.right.isDown) {
14    this.jugador.setVelocityX(160);
15  }
16
17  // Saltar solo si está en el suelo
18  if (this.cursors.up.isDown && this.jugador.body.touching.down) {
19    this.jugador.setVelocityY(-330);
20  }
21 }

```



## Teclado - Combos

### Detectar secuencias de teclas:

- Útil para códigos secretos o trucos
- Detecta automáticamente la secuencia correcta

```

1 function create() {
2     // Combo con letras
3     let combo1 = this.input.keyboard.createCombo('KONAMI');
4
5     // Combo Konami: ↑↑↓↓←←→→
6     let combo2 = this.input.keyboard.createCombo(
7         [38, 38, 40, 40, 37, 39, 37, 39]
8     );
9
10    // Detectar cuando se completa
11    this.input.keyboard.on('keycombonmatch', (combo) => {
12        console.log('¡Código secreto desbloqueado!');
13        this.activarModoEspecial();
14    });
15 }
```



## Ratón y Táctil

### API unificada para ratón y pantallas táctiles:

```

1 function create() {
2     // Detectar clic/toque
3     this.input.on('pointerdown', (pointer) => {
4         if (pointer.leftButtonDown()) {
5             console.log('Clic izquierdo:', pointer.x, pointer.y);
6         }
7         if (pointer.rightButtonDown()) {
8             console.log('Clic derecho');
9         }
10    });
11
12    // Detectar cuando se suelta
13    this.input.on('pointerup', (pointer) => {
14        console.log('Botón soltado');
15    });
16
17    // Detectar movimiento
18    this.input.on('pointermove', (pointer) => {
19        console.log('Posición:', pointer.x, pointer.y);
20    });
21 }
```



## Detección de Colisiones



### Collider vs Overlap

Dos formas de detectar interacciones:

**Collider:**

- Detecta **y resuelve** colisiones físicamente
- Separa objetos automáticamente
- Aplica masa, velocidad, rebote

**Overlap:**

- Solo **detecta** superposición espacial
- Los objetos pueden atravesarse
- Sin resolución física



Collider para plataformas, Overlap para colecciónables



## Ejemplos de Uso

### Collider para plataformas:

```

1 function create() {
2     let jugador = this.physics.add.image(100, 450, 'personaje');
3
4     let plataforma = this.physics.add.image(400, 500, 'suelo');
5     plataforma.setImmovable(true); // No se mueve al impacto
6
7     // Colisión con resolución física
8     this.physics.add.collider(jugador, plataforma);
9 }
```

### Overlap para colecciónables:

```

1 let estrella = this.physics.add.image(200, 200, 'estrella');
2
3 this.physics.add.overlap(jugador, estrella, (j, e) => {
4     e.destroy(); // Eliminar estrella
5     this.puntuacion += 10;
6 });
```



## Grupos Estáticos

### Para objetos inmóviles (plataformas, paredes):

- Más eficientes que objetos dinámicos
- No se mueven ni responden a físicas
- Ideales para nivel/escenario

```

1 function create() {
2     // Crear grupo estático
3     let plataformas = this.physics.add.staticGroup();
4
5     // Añadir plataformas
6     plataformas.create(400, 568, 'suelo')
7         .setScale(2)
8         .refreshBody(); // Actualizar tras modificar
9
10    plataformas.create(600, 400, 'suelo');
11    plataformas.create(50, 250, 'suelo');
12
13    // Colisión con todas
14    this.physics.add.collider(this.jugador, plataformas);
15 }
```



Advertencia



## Grupos Dinámicos

### Para objetos con físicas completas:

- Se mueven y responden a gravedad
- Pueden colisionar entre sí
- Más costosos computacionalmente

```

1 // Crear 12 estrellas espaciadas
2 let estrellas = this.physics.add.group({
3   key: 'estrella',
4   repeat: 11, // 1 + 11 = 12 total
5   setXY: { x: 12, y: 0, stepX: 70 } // Cada 70px
6 });
7
8 // Aplicar propiedades a cada una
9 estrellas.children.iterate((estrella) => {
10   estrella.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));
11 });
12
13 // Overlap para recogerlas
14 this.physics.add.overlap(this.jugador, estrellas, (j, e) => {
15   e.disableBody(true, true); // Desactivar
16   this.puntuacion += 10;
17 });

```



## Callbacks de Colisión

### Ejecutar código cuando ocurre colisión:

```

1 // Callback básico
2 this.physics.add.collider(jugador, enemigo, (j, e) => {
3   // Se ejecuta cada vez que colisionan
4   j.setTint(0xff0000); // Jugador en rojo
5   j.setVelocityX(-200); // Retroceder
6 });

```

### Process callback (condición):

- Decide si procesar la colisión
- Retorna `true` para procesar, `false` para ignorar

```

1 this.physics.add.collider(
2   jugador,
3   enemigo,
4   this.dañar, // Si procesa
5   this.puedeRecibirDaño, // Condición
6   this
7 );

```



## Sistema de Invulnerabilidad

### Ejemplo: no recibir daño temporalmente

```

1 function puedeRecibirDaño(jugador, enemigo) {
2     // Solo procesar si no es invulnerable
3     return !jugador.invulnerable;
4 }
5
6 function dañar(jugador, enemigo) {
7     jugador.invulnerable = true;
8     jugador.setTint(0xff0000); // Rojo
9
10    // Volver vulnerable tras 2 segundos
11    this.time.delayedCall(2000, () => {
12        jugador.invulnerable = false;
13        jugador.clearTint();
14    });
15 }
```



## Detectar Contacto con Superficies

### Propiedades útiles para mecánicas:

- `touching.down`: Tocando suelo
- `touching.up`: Tocando techo
- `touching.left/right`: Tocando paredes
- `blocked.*`: Similar pero solo objetos inmóviles

```

1 function update() {
2     // Verificar si toca el suelo
3     if (this.jugador.body.touching.down) {
4         console.log('En el suelo');
5     }
6
7     // Saltar solo si está en el suelo
8     if (this.cursors.up.isDown &&
9         this.jugador.body.touching.down) {
10        this.jugador.setVelocityY(-330);
11    }
12
13     // Detectar si está contra pared
14     if (this.jugador.body.touching.left) {
15         console.log('Pared izquierda');
16     }
17 }
```



# Patrón Command



## ¿Qué es el Patrón Command?

**Problema en juegos multijugador:**

- Capturar acciones del jugador (input)
- Ejecutar localmente (respuesta inmediata)
- Transmitir por red a otros jugadores
- Reproducir acciones recibidas

**Solución: Patrón Command**

- Encapsula cada acción como un objeto
- Separa input, lógica y networking
- Facilita testing y debugging



## Beneficios del Patrón

### Ventajas principales:

- **Separación de responsabilidades:** Input, lógica y red independientes
- **Testabilidad:** Probar comandos sin juego completo
- **Flexibilidad:** Añadir comandos sin modificar código existente
- **Networking transparente:** Mismo comando para local y red
- **Debugging:** Registrar/reproducir todas las acciones
- **Predicción cliente:** Respuesta inmediata + reconciliación servidor



## Estructura Base - Command

### Clase abstracta con interfaz común:

```

1 class Command {
2     execute() {
3         // Implementar en subclases
4         // Ejecuta la acción en el juego
5     }
6
7     serialize() {
8         // Convertir a JSON para enviar por red
9         return {};
10    }
11
12    getPlayer() {
13        // Retorna la entidad asociada
14        return null;
15    }
16 }
```

Nota

Esta es la base que heredarán todos los comandos específicos



## Comando Concreto - Constructor

Definir el comando específico:

```

1 class MovePaddleCommand extends Command {
2     constructor(paddle, direction) {
3         super();
4         this.paddle = paddle;           // Referencia al objeto
5         this.direction = direction;   // 'up', 'down', 'stop'
6     }
7
8     // Métodos en siguientes slides...
9 }
```



Cada comando guarda toda la información necesaria para ejecutarse



## Comando Concreto - Ejecutar

Implementar la lógica del comando:

```

1 class MovePaddleCommand extends Command {
2     // ... constructor ...
3
4     execute() {
5         const speed = 300; // Píxeles por segundo
6
7         if (this.direction === 'up') {
8             this.paddle.setVelocityY(-speed);
9         } else if (this.direction === 'down') {
10            this.paddle.setVelocityY(speed);
11        } else { // 'stop'
12            this.paddle.setVelocityY(0);
13        }
14    }
15
16    // Métodos serialize() y getPlayer() en siguiente slide...
17 }
```



## Comando Concreto - Serializar

Preparar para transmisión por red:

```

1 class MovePaddleCommand extends Command {
2     // ... constructor y execute() ...
3
4     serialize() {
5         // Convertir a formato JSON simple
6         return {
7             type: 'MOVE_PADDLE',
8             playerId: this.paddle.id,
9             direction: this.direction
10        };
11    }
12
13    getPlayer() {
14        return this.paddle;
15    }
16 }
```



Solo se envían datos esenciales, no referencias a objetos



## CommandProcessor - Estructura

Coordina comandos locales y remotos:

```

1 class CommandProcessor {
2     constructor() {
3         this.players = new Map(); // Registro de jugadores
4         this.network = null; // Gestor de red
5     }
6
7     setNetwork(networkManager) {
8         this.network = networkManager;
9     }
10
11    // Métodos process() y receiveCommand() en siguientes slides...
12 }
```



## CommandProcessor - Procesar Local

Ejecutar comandos del jugador local:

```

1 class CommandProcessor {
2     // ... constructor ...
3
4     process(command) {
5         const player = command.getPlayer();
6
7         // Solo ejecutar si es jugador local
8         if (player && player.authority === 'LOCAL') {
9             command.execute(); // Respuesta inmediata
10
11         // Transmitir a otros jugadores
12         if (this.network && this.network.isConnected()) {
13             this.network.send(command.serialize());
14         }
15     }
16 }
17 }
```

 **Importante**

La autoridad `LOCAL` evita ejecutar el comando dos veces



## CommandProcessor - Recibir Remoto

Procesar comandos de otros jugadores:

```

1 class CommandProcessor {
2     // ... process() ...
3
4     receiveCommand(data) {
5         const player = this.players.get(data.playerId);
6
7         // Solo ejecutar si es jugador remoto
8         if (player && player.authority === 'REMOTE') {
9             const command = this.deserialize(data, player);
10            if (command) {
11                command.execute();
12            }
13        }
14    }
15 }
```

 **Nota**

Comandos remotos solo se ejecutan al recibirlas por red



## CommandProcessor - Deserializar

Reconstruir comandos desde JSON:

```

1 class CommandProcessor {
2     // ... receiveCommand() ...
3
4     deserialize(data, player) {
5         switch(data.type) {
6             case 'MOVE_PADDLE':
7                 return new MovePaddleCommand(player, data.direction);
8
9             case 'SHOOT':
10                return new ShootCommand(player, data.x, data.y);
11
12             default:
13                 console.warn('Comando desconocido:', data.type);
14                 return null;
15         }
16     }
17 }
```



## Integración en Phaser - Create

Configurar escena con autoridad:

```

1 class GameScene extends Phaser.Scene {
2     constructor() {
3         super('GameScene');
4         this.commandProcessor = new CommandProcessor();
5     }
6
7     create() {
8         // Paleta local (controlada por este jugador)
9         this.localPaddle = this.physics.add.image(50, 300, 'paddle');
10        this.localPaddle.id = 'player1';
11        this.localPaddle authority = 'LOCAL';
12        this.localPaddle.setCollideWorldBounds(true);
13
14        // Continúa en siguiente slide...
15    }
16 }
```



## Integración en Phaser - Create (2)

Configurar paleta remota y registrar:

```

1 create() {
2     // ... localPaddle ...
3
4     // Paleta remota (controlada por otro jugador)
5     this.remotePaddle = this.physics.add.image(750, 300, 'paddle');
6     this.remotePaddle.id = 'player2';
7     this.remotePaddle.authority = 'REMOTE';
8     this.remotePaddle.setCollideWorldBounds(true);
9
10    // Registrar ambos jugadores
11    this.commandProcessor.players.set('player1', this.localPaddle);
12    this.commandProcessor.players.set('player2', this.remotePaddle);
13
14    // Configurar input
15    this.cursors = this.input.keyboard.createCursorKeys();
16 }
```



## Integración en Phaser - Update

Convertir input a comandos:

```

1 update() {
2     // Crear comando según tecla presionada
3     let command;
4
5     if (this.cursors.up.isDown) {
6         command = new MovePaddleCommand(this.localPaddle, 'up');
7     } else if (this.cursors.down.isDown) {
8         command = new MovePaddleCommand(this.localPaddle, 'down');
9     } else {
10        command = new MovePaddleCommand(this.localPaddle, 'stop');
11    }
12
13    // Procesar (ejecuta local y envía a red)
14    this.commandProcessor.process(command);
15 }
```



## Flujo Completo del Patrón

### Jugador Local:

1. Presiona tecla en `update()`
2. Crea `MovePaddleCommand`
3. `CommandProcessor.process()` ejecuta inmediatamente
4. Serializa y envía por red

### Jugador Remoto:

1. Recibe datos JSON por red
2. `CommandProcessor.receiveCommand()` deserializa
3. Crea `MovePaddleCommand` con datos recibidos
4. Ejecuta el comando



## Ventajas en Arquitectura

### Transparencia de red:

- Mismo código para local y remoto
- Solo cambia la autoridad (LOCAL/REMOTE)

### Preparado para evolución:

- Fácil cambiar de REST a WebSockets
- Solo modificar NetworkManager
- Comandos permanecen iguales

### Escalabilidad:

- Añadir nuevos comandos = nueva clase
- No modificar código existente
- Principio Open/Closed



# Resumen



## Conceptos Clave de Phaser

### Estructura básica:

- Canvas con coordenadas (0,0 arriba-izquierda)
- Tres funciones: `preload`, `create`, `update` (dentro de escenas)
- SceneManager para múltiples pantallas
- Pasar datos entre escenas con parámetro en `create()`

### Físicas:

- Arcade Physics (simple y rápido)
- Collider (con física) vs Overlap (sin física)
- Grupos estáticos y dinámicos

### Input:



## Conceptos Clave del Patrón Command

### Arquitectura:

- Command: Encapsula cada acción
- CommandProcessor: Coordina local y red
- Autoridad LOCAL/REMOTE

### Beneficios:

- Separación input/lógica/red
- Testing simplificado
- Debugging con registro de comandos
- Preparado para networking



## 16.10. APIs REST

## APIs REST

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez    Natalia Madrueño  
 ruben.rodriguez@urjc.es    natalia.madrueño@urjc.es  
 URJC    URJC

2025-09-09



## Tabla de contenidos

- [Introducción a APIs REST](#)
- [Niveles de Madurez REST](#)
- [Formato JSON](#)
- [Principios de Diseño REST](#)
- [Métodos HTTP](#)
- [Anatomía de Peticiones y Respuestas](#)
- [Códigos de Estado HTTP](#)
- [Cliente JavaScript con fetch\(\)](#)
- [Servidor con Node.js y Express](#)
- [Ventajas de la Arquitectura REST](#)
- [Resumen](#)



# Introducción a APIs REST



## Contexto en Juegos Multijugador

### Después de implementar el juego con Phaser 3 y el Patrón Command

- Cliente y servidor necesitan comunicarse
- Gestionar registro y autenticación de jugadores
- Almacenar puntuaciones y estadísticas
- Gestionar partidas y salas de juego
- Consultar rankings y perfiles



REST para operaciones que no requieren actualización instantánea



## Comunicación HTTP Tradicional vs API REST

### Aplicaciones web tradicionales:

- Peticiones HTTP devuelven documentos HTML
- El navegador renderiza la página completa

### Aplicaciones con AJAX y SPA:

- Peticiones HTTP intercambian información estructurada
- No devuelven HTML, sino datos (típicamente JSON)
- El cliente procesa y actualiza la interfaz

```
Petición:  
GET http://www.mygame.com/players/alice  
  
Respuesta:  
{  
  "id": "alice",  
  "name": "Alice Smith",  
  "level": 15,  
  "score": 8500  
}
```



## ¿Qué es una API REST?

### REST (Representational State Transfer)

- Estilo de arquitectura de software basado en HTTP
- Acuñado en 2000 por **Roy Fielding** (coautor de HTTP)
- Cuando un servicio cumple estos principios es **RESTful**

### Características clave:

- Operaciones CRUD sobre recursos del servidor
- Aprovecha URL, métodos HTTP, códigos de estado
- Intercambio de información en formato JSON
- Simplicidad y eficiencia



REST se ha convertido en el estándar de facto frente a otras alternativas como SOAP



## Niveles de Madurez REST



## Modelo de Richardson

4 niveles según conformidad con principios REST



## Nivel 0 - The Swamp of POX

### Plain Old XML

- HTTP solo como sistema de transporte
- Todas las peticiones POST a una única URL
- El cuerpo indica qué operación realizar
- No aprovecha características de HTTP

```
POST /api HTTP/1.1
Body: <operation>getPlayer</operation><id>alice</id>
```

**Advertencia**

Similar a usar HTTP como túnel para otros protocolos



## Nivel 1 - Resources

### Introducción de recursos individuales

- Cada recurso tiene su propia URI específica
- `/players/alice` y `/matches/123` son URLs distintas
- Todavía no usa correctamente los métodos HTTP
- Típicamente solo POST para todas las operaciones

```
POST /players/alice HTTP/1.1
POST /matches/123 HTTP/1.1
```



## Nivel 2 - HTTP Verbs

### Uso correcto de métodos y códigos HTTP

- GET para obtener, POST para crear
- PUT para actualizar, DELETE para eliminar
- Códigos de estado apropiados (200, 404, 500, etc.)
- Aprovecha la semántica del protocolo HTTP

! Importante

Este es el nivel que usaremos en este curso

```
GET /players/alice HTTP/1.1      → 200 OK
POST /matches HTTP/1.1           → 201 Created
DELETE /players/bob HTTP/1.1     → 204 No Content
```



## Nivel 3 - HATEOAS

### Hypermedia as the Engine of Application State

- Las respuestas incluyen enlaces hipermedia
- Guían al cliente sobre acciones disponibles
- Los clientes descubren dinámicamente la API

```
1 {
2   "id": "alice",
3   "name": "Alice Smith",
4   "links": {
5     "matches": "/players/alice/matches",
6     "friends": "/players/alice/friends"
7   }
8 }
```



Pocas APIs implementan HATEOAS en la práctica debido a su complejidad



## Formato JSON



## JavaScript Object Notation

### Formato estándar para intercambio de datos

- Ligero y fácil de leer para humanos
- Fácil de parsear para máquinas
- Basado en JavaScript pero independiente del lenguaje

### Estructuras principales:

#### Objetos:

- Colección de pares clave-valor
- Encerrados en {}

```

1 {
2   "nombre": "Alice",
3   "nivel": 15
4 }
```

#### Arrays:

- Lista ordenada de valores
- Encerrados en []

```

1 [
2   "item1",
3   "item2",
4   "item3"
5 ]
```



## Tipos de Valores JSON

### Los valores pueden ser:

- **Cadenas de texto:** "Hola mundo"
- **Números:** 42, 3.14, -10
- **Booleanos:** true, false
- **Null:** null
- **Objetos:** { "key": "value" }
- **Arrays:** [1, 2, 3]



## Ejemplo Completo: Estado de Partida

```

1 {  
2   "game": {  
3     "id": "match_12345",  
4     "type": "pong",  
5     "status": "active",  
6     "players": [  
7       {  
8         "id": "alice",  
9         "name": "Alice Smith",  
10        "score": 5,  
11        "ready": true  
12      },  
13      {  
14        "id": "bob",  
15        "name": "Bob Johnson",  
16        "score": 3,  
17        "ready": true  
18      }  
19    ],  
20    "settings": {  
21      "max_score": 11,  
22      "ball_speed": 300  
23    }  
24  }  
25 }

```



## Usos de JSON

### Más allá de APIs REST:

- Ficheros de configuración
- Almacenamiento de datos en disco
- Bases de datos NoSQL (MongoDB)
- Comunicación entre servicios
- WebSockets y tiempo real



# Principios de Diseño REST



## Todo es un Recurso

**Recurso:** ítem de información identificado por URI única

```
http://api.game.com/players/alice  
http://api.game.com/matches/12345  
http://api.game.com/players/alice/scores  
http://api.game.com/leaderboard
```

**Estructura de URI:**

- Parte fija: dominio y ruta base
- Parte variable: identifica el recurso específico



## Principios de Diseño de URIs

Reglas de buenas prácticas:

Hacer:

- Usar sustantivos, no verbos
  - `/players/alice` ✓
- Usar plurales para colecciones
  - `/players` para lista
  - `/players/alice` para uno
- Crear jerarquías lógicas
  - `/players/alice/matches`

Evitar:

- Verbos en la URI
  - `/getPlayer?id=alice` ✗
- Nombres inconsistentes
  - Mezclar singular/plural
- URIs planas sin relación
- Mayúsculas y guiones bajos



Usar minúsculas y guiones: `/game-sessions/active`



## Métodos HTTP



## Los Cuatro Métodos Principales

### Operaciones CRUD sobre recursos

Método	Operación	Seguro	Idempotente	Cacheable
GET	Obtener	✓	✓	✓
POST	Crear	✗	✗	✗
PUT	Actualizar	✗	✓	✗
DELETE	Eliminar	✗	✓	✗



**Seguro:** no modifica el servidor **Idempotente:** múltiples peticiones = mismo resultado



## GET - Obtener Información

### Características:

- Obtiene información sin modificar el servidor
- Seguro e idempotente
- Cacheable por navegadores y proxies

### Ejemplos:

```
GET /players/alice HTTP/1.1
→ Devuelve información del jugador alice

GET /matches HTTP/1.1
→ Devuelve lista de todas las partidas

GET /leaderboard?limit=10 HTTP/1.1
→ Devuelve top 10 del ranking
```



## POST - Crear Recursos

### Características:

- Crea nuevos recursos en el servidor
- El servidor decide el ID del recurso
- Devuelve el recurso creado en la respuesta
- No es seguro ni idempotente

### Ejemplo:

```
POST /matches HTTP/1.1
Content-Type: application/json

{
  "type": "pong",
  "mode": "ranked",
  "player_id": "alice"
}

→ 201 Created
Location: /matches/67890
```



## PUT - Actualizar Recursos

### Características:

- Actualiza un recurso existente
- Envía el recurso completo (reemplazo)
- No es seguro pero sí idempotente
- Múltiples PUT producen el mismo resultado

### Ejemplo:

```
PUT /players/alice HTTP/1.1
Content-Type: application/json

{
  "name": "Alice Smith",
  "level": 16,
  "score": 9000
}

→ 200 OK
```

Nota

PATCH se usa para actualizaciones parciales (no cubierto aquí)



## DELETE - Eliminar Recursos

### Características:

- Elimina un recurso del servidor
- No es seguro pero sí idempotente
- Eliminar varias veces = mismo resultado (no existe)
- Responde con 204 (sin contenido) o 200

### Ejemplo:

```
DELETE /matches/12345 HTTP/1.1
```

```
→ 204 No Content
```



## Anatomía de Peticiones y Respuestas



## Petición HTTP REST Completa

```
POST /matches HTTP/1.1
Host: api.game.com
Content-Type: application/json
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9...
Content-Length: 89

{
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "player_id": "alice"
}
```

Tres partes:

1. **Línea de petición:** Método, ruta y versión HTTP
2. **Headers:** Metadatos (tipo contenido, auth, etc.)
3. **Body:** Datos JSON enviados



## Headers Comunes en Peticiones

Cabeceras importantes:

- **Content-Type: application/json**
  - Indica que enviamos JSON
- **Accept: application/json**
  - Indica que queremos recibir JSON
- **Authorization: Bearer <token>**
  - Token de autenticación
- **Content-Length: 89**
  - Tamaño del cuerpo en bytes



## Respuesta HTTP REST Completa

```

HTTP/1.1 201 Created
Content-Type: application/json
Location: http://api.game.com/matches/67890
Content-Length: 156

{
  "id": "67890",
  "type": "pong",
  "mode": "ranked",
  "max_score": 11,
  "status": "waiting",
  "created_at": "2025-01-15T14:30:00Z",
  "players": ["alice"]
}

```

Tres partes:

1. **Línea de estado:** Versión HTTP, código y mensaje
2. **Headers:** Metadatos de la respuesta
3. **Body:** Datos JSON del recurso



## Headers Comunes en Respuestas

Cabeceras importantes:

- **Content-Type: application/json**
  - La respuesta es JSON
- **Location: http://api.game.com/matches/67890**
  - URL del recurso creado (en POST)
- **Content-Length: 156**
  - Tamaño de la respuesta en bytes



# Códigos de Estado HTTP



## Categorías de Códigos

Números de tres dígitos que comunican el resultado

Categoría	Significado	Uso
1xx	Informativas	Raramente usados
2xx	Éxito	Operación exitosa
3xx	Redirecciones	Acciones adicionales
4xx	Error del cliente	Error en la petición
5xx	Error del servidor	Fallo del servidor



## 1xx - Respuestas Informativas

Indican que la petición fue recibida y continúa

- Raramente usados en REST
- Más relevante:
  - **101 Switching Protocols**
    - Cambio de HTTP a WebSockets



Veremos el código 101 cuando estudiemos WebSockets



## 2xx - Respuestas Exitosas

La petición fue procesada correctamente

- **200 OK**
  - Éxito general en GET, PUT
  - Respuesta con contenido
- **201 Created**
  - Recurso creado tras POST
  - Debe incluir header [Location](#)
- **204 No Content**
  - Éxito sin contenido
  - Típico en DELETE



## 3xx - Redirecciones

El cliente debe tomar acciones adicionales

- **301 Moved Permanently**
  - Recurso movido permanentemente
  - Nueva ubicación en header `Location`
- **304 Not Modified**
  - Usado con caché
  - El recurso no ha cambiado desde última petición



Las redirecciones ayudan a mantener APIs retrocompatibles



## 4xx - Errores del Cliente

Error en la petición del cliente

- **400 Bad Request:** Petición mal formada
- **401 Unauthorized:** Autenticación requerida
- **403 Forbidden:** Sin permisos (aunque esté autenticado)
- **404 Not Found:** Recurso no existe
- **409 Conflict:** Conflicto con estado actual (ej: usuario ya existe)
- **422 Unprocessable Entity:** Formato correcto pero errores de validación



## 5xx - Errores del Servidor

El servidor no pudo completar la petición

- **500 Internal Server Error**
  - Error genérico del servidor
  - Algo falló en el procesamiento
- **503 Service Unavailable**
  - Servidor temporalmente no disponible
  - Debe incluir [Retry-After](#)
- **504 Gateway Timeout**
  - Timeout esperando respuesta de otro servidor

 **Advertencia**

Los errores 5xx indican problemas del servidor, no del cliente



## Cliente JavaScript con fetch()



## La API fetch()

### API estándar moderna para peticiones HTTP

- Integrada en todos los navegadores modernos
- Utiliza Promises para operaciones asíncronas
- Sintaxis limpia y potente

### Dos formas de manejar respuestas:

#### Callbacks con .then()

```
1 fetch(url)
2   .then(response => response.json())
3   .then(data => console.log(data))
4   .catch(error => console.error(error));
```

#### async/await (recomendado)

```
1 async function getData() {
2   const response = await fetch(url);
3   const data = await response.json();
4   console.log(data);
5 }
```



## async/await: Sintaxis Recomendada

### Código más legible y natural

```
1 async function getPlayer() {
2   try {
3     const response = await fetch('https://api.game.com/players/alice');
4
5     if (!response.ok) {
6       throw new Error('HTTP error ' + response.status);
7     }
8
9     const data = await response.json();
10    console.log(data);
11
12  } catch (error) {
13    console.error('Error:', error);
14  }
15 }
```



Siempre usar try/catch para manejar errores



## GET - Obtener Datos

### Método por defecto de fetch()

```

1 // Obtener un jugador específico
2 async function getPlayer(id) {
3     const response = await fetch(`https://api.game.com/players/${id}`);
4     return await response.json();
5 }
6
7 // Uso
8 const player = await getPlayer('alice');
9 console.log(player.name);
10 console.log(player.level);

```



Template literals ( ` ) permiten insertar variables en strings



## POST - Crear Recurso

### Especificar método, headers y body

```

1 async function createPlayer(data) {
2     const response = await fetch('https://api.game.com/players', {
3         method: 'POST',
4         headers: {
5             'Content-Type': 'application/json'
6         },
7         body: JSON.stringify(data)
8     });
9
10    return await response.json();
11 }
12
13 // Uso
14 const newPlayer = await createPlayer({
15     username: 'charlie',
16     email: 'charlie@example.com'
17 });
18 console.log('Creado con ID:', newPlayer.id);

```



## PUT - Actualizar Recurso

### Reemplazar completamente el recurso

```

1 async function updatePlayer(id, updates) {
2   const response = await fetch(`https://api.game.com/players/${id}`, {
3     method: 'PUT',
4     headers: {
5       'Content-Type': 'application/json'
6     },
7     body: JSON.stringify(updates)
8   });
9
10  return await response.json();
11 }
12
13 // Uso
14 await updatePlayer('charlie', {
15   email: 'newemail@example.com',
16   level: 5
17 });

```

Nota

PUT reemplaza todo el recurso; PATCH actualiza solo campos específicos



## DELETE - Eliminar Recurso

### Método más simple

```

1 async function deletePlayer(id) {
2   const response = await fetch(`https://api.game.com/players/${id}`, {
3     method: 'DELETE'
4   });
5
6   return response.ok;
7 }
8
9 // Uso
10 const eliminado = await deletePlayer('charlie');
11 if (eliminado) {
12   console.log('Jugador eliminado correctamente');
13 }

```

Tip

`response.ok` es `true` para códigos 2xx



## Manejo de Errores y Códigos de Estado

### Manejo robusto con switch

```
1 async function fetchWithErrorHandling(url) {
2   try {
3     const response = await fetch(url);
4
5     switch (response.status) {
6       case 200:
7         return await response.json();
8       case 401:
9         throw new Error('No autenticado');
10      case 403:
11        throw new Error('Sin permisos');
12      case 404:
13        throw new Error('Recurso no encontrado');
14      case 500:
15        throw new Error('Error del servidor');
16      default:
17        throw new Error(`Error: ${response.status}`);
18    }
19  } catch (error) {
20    console.error('Error en petición:', error);
21    throw error;
22  }
23 }
```



## Servidor con Node.js y Express



## Express.js

Framework web minimalista para Node.js

Características:

- Sistema de enrutamiento robusto
- Soporte para middlewares
- Manejo de errores integrado
- Compatibilidad con todos los métodos HTTP
- Estándar de la industria

Instalación:

```
1 npm init -y  
2 npm install express
```



## Configuración de Proyecto

Habilitar módulos ES6:

```
1 {  
2   "name": "mi-api-juego",  
3   "version": "1.0.0",  
4   "type": "module",  
5   "dependencies": {  
6     "express": "^4.18.0"  
7   }  
8 }
```

⚠ Importante

"type": "module" permite usar `import` en lugar de `require`



## Estructura de Proyecto

```
mi-api-juego/
├── node_modules/      # Dependencias (no subir a git)
└── src/
    ├── controllers/   # Lógica de negocio
    ├── routes/         # Definición de rutas
    └── app.js          # Punto de entrada
    └── package.json
    └── package-lock.json
```



## Servidor Básico con Express

```
1 import express from 'express';
2
3 const app = express();
4
5 // Middleware para parsear JSON
6 app.use(express.json());
7
8 // Iniciar servidor
9 app.listen(8080, () => {
10   console.log('Servidor ejecutándose en http://localhost:8080');
11});
```

**! Importante**

`express.json()` es fundamental para leer el body de las peticiones

**Ejecutar:**

```
1 node src/app.js
```



## Controladores

### Funciones con lógica de negocio

- Procesan las peticiones HTTP
- Manipulan datos recibidos
- Devuelven respuestas apropiadas
- Separan lógica del enrutamiento

### Patrón con closures para estado privado:

```

1 const createAnunciosController = () => {
2   // Estado privado
3   const anuncios = [];
4   let nextId = 1;
5
6   const getAll = (req, res) => {
7     res.json(anuncios);
8   };
9
10  return { getAll };
11 };
12
13 export default createAnunciosController;

```



## Controlador Completo - Ejemplo

```

1 // src/controllers/anunciosController.js
2 const createAnunciosController = () => {
3   const anuncios = [];
4   let nextId = 1;
5
6   const getAll = (req, res) => {
7     res.json(anuncios);
8   };
9
10  const create = (req, res) => {
11    const { nombre, asunto, comentario } = req.body;
12
13    const nuevoAnuncio = {
14      id: nextId++,
15      nombre,
16      asunto,
17      comentario
18    };
19
20    anuncios.push(nuevoAnuncio);
21    res.status(201).json(nuevoAnuncio);
22  };
23
24  return { getAll, create };
25 };

```



## Rutas

### Conectan URLs con controladores

```

1 // src/routes/anunciosRoutes.js
2 import express from 'express';
3 import createAnunciosController from '../controllers/anunciosController.js';
4
5 const router = express.Router();
6 const controller = createAnunciosController();
7
8 router.get('/', controller.getAll);
9 router.post('/', controller.create);
10
11 export default router;

```

 Nota

`router.get('/')` se convertirá en `GET /anuncios` con el prefijo



## Integrar Rutas en la Aplicación

```

1 // src/app.js
2 import express from 'express';
3 import anunciosRoutes from './routes/anunciosRoutes.js';
4
5 const app = express();
6 app.use(express.json());
7
8 // Registrar rutas con prefijo
9 app.use('/anuncios', anunciosRoutes);
10
11 app.listen(8080);

```

**Resultado:**

- `GET /anuncios` → obtener todos
- `POST /anuncios` → crear nuevo



## Operación GET - Recurso Específico

```

1 app.get('/anuncios/:id', (req, res) => {
2   const { id } = req.params;
3   const anuncio = anuncios.find(a => a.id === parseInt(id));
4
5   if (!anuncio) {
6     return res.status(404).json({ error: 'No encontrado' });
7   }
8
9   res.json(anuncio);
10 });

```



:id es un parámetro dinámico disponible en req.params.id



## Operación POST - Validación

```

1 app.post('/anuncios', (req, res) => {
2   const { nombre, asunto, comentario } = req.body;
3
4   // Validar datos
5   if (!nombre || !asunto) {
6     return res.status(400).json({
7       error: 'Nombre y asunto son requeridos'
8     });
9   }
10
11  const nuevoAnuncio = {
12    id: Date.now(),
13    nombre,
14    asunto,
15    comentario
16  };
17
18  anuncios.push(nuevoAnuncio);
19  res.status(201).json(nuevoAnuncio);
20 });

```



## Operación PUT

```

1 app.put('/anuncios/:id', (req, res) => {
2   const { id } = req.params;
3   const { nombre, asunto, comentario } = req.body;
4
5   const anuncio = anuncios.find(a => a.id === parseInt(id));
6
7   if (!anuncio) {
8     return res.status(404).json({ error: 'No encontrado' });
9   }
10
11  // Actualizar propiedades
12  anuncio.nombre = nombre;
13  anuncio.asunto = asunto;
14  anuncio.comentario = comentario;
15
16  res.json(anuncio);
17 });

```



## Operación DELETE

```

1 app.delete('/anuncios/:id', (req, res) => {
2   const { id } = req.params;
3   const index = anuncios.findIndex(a => a.id === parseInt(id));
4
5   if (index === -1) {
6     return res.status(404).json({ error: 'No encontrado' });
7   }
8
9   const anuncioEliminado = anuncios.splice(index, 1)[0];
10
11  res.json(anuncioEliminado);
12 });

```

Nota

También se puede devolver 204 (No Content) sin body



## Middlewares

Funciones que procesan peticiones antes del controlador

Usos comunes:

- Logging de peticiones
- Autenticación y autorización
- Validación de datos
- Manejo de errores

Firma:

```
1 (req, res, next) => {
2   // Procesar
3   next(); // Pasar al siguiente middleware/ruta
4 }
```



## Middleware de Logging

```
1 app.use((req, res, next) => {
2   console.log(`${req.method} ${req.path}`);
3   next(); // Importante: pasar control
4 });
```



Olvidar llamar a `next()` bloquea la petición



## Middleware de Manejo de Errores

### Firma especial con 4 parámetros

```
1 app.use((err, req, res, next) => {
2   console.error(err.stack);
3   res.status(500).json({
4     error: 'Error interno del servidor'
5   });
6});
```



Debe colocarse **después** de todas las rutas



## Servir Archivos Estáticos

### Express puede servir HTML, CSS, JS del cliente

```
1 app.use(express.static('public'));
```

#### Estructura:

```
mi-api-juego/
├── public/
│   ├── index.html
│   ├── script.js
│   └── style.css
└── src/
    └── app.js
```

#### Acceso:

- <http://localhost:8080/index.html>
- <http://localhost:8080/script.js>



## Ventajas de la Arquitectura REST



### Separación de Responsabilidades

Componentes independientes:

#### API REST:

- Operaciones de gestión
- Registro y autenticación
- Puntuaciones y estadísticas
- Perfiles y rankings

#### Sistema de Comandos:

- Lógica del juego
- Acciones de jugadores
- Estado del juego



Cada componente puede evolucionar independientemente



## Escalabilidad

### Escalar servicios según necesidades

- Operaciones de gestión (REST) tienen diferentes cargas
- Operaciones de tiempo real (WebSockets) requieren más recursos
- Podemos escalar cada servicio independientemente
- Optimización específica para cada tipo de comunicación



## Testabilidad

### Pruebas independientes:

- Probar API REST sin necesidad del juego completo
- Probar sistema de comandos sin conexión a red
- Herramientas específicas para cada tipo de prueba
- Tests automatizados más simples



## Flexibilidad

La misma API sirve a múltiples clientes:

- Juego web (navegador)
- Aplicación móvil (iOS/Android)
- Herramientas de administración
- Servicios backend



Arquitectura preparada para futuras expansiones



## Resumen



## Conceptos Clave

### API REST:

- Estilo arquitectónico basado en HTTP
- Operaciones CRUD sobre recursos
- URLs bien diseñadas (sustantivos, jerarquías)
- Métodos HTTP correctos (GET, POST, PUT, DELETE)
- Códigos de estado apropiados (2xx, 4xx, 5xx)

### JSON:

- Formato estándar de intercambio
- Ligero y legible
- Objetos y arrays anidados



## Implementación

### Cliente JavaScript:

- API `fetch()` con `async/await`
- Manejo de errores robusto
- Validación de códigos de estado

### Servidor Node.js:

- Express.js como framework
- Controladores con lógica de negocio
- Rutas para organizar endpoints
- Middlewares para funcionalidad transversal



## 16.11. WebSockets

1

# WebSockets

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez   Natalia Madrueño  
ruben.rodriguez@urjc.es   natalia.madrueño@urjc.es  
URJC   URJC

2025-09-09



## Tabla de contenidos

2

- [Introducción a Sockets](#)
- [WebSockets](#)
- [Negociación de Conexión](#)
- [WebSockets en Juegos](#)
- [API WebSocket en JavaScript](#)
- [Servidor WebSocket con Node.js](#)
- [Integración REST + WebSockets](#)
- [Resumen](#)



# Introducción a Sockets



## Concepto Básico de Socket

**Socket:** extremo de comunicación bidireccional

- Identificado por **IP + Puerto**
- Permite comunicación entre dos programas
- Bidireccional: ambos pueden enviar y recibir
- Simultáneo: no hay que esperar turnos

**Ejemplo:**

- Servidor: [212.128.240.50:10000](http://212.128.240.50:10000)
- Cliente: [1XX.XXX.XXX.XX:YYYY](http://1XX.XXX.XXX.XX:YYYY)



## Proceso de Comunicación con Sockets

### Flujo típico:

1. **Servidor** escucha en puerto específico
2. **Cliente** inicia petición de conexión
3. **Servidor** acepta la conexión
4. Se establece **canal bidireccional**
5. Ambos pueden enviar/recibir datos

 **Importante**

No es solicitud-respuesta: ambos pueden iniciar comunicación



## WebSockets



## ¿Qué son los WebSockets?

### Protocolo de comunicación full-duplex

- Canal permanente entre cliente y servidor
- Comunicación **bidireccional** simultánea
- No necesita abrir nueva conexión cada vez
- Ideal para aplicaciones en tiempo real



Full-duplex: ambas partes pueden enviar mensajes independientemente



## Funcionamiento de WebSockets

### Proceso:

1. Cliente solicita conexión al servidor
2. Servidor acepta la conexión
3. Se genera **conexión permanente**
4. Procesos en ambos lados escuchan mensajes



El servidor puede enviar sin que el cliente pregunte constantemente



## WebSockets como Estándar

RFC 6455 sobre socket TCP

Ventajas:

- Usa **puerto 80** (mismo que HTTP)
- Multiplexado: múltiples comunicaciones por puerto
- Evita problemas con firewalls
- Pensado para navegadores y servidores web



Funciona sobre HTTP pero puede usar protocolo propio



## Protocolos WebSocket

Dos esquemas disponibles:

- **ws://** - WebSocket en claro
  - Equivalente a **http://**
- **wss://** - WebSocket seguro (TLS)
  - Equivalente a **https://**

```
1 new WebSocket('ws://example.com/demo');
2 new WebSocket('wss://example.com/demo');
```



## Negociación de Conexión



### Handshake Inicial

La negociación comienza sobre HTTP

Petición del cliente:

```
1 GET /demo HTTP/1.1
2 Host: example.com
3 Connection: Upgrade
4 Upgrade: WebSocket
5 Origin: http://example.com
```

- **Connection: Upgrade** → queremos cambiar de protocolo
- **Upgrade: WebSocket** → especifica WebSocket
- **Origin** → origen de la petición (seguridad)



## Respuesta del Servidor

### Si acepta la conexión:

```
1 HTTP/1.1 101 WebSocket Protocol Handshake
2 Upgrade: WebSocket
3 Connection: Upgrade
4 Sec-WebSocket-Origin: http://example.com
5 Sec-WebSocket-Location: ws://example.com/demo
```

- Código **101**: cambio de protocolo aceptado
- **Sec-WebSocket-Location** → nueva ubicación con `ws://`

 **Importante**

Después del handshake, HTTP se transforma en WebSocket



## Uso Práctico

### La negociación es automática

```
1 // Esto maneja todo el handshake automáticamente
2 const connection = new WebSocket('ws://example.com/demo');
```

 **Tip**

No necesitas implementar manualmente el handshake HTTP



## WebSockets en Juegos



### Ventajas para Juegos en Red

Por qué usar WebSockets:

- **Tiempo real**
  - Actualizaciones instantáneas
  - Sin demoras
- **Bidireccional**
  - Servidor puede enviar eventos
  - Sin polling constante
- **Eficiente**
  - Una conexión persistente
  - Menos recursos
- **Baja latencia**
  - Ideal para respuestas rápidas
  - Crítico en juegos competitivos



## WebSockets vs REST en Juegos

Cuándo usar cada uno:

REST	WebSockets
Registro/autenticación	Movimiento de jugadores
Puntuaciones	Estado del juego
Rankings	Eventos en tiempo real
Perfiles	Chat del juego
Configuración	Notificaciones instantáneas



Ambas tecnologías se complementan en juegos modernos



## API WebSocket en JavaScript



## Crear Conexión

### Instanciar el objeto WebSocket

```
1 const connection = new WebSocket('ws://IP:PUERTO/RUTA');
```

#### Componentes:

- ws:// o wss:// → protocolo
- IP → dirección o dominio del servidor
- PUERTO → puerto donde escucha el servidor
- RUTA → endpoint específico

#### Ejemplo:

```
1 const connection = new WebSocket('ws://127.0.0.1:8080/echo');
```



La conexión se establece automáticamente



## Métodos Principales

### send() - Enviar datos al servidor

```
1 // Texto simple
2 connection.send('Hola servidor');
3
4 // JSON (debe convertirse a string)
5 const data = {
6   type: 'player_move',
7   x: 150,
8   y: 200
9 };
10 connection.send(JSON.stringify(data));
```

### close() - Cerrar conexión

```
1 connection.close();
```



close() inicia cierre ordenado de la conexión



## Event Listeners - onopen

Se ejecuta cuando la conexión se abre

```

1 connection.onopen = function() {
2   console.log('Conectado al servidor');
3
4   // Enviar información inicial
5   connection.send(JSON.stringify({
6     type: 'player_join',
7     username: 'Jugador1'
8   }));
9 };

```



Momento perfecto para enviar mensaje inicial



## Event Listeners - onmessage

Se ejecuta al recibir mensaje del servidor

```

1 connection.onmessage = function(msg) {
2   console.log("Mensaje recibido: " + msg.data);
3
4   // Si es JSON, parsearlo
5   const data = JSON.parse(msg.data);
6
7   // Procesar según tipo
8   if (data.type === 'player_position') {
9     actualizarPosicionJugador(data.playerId, data.x, data.y);
10  } else if (data.type === 'game_over') {
11    mostrarPantallaFinJuego(data.winner);
12  }
13 };

```



El dato real está en `msg.data`



## Event Listeners - onerror

Se ejecuta cuando hay un error

```
1 connection.onerror = function(error) {
2   console.log("Error en WebSocket: ", error);
3
4   // Informar al usuario
5   alert('No se pudo conectar al servidor. Verifica tu conexión.');
6 };
```

Situaciones que disparan error:

- No se puede conectar al servidor
- Problema de red
- Servidor rechaza la conexión



## Event Listeners - onclose

Se ejecuta cuando la conexión se cierra

```
1 connection.onclose = function(event) {
2   console.log('Desconectado del servidor');
3   console.log('Código de cierre:', event.code);
4
5   // Notificar al usuario
6   alert('Se ha perdido la conexión');
7
8   // Intentar reconectar después de 3 segundos
9   setTimeout(function() {
10     connection = new WebSocket('ws://127.0.0.1:8080/echo');
11     configurarListeners(connection);
12   }, 3000);
13 };
```

Razones del cierre:

- Llamamos a `connection.close()`
- El servidor cierra la conexión
- Se pierde la conexión de red



## Ejemplo Completo - Cliente

```

1 // Crear conexión
2 const ws = new WebSocket('ws://127.0.0.1:8080/game');
3
4 // Configurar listeners
5 ws.onopen = function() {
6   console.log('Conectado');
7   ws.send(JSON.stringify({ type: 'join', name: 'Alice' }));
8 };
9
10 ws.onmessage = function(msg) {
11   const data = JSON.parse(msg.data);
12   console.log('Recibido:', data);
13 };
14
15 ws.onerror = function(error) {
16   console.error('Error:', error);
17 };
18
19 ws.onclose = function() {
20   console.log('Desconectado');
21 };
22
23 // Enviar movimiento
24 function moverJugador(x, y) {
25   ws.send(JSON.stringify({ type: 'move', x, y }));

```



## Servidor WebSocket con Node.js



## Librería ws

Implementación robusta y eficiente

Instalación:

```
1 npm init -y
2 npm install ws express
```

Configurar package.json:

```
1 {
2   "name": "websocket-server",
3   "version": "1.0.0",
4   "type": "module",
5   "dependencies": {
6     "ws": "^8.0.0",
7     "express": "^4.18.0"
8   }
9 }
```



## Servidor Básico

```
1 import express from 'express';
2 import { WebSocketServer } from 'ws';
3 import { createServer } from 'http';
4
5 const app = express();
6 const server = createServer(app);
7
8 // Servir archivos estáticos
9 app.use(express.static('public'));
10
11 // Crear servidor WebSocket
12 const wss = new WebSocketServer({ server });
13
14 // Iniciar servidor
15 const PORT = 8080;
16 server.listen(PORT, () => {
17   console.log(`Servidor en http://localhost:${PORT}`);
18 });
```



## Estructura del Servidor

### Componentes:

1. **Servidor HTTP** ( `createServer` )
  - Sirve archivos estáticos
  - Comparte puerto con WebSocket
2. **Express** para archivos estáticos
  - Carpeta `public` para cliente HTML/CSS/JS
3. **WebSocketServer** asociado al servidor HTTP
  - Ambos protocolos en el mismo puerto



## Evento connection

### Se dispara cuando un cliente se conecta

```

1 wss.on('connection', (ws) => {
2   console.log('Nuevo cliente conectado');
3
4   // ws representa la conexión con este cliente específico
5 });

```

Nota

Cada cliente tiene su propio objeto `ws`



## Recibir Mensajes del Cliente

```

1 wss.on('connection', (ws) => {
2   console.log('Nuevo cliente conectado');
3
4   ws.on('message', (message) => {
5     console.log('Mensaje recibido:', message.toString());
6
7     // Si es JSON, parsearlo
8     const data = JSON.parse(message.toString());
9     console.log('Nombre:', data.nombre);
10    console.log('Mensaje:', data.mensaje);
11  });
12 });

```

 **Importante**

Los mensajes llegan como `Buffer`, usar `.toString()`



## Enviar Mensajes al Cliente

**A un cliente específico:**

```

1 ws.on('message', (message) => {
2   const received = message.toString();
3
4   // Enviar respuesta
5   ws.send(`Echo: ${received}`);
6 });

```

**Enviar JSON:**

```

1 ws.on('message', (message) => {
2   const respuesta = {
3     tipo: 'confirmacion',
4     timestamp: new Date().toISOString(),
5     mensaje: 'Recibido correctamente'
6   };
7
8   ws.send(JSON.stringify(respuesta));
9 });

```



## Eventos close y error

Limpiar recursos al desconectar:

```

1 wss.on('connection', (ws) => {
2   console.log('Cliente conectado');
3
4   ws.on('close', () => {
5     console.log('Cliente desconectado');
6     // Limpiar datos asociados a este cliente
7   });
8
9   ws.on('error', (error) => {
10    console.error('Error en la conexión:', error);
11  });
12
13   ws.on('message', (message) => {
14     // Manejar mensajes
15   });
16 });

```



## Ejemplo: Servidor Echo

Devuelve cualquier mensaje recibido

```

1 import express from 'express';
2 import { WebSocketServer } from 'ws';
3 import { createServer } from 'http';
4
5 const app = express();
6 const server = createServer(app);
7
8 app.use(express.static('public'));
9
10 const wss = new WebSocketServer({ server });
11
12 wss.on('connection', (ws) => {
13   console.log('Nuevo cliente');
14
15   ws.on('message', (message) => {
16     const data = message.toString();
17     console.log('Recibido:', data);
18
19     // Echo: devolver al cliente
20     ws.send(`Echo: ${data}`);
21   });
22
23   ws.on('close', () => console.log('Cliente desconectado'));
24   ws.on('error', (error) => console.error('Error:', error));
25 });

```



## Broadcast: Enviar a Todos

### Enviar mensaje a todos los clientes conectados

```

1 wss.on('connection', (ws) => {
2   ws.on('message', (message) => {
3     const data = JSON.parse(message.toString());
4
5     // Enviar a todos los clientes
6     ws.clients.forEach((client) => {
7       if (client.readyState === ws.OPEN) {
8         client.send(JSON.stringify(data));
9       }
10    });
11  });
12 });

```



Verificar `readyState === ws.OPEN` antes de enviar



## Ejemplo: Servidor de Chat

```

1 import express from 'express';
2 import { WebSocketServer } from 'ws';
3 import { createServer } from 'http';
4
5 const app = express();
6 const server = createServer(app);
7 app.use(express.static('public'));
8
9 const wss = new WebSocketServer({ server });
10
11 wss.on('connection', (ws) => {
12   console.log('Cliente conectado al chat');
13
14   ws.on('message', (message) => {
15     try {
16       const datos = JSON.parse(message.toString());
17       console.log(`${datos.nombre}: ${datos.mensaje}`);
18
19       const respuesta = {
20         nombre: datos.nombre,
21         mensaje: datos.mensaje,
22         timestamp: new Date().toISOString()
23       };
24
25       // Broadcast a todos

```



## Gestionar Estado de Clientes

### Asociar datos a cada conexión

```

1 const clientes = new Map();
2
3 wss.on('connection', (ws) => {
4   // Asignar ID único
5   const clientId = Date.now();
6   clientes.set(ws, { id: clientId, nombre: null });
7
8   console.log(`Cliente ${clientId} conectado`);
9
10  ws.on('message', (message) => {
11    const datos = JSON.parse(message.toString());
12
13    // Guardar nombre del cliente
14    const clienteInfo = clientes.get(ws);
15    clienteInfo.nombre = datos.nombre;
16
17    // Procesar mensaje...
18  });
19
20  ws.on('close', () => {
21    const clienteInfo = clientes.get(ws);
22    console.log(`Cliente ${clienteInfo.id} desconectado`);
23    clientes.delete(ws);
24  });
25 });

```



## Integración REST + WebSockets



## Arquitectura Híbrida

### Combinar lo mejor de ambos mundos

```

1 import express from 'express';
2 import { WebSocketServer } from 'ws';
3 import { createServer } from 'http';
4
5 const app = express();
6 const server = createServer(app);
7
8 app.use(express.json());
9 app.use(express.static('public'));
10
11 // Rutas REST
12 app.get('/api/status', (req, res) => {
13   res.json({
14     clientesConectados: wss.clients.size,
15     estado: 'activo'
16   });
17 });
18
19 // Servidor WebSocket
20 const wss = new WebSocketServer({ server });
21
22 wss.on('connection', (ws) => {
23   // Manejar conexiones WebSocket
24 });
25

```



## División de Responsabilidades

### REST para:

- Autenticación
- Registro de usuarios
- Consulta de rankings
- Subida de archivos
- Operaciones puntuales

### WebSocket para:

- Movimiento de jugadores
- Estado del juego en tiempo real
- Notificaciones instantáneas
- Chat del juego
- Eventos en vivo

! Importante

Ambos protocolos comparten servidor y puerto



## Ventajas de Arquitectura Híbrida

Lo mejor de ambos mundos:

- **Simplicidad REST** para operaciones CRUD
- **Tiempo real WebSocket** para interactividad
- **Mismo servidor** simplifica despliegue
- **Mismo puerto** evita problemas de firewall
- **Código organizado** por tipo de comunicación



## Resumen



## Conceptos Clave de WebSockets

### Fundamentos:

- Socket: extremo de comunicación bidireccional
- WebSocket: protocolo full-duplex sobre TCP
- Conexión permanente entre cliente y servidor
- Handshake inicial sobre HTTP (código 101)
- Protocolos `ws://` y `wss://`

### Ventajas:

- Comunicación en tiempo real
- Bidireccional y simultánea
- Eficiente (una conexión persistente)
- Baja latencia para juegos



## API JavaScript

### Objeto WebSocket:

```
1 const ws = new WebSocket('ws://host:port/path');
```

### Métodos:

- `send(data)` - enviar al servidor
- `close()` - cerrar conexión

### Event Listeners:

- `onopen` - conexión establecida
- `onmessage` - mensaje recibido
- `onerror` - error en conexión
- `onclose` - conexión cerrada



## Servidor Node.js

### Librería ws:

```
1 import { WebSocketServer } from 'ws';
2 const wss = new WebSocketServer({ server });
```

### Eventos del servidor:

- `connection` - nuevo cliente conectado
- `message` - mensaje del cliente
- `close` - cliente desconectado
- `error` - error en conexión

### Broadcast:

```
1 wss.clients.forEach(client => {
2   if (client.readyState === ws.OPEN) {
3     client.send(data);
4   }
5});
```



## REST vs WebSockets

### Cuándo usar cada tecnología:

Criterio	REST	WebSockets
<b>Frecuencia</b>	Ocasional	Continua
<b>Latencia</b>	Aceptable	Crítica
<b>Dirección</b>	Cliente inicia	Ambos inician
<b>Casos de uso</b>	CRUD, auth	Tiempo real, chat



En juegos modernos: ambas tecnologías se complementan



# 17. Ejercicios de la asignatura

---

## 17.1. Introducción a Redes

---

### 17.1.1. Investigación del Modelo TCP/IP

Investiga y compara el modelo de capas TCP/IP con tu propia experiencia navegando por Internet.

**Tareas:**

**1. Traza la ruta de una petición web:**

- Ejecuta el comando `traceroute www.google.com` (Linux/Mac) o `tracert www.google.com` (Windows)
- Documenta cuántos saltos (hops) hay hasta llegar al destino
- Identifica si alguno de los routers intermedios pertenece a tu ISP

**2. Análisis por capas:** Para acceder a una página web (ej: <https://www.google.com>), describe qué ocurre en cada capa del modelo TCP/IP:

- **Capa de Aplicación:** ¿Qué protocolo se utiliza?
- **Capa de Transporte:** ¿TCP o UDP? ¿Por qué?
- **Capa de Red:** ¿Qué información contienen los paquetes IP?
- **Capa de Acceso a la Red:** ¿Qué tecnología física usas (WiFi, Ethernet, 4G)?

**3. Encapsulación:**

- Dibuja un diagrama mostrando cómo se encapsulan los datos en cada capa
- Indica qué información añade cada capa (cabeceras)

**4. Reflexión:**

- ¿Por qué crees que el modelo está organizado en capas?
- ¿Qué ventajas tiene esta separación?
- Da un ejemplo de cómo puedes cambiar una capa sin afectar a las demás

## 17.2. Capa de Acceso

---

### 17.2.1. Análisis de Direcciones MAC y ARP

Investiga las direcciones MAC en tu red local y el protocolo ARP.

**Tareas:**

**1. Descubre tu dirección MAC:**

- En Linux/Mac: `ip link show` o `ifconfig`
- En Windows: `ipconfig /all`
- Anota la dirección MAC de tu interfaz de red principal

**2. Analiza la tabla ARP:**

- En Linux/Mac: `arp -a` o `ip neigh show`
- En Windows: `arp -a`
- Documenta:
  - ¿Cuántas entradas tiene tu tabla ARP?
  - ¿Reconoces alguna de las IPs? (router, otros dispositivos)
  - ¿Hay entradas incompletas o que fallan?

**3. Experimenta con ARP:**

- Haz ping a tu router/gateway: `ping 192.168.1.1`
- Vuelve a consultar la tabla ARP
- ¿Qué ha cambiado?

**4. Investigación:**

- ¿Qué pasaría si dos dispositivos en la misma red tuvieran la misma dirección MAC?
- ¿Por qué las direcciones MAC no se usan para enrutamiento en Internet?
- ¿Cuál es la diferencia entre una dirección MAC y una dirección IP en términos de alcance?

**5. Analiza la estructura de la dirección MAC:**

- Tu MAC tiene formato: `XX:XX:XX:YY:YY:YY`
- Los primeros 3 bytes (`XX:XX:XX`) son el OUI (Organizationally Unique Identifier)
- Busca el fabricante de tu tarjeta de red en: <https://maclookup.app/>

## 17.3. Capa de Red

### 17.3.1. Análisis de Direccionamiento IP y Subredes

Dada la siguiente salida del comando `ip addr` en un sistema Linux, analiza la configuración de red y responde las preguntas:

```
$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
        inet6 ::1/128 scope host
            valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
    link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
        inet 192.168.1.100/24 brd 192.168.1.255 scope global eth0
            valid_lft forever preferred_lft forever
        inet6 fe80::5054:ff:fe12:3456/64 scope link
            valid_lft forever preferred_lft forever

3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
    link/ether a4:5e:60:ab:cd:ef brd ff:ff:ff:ff:ff:ff
        inet 10.0.5.42/22 brd 10.0.7.255 scope global wlan0
            valid_lft forever preferred_lft forever
```

#### Preguntas:

##### 1. Direcciones de red:

- La dirección de red
- La dirección de broadcast
- El rango de IPs válidas para hosts
- El número total de hosts disponibles

##### 2. Clasificación:

- ¿Qué interfaces tienen direcciones IP privadas? ¿Por qué?
- ¿La dirección de loopback es pública o privada? ¿Para qué se utiliza?

##### 3. Máscara de subred:

- Convierte las máscaras CIDR (/8, /24, /22) a formato decimal (ej: 255.255.255.0)

- ¿Cuál es la diferencia entre /24 y /22 en términos de número de hosts?

#### 4. Escenario práctico:

- Si quieras enviar un paquete desde esta máquina (192.168.1.100) a un servidor web en Internet (8.8.8.8), ¿qué interfaz se utilizaría?
- ¿Necesitarías NAT? ¿Por qué?

#### 5. MTU:

- ¿Qué significa MTU 1500?
- Si envías un datagrama IP de 2000 bytes por eth0, ¿qué ocurrirá?

## 17.4. Capa de Transporte

---

### 17.4.1. Análisis de Conexiones TCP y UDP

Investiga las conexiones activas en tu sistema y analiza el comportamiento de TCP y UDP.

#### Tareas:

##### 1. Visualiza conexiones activas:

- En Linux/Mac: `netstat -tunap` o `ss -tunap`
- En Windows: `netstat -ano`
- Identifica:
  - ¿Cuántas conexiones TCP tienes activas?
  - ¿Hay conexiones UDP? ¿A qué puertos?
  - ¿Reconoces algún servicio por su puerto? (80=HTTP, 443=HTTPS, 53=DNS, etc.)

##### 2. Analiza estados TCP:

- Busca conexiones en estado ESTABLISHED, LISTEN, TIME\_WAIT
- ¿Qué significa cada estado?
- ¿Por qué hay puertos en estado LISTEN?

##### 3. Compara TCP vs UDP:

Completa la tabla:

Característica	TCP	UDP
Orientado a conexión	?	?
Garantiza entrega	?	?
Garantiza orden	?	?
Control de flujo	?	?
Overhead	?	?
Velocidad	?	?
Ejemplo de uso	?	?

#### 4. Investiga el handshake de TCP:

- Dibuja el proceso de establecimiento de conexión (3-way handshake)
- ¿Qué flags se utilizan? (SYN, ACK)
- ¿Qué ocurre si se pierde uno de los paquetes?

#### 5. Caso práctico: Para un videojuego multijugador, ¿usarías TCP o UDP para los siguientes casos?

- Chat de texto entre jugadores
- Posición de los jugadores en tiempo real
- Transferencia de archivos (mapas, texturas)
- Sistema de login/autenticación

Justifica cada respuesta.

## 17.5. Capa de Aplicación

### 17.5.1. Análisis de Petición HTTP

Investiga cómo funciona el protocolo HTTP usando herramientas de línea de comandos.

**Tareas:**

#### 1. Realiza una petición HTTP simple:

```
curl -v http://example.com
```

- Analiza la salida
- Identifica las cabeceras de petición (Request headers)
- Identifica las cabeceras de respuesta (Response headers)
- ¿Qué código de estado se recibe?

#### 2. Compara HTTP vs HTTPS:

```
curl -v http://google.com
curl -v https://google.com
```

- ¿Qué diferencias observas?
- ¿Hay una redirección en la versión HTTP?
- ¿Qué información adicional aparece en HTTPS?

#### 3. Investiga los métodos HTTP:

Para cada método, describe:

- GET
- POST
- PUT
- DELETE
- ¿Cuándo usarías cada uno en un videojuego?

#### 4. Analiza códigos de estado HTTP:

Agrupa estos códigos por categoría (1xx, 2xx, 3xx, 4xx, 5xx):

- 200, 201, 301, 302, 400, 401, 403, 404, 500, 502, 503
- ¿Qué significa cada categoría?
- Da un ejemplo de cuándo podrías recibir cada código en un juego

#### 5. Investiga DNS:

```
nslookup google.com
```

- ¿Qué IP(s) se resuelven?
- ¿Por qué hay múltiples IPs?
- ¿Qué pasaría si el DNS no funcionara?

## 17.6. JavaScript

---

### 17.6.1. Sistema de Inventario de Videojuego

Crea un sistema de inventario para un videojuego utilizando JavaScript moderno (ES2015+). El sistema debe:

1. Crear una clase `Inventario` que pueda almacenar items con las siguientes propiedades:
  - `nombre` (string)
  - `cantidad` (number)
  - `tipo` (string): "arma", "consumible", o "quest"
2. La clase `Inventario` debe tener los siguientes métodos:
  - `agregarItem(nombre, cantidad, tipo)`: Añade un item al inventario. Si el item ya existe, incrementa su cantidad.
  - `usarItem(nombre, cantidad)`: Reduce la cantidad del item especificado. Si llega a 0, elimina el item.
  - `obtenerItemsPorTipo(tipo)`: Retorna un array con todos los items de un tipo específico.
  - `get totalItems()`: Getter que retorna el número total de items diferentes en el inventario.
3. Implementa validaciones:
  - La cantidad nunca puede ser negativa
  - El tipo debe ser uno de los válidos ("arma", "consumible", "quest")
  - No se puede usar más cantidad de la disponible
4. Guarda el inventario en `localStorage` cada vez que se modifique y permite cargarlo cuando se crea una nueva instancia.

#### Requisitos técnicos:

- Utiliza la sintaxis de clases ES2015+
- Usa getters/setters cuando sea apropiado
- Emplea métodos de arrays modernos (map, filter, etc.)
- Implementa manejo de errores con try-catch
- Usa template literals para mensajes

### Ejemplo de uso:

```
const inventario = new Inventario();
inventario.agregarItem("Espada de Fuego", 1, "arma");
inventario.agregarItem("Poción de Salud", 5, "consumible");
inventario.agregarItem("Llave Antigua", 1, "quest");

console.log(`Total de items: ${inventario.totalItems}`); // 3

inventario.usarItem("Poción de Salud", 2);
const consumibles = inventario.obtenerItemsPorTipo("consumible");
console.log(consumibles); // [{ nombre: "Poción de Salud", cantidad: 3,
    tipo: "consumible" }]
```

## 17.7. Phaser

### 17.7.1. Sistema de Puntuación Persistente

Crea un sistema de puntuación para un juego Phaser que:

1. Muestre la puntuación actual en pantalla durante el juego
2. Guarde la puntuación más alta (high score) en `localStorage`
3. Muestre el high score en la pantalla principal
4. Permita resetear el high score

#### Requisitos:

- Crea una escena de Phaser con:
  - Un texto que muestre "Score: X"
  - Un texto que muestre "High Score: Y"
  - Un botón o tecla para incrementar la puntuación (simular evento del juego)
  - Un botón para resetear el high score
- Implementa la lógica para:
  - Incrementar la puntuación cuando ocurra un evento (ej: presionar SPACE)
  - Comparar la puntuación actual con el high score
  - Actualizar el high score si se supera
  - Persistir el high score en `localStorage`
  - Cargar el high score guardado al iniciar

**Pista:** Utiliza `localStorage.setItem()` y `localStorage.getItem()` para la persistencia.

**Estructura sugerida:**

```
class GameScene extends Phaser.Scene {  
    constructor() {  
        super('GameScene');  
        this.score = 0;  
        this.highScore = 0;  
    }  
  
    create() {  
        // Cargar high score  
        // Crear textos  
        // Configurar controles  
    }  
  
    update() {  
        // Actualizar puntuación  
    }  
}
```

## 17.8. REST - Introducción

### 17.8.1. Diseño de API REST para Videojuego

Diseña una API RESTful para un sistema de gestión de jugadores y partidas de un videojuego.

**Requisitos:**

**1. Define los recursos principales:**

- Jugadores (players)
- Partidas (games)
- Puntuaciones (scores)

**2. Diseña los endpoints siguiendo principios REST:** Para cada operación, especifica:

- Método HTTP (GET, POST, PUT, DELETE)
- Ruta del endpoint
- Código de estado esperado
- Ejemplo de request/response

### **Operaciones requeridas:**

- Listar todos los jugadores
- Obtener un jugador específico
- Crear un nuevo jugador
- Actualizar información de un jugador
- Eliminar un jugador
- Obtener todas las partidas de un jugador
- Crear una nueva partida
- Obtener el top 10 de puntuaciones

### **3. Aplica convenciones RESTful:**

- Usa nombres en plural para recursos
- Usa IDs en la URL para recursos específicos
- Usa query parameters para filtros y paginación
- Diseña URLs jerárquicas para relaciones

### **4. Diseña las estructuras de datos:** Define el formato JSON para:

- Un jugador (Player)
- Una partida (Game)
- Una puntuación (Score)

#### **Ejemplo de formato:**

```
GET /api/players/123
Respuesta: 200 OK
{
  "id": 123,
  "username": "player1",
  ...
}
```

## **17.9. REST - Cliente**

### **17.9.1. Cliente REST con Fetch API**

Implementa un cliente JavaScript que consuma la API REST de un videojuego usando Fetch API.

## **Requisitos:**

- 1. Crea funciones para consumir la API:** Implementa las siguientes funciones usando `fetch()`:

```
async function getPlayers() { ... }
async function getPlayer(id) { ... }
async function createPlayer(playerData) { ... }
async function updatePlayer(id, updates) { ... }
async function deletePlayer(id) { ... }
async function getTopScores(limit = 10) { ... }
```

- 2. Manejo de errores:**

- Verifica el código de estado de la respuesta
- Lanza excepciones personalizadas para errores HTTP
- Maneja errores de red (ej: servidor no disponible)

- 3. Implementa una clase `GameAPI`:** Encapsula todas las funciones en una clase con:

- Base URL configurable
- Métodos para cada operación
- Manejo centralizado de errores
- Headers comunes (Content-Type, Authorization si aplica)

- 4. Crea una interfaz de usuario simple:**

- Botón para obtener el top 10 de puntuaciones
- Mostrar los resultados en una lista HTML
- Indicador de carga durante la petición
- Mensaje de error si falla

## **Estructura sugerida:**

```
class GameAPI {  
    constructor(baseURL) {  
        this.baseURL = baseURL;  
    }  
  
    async request(endpoint, options = {}) {  
        // Método helper para hacer peticiones  
    }  
  
    async getPlayers() { ... }  
    async createPlayer(data) { ... }  
    // ... más métodos  
}
```

#### Uso esperado:

```
const api = new GameAPI('https://api.mygame.com');  
  
// Obtener jugador  
const player = await api.getPlayer(123);  
console.log(player.username);  
  
// Crear nuevo jugador  
const newUser = await api.createPlayer({  
    username: 'newbie',  
    email: 'newbie@example.com'  
});
```

## 17.10. REST - Servidor

### 17.10.1. Implementar un Servidor REST con Express

Crea un servidor REST básico para gestionar jugadores de un videojuego usando Node.js y Express.

#### Requisitos:

##### 1. Configura un proyecto Express:

```
npm init -y  
npm install express
```

## 2. Implementa los siguientes endpoints:

- GET /api/players - Listar todos los jugadores
- GET /api/players/:id - Obtener un jugador por ID
- POST /api/players - Crear un nuevo jugador
- PUT /api/players/:id - Actualizar un jugador
- DELETE /api/players/:id - Eliminar un jugador

## 3. Almacenamiento en memoria:

- Usa un array para almacenar los jugadores temporalmente
- Cada jugador debe tener: id, username, level, score

## 4. Validaciones:

- Verificar que username no esté vacío
- Verificar que level y score sean números
- Retornar error 400 si los datos son inválidos
- Retornar error 404 si el jugador no existe

## 5. Middleware:

- Usa express.json() para parsear JSON
- Añade un middleware de logging que imprima cada petición

### Estructura sugerida:

```
const express = require('express');
const app = express();

// Datos en memoria
let players = [
    { id: 1, username: "player1", level: 5, score: 1000 },
    { id: 2, username: "player2", level: 3, score: 500 }
];
let nextId = 3;

// Middleware
app.use(express.json());

// TODO: Implementar endpoints

app.listen(3000, () => {
    console.log('Server running on http://localhost:3000');
});
```

**Bonus:**

- Añade un endpoint `GET /api/players?level=X` para filtrar por nivel
- Implementa paginación con `limit` y `offset`

## 17.11. WebSockets - Introducción

---

### 17.11.1. Comparación REST vs WebSockets

Analiza las diferencias entre REST y WebSockets y determina cuándo usar cada uno.

**Tareas:****1. Completa la tabla comparativa:**

Aspecto	REST (HTTP)	WebSockets
Protocolo base	?	?
Conexión	?	?
Comunicación	? (cliente→servidor)	?
Overhead	?	?
Tiempo real	?	?
Escalabilidad	?	?
Complejidad	?	?

**2. Casos de uso - ¿REST o WebSocket?** Para cada escenario de videojuego, indica si usarías REST, WebSocket o ambos, y justifica:

1. Chat entre jugadores en tiempo real
2. Cargar lista de mapas disponibles
3. Sincronizar posición de jugadores en un shooter multijugador
4. Subir una puntuación al leaderboard
5. Notificaciones de amigos que se conectan
6. Descargar un mapa de 50MB
7. Partida de ajedrez online por turnos

## 8. Sistema de login/autenticación

### 3. Investigación:

- ¿Qué es el “handshake” de WebSocket?
- ¿Por qué WebSocket usa HTTP inicialmente?
- ¿Pueden atravesar WebSockets un firewall/proxy más fácilmente que un socket TCP puro?

### 4. Diseño arquitectónico: Para un juego multijugador tipo battle royale:

- ¿Qué información enviarías por REST?
- ¿Qué información enviarías por WebSocket?
- Dibuja un diagrama simple de la arquitectura

## 17.12. WebSockets - Cliente

---

### 17.12.1. Cliente WebSocket para Chat de Juego

Implementa un cliente WebSocket simple para un sistema de chat en tiempo real de un videojuego.

#### Requisitos:

##### 1. Crea una clase `GameChat` :

- Constructor que reciba la URL del servidor WebSocket
- Método `connect()` para establecer conexión
- Método `sendMessage(message)` para enviar mensajes
- Método `disconnect()` para cerrar la conexión
- Callbacks para eventos: `onMessage` , `onConnect` , `onDisconnect` , `onError`

##### 2. Manejo de estados de conexión:

- Mostrar indicador cuando se está conectando
- Mostrar indicador cuando está conectado
- Mostrar error si falla la conexión
- Intentar reconnectar automáticamente si se pierde la conexión

### 3. Interfaz de usuario HTML:

- Input para el nombre de usuario
- Botón para conectar
- Área de mensajes (mostrar historial)
- Input para escribir mensaje
- Botón para enviar mensaje
- Indicador de estado de conexión

### 4. Formato de mensajes JSON:

```
// Cliente → Servidor (enviar mensaje)
{
  type: "message",
  username: "player1",
  text: "Hola a todos"
}

// Servidor → Cliente (mensaje broadcast)
{
  type: "message",
  username: "player2",
  text: "Hola player1!",
  timestamp: "2025-11-06T10:30:00Z"
}

// Servidor → Cliente (usuario conectado)
{
  type: "user_joined",
  username: "player3"
}
```

**Estructura sugerida:**

```

class GameChat {
    constructor(serverURL) {
        this.serverURL = serverURL;
        this.ws = null;
        this.username = null;

        // Callbacks
        this.onMessage = null;
        this.onConnect = null;
        this.onDisconnect = null;
        this.onError = null;
    }

    connect(username) {
        // TODO: Establecer conexión WebSocket
    }

    sendMessage(text) {
        // TODO: Enviar mensaje al servidor
    }

    disconnect() {
        // TODO: Cerrar conexión
    }
}

```

**Bonus:**

- Mostrar quién está escribiendo ("player2 is typing...")
- Soporte para comandos especiales ("/help", "/users")
- Emoticones o formato básico de texto

## 17.13. WebSockets - Servidor

### 17.13.1. Servidor WebSocket para Chat de Juego

Implementa un servidor WebSocket simple usando Node.js y la librería `ws`.

**Requisitos:**

1. Instala las dependencias:

```
npm init -y  
npm install ws
```

## 2. Implementa el servidor WebSocket:

- Servidor escuchando en el puerto 8080
- Manejar conexiones de múltiples clientes simultáneamente
- Broadcast de mensajes a todos los clientes conectados
- Gestionar usuarios (username por cliente)

## 3. Tipos de mensajes:

- register : Cliente envía su username al conectar
- message : Cliente envía mensaje de chat
- user\_joined : Servidor notifica cuando alguien se une
- user\_left : Servidor notifica cuando alguien se va

## 4. Funcionalidades:

- Almacenar usuarios conectados con su websocket y username
- Broadcast a todos excepto al emisor
- Notificar a todos cuando un usuario se conecta/desconecta
- Manejo de errores y desconexiones

### Estructura sugerida:

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

// Almacenar clientes conectados
const clients = new Map() // ws -> {username, ws}

wss.on('connection', (ws) => {
    console.log('New client connected');

    ws.on('message', (message) => {
        // TODO: Manejar mensajes
    });

    ws.on('close', () => {
        // TODO: Manejar desconexión
    });
});
```

#### Bonus:

- Logging con timestamp de cada evento
- Comando “/users” para listar usuarios conectados
- Límite máximo de clientes simultáneos
- Heartbeat/ping para detectar conexiones muertas

# 18. Software utilizado en la asignatura

---

## 18.1. Visualizaciones

**Ubicación:** [jergames.dslabapps.es](http://jergames.dslabapps.es)

**Descripción:** Página web con visualizaciones interactivas que incluye:

- Juego del Router: Simula protocolos de enrutamiento y reenvío de paquetes
- CIDR y Coincidencia de Prefijo Más Largo: Comprende el enmascaramiento de subredes y las decisiones de enrutamiento
- Distribución Estática de Ancho de Banda TCP: Visualiza el reparto de ancho de banda entre conexiones TCP
- Distribución Dinámica de Ancho de Banda TCP: Simulación interactiva de asignación de ancho de banda

**Identificador SWH:**

```
swh:1:dir:e759621272f530009eb589570687a2d56d0d4bcd;
origin=https://github.com/rrunix/jer-visualizations;
visit=swh:1:snp:486873193776051cae0e315d0a88a076d0ce242;
anchor=swh:1:rev:3f99cd855ca1298ff61620308ba51d1dd37b7a41
```

**Enlace permanente:** [Software Heritage Archive](#)

- 
1. 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. ↩
  2. Algunos componentes de Internet tienen una arquitectura híbrida, como los ISPs grandes y DNS. ↩
  3. Los routers comparten y propagan información de congestión y destinos disponibles con los routers adyacentes. ↩
  4. Técnicamente sí pueden afectar ciertos factores, por ejemplo, campos magnéticos intensos o interferencias electromagnéticas sobre el cable Ethernet. ↩

5. 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. ↵
6. 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. ↵
7. 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. ↵
8. Las direcciones 0.0.0.0/8 y 127.0.0.0/8 están reservadas para funciones especiales. ↵
9. 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. ↵
10. En realidad son para sistemas distribuidos. Pero las aplicaciones de red son inherentemente sistemas distribuidos. ↵