

# Capa de Aplicación

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez Natalia Madrueño

[ruben.rodriguez@urjc.es](mailto:ruben.rodriguez@urjc.es)

URJC

[natalia.madrueno@urjc.es](mailto:natalia.madrueno@urjc.es)

URJC

2025-09-09



# Tabla de contenidos

- [Introducción](#)
- [Sockets](#)
- [Arquitecturas de Aplicaciones Distribuidas](#)
- [Protocolos de Aplicación](#)
- [Servicios](#)
- [Resumen](#)

# Introducción

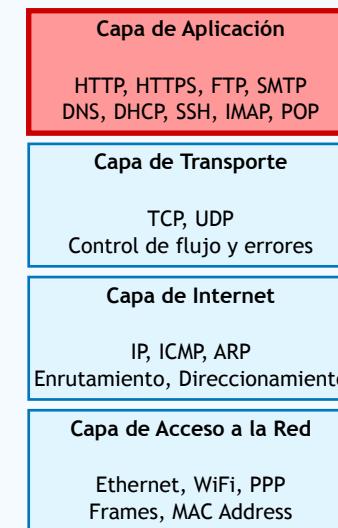
# ¿Qué es la Capa de Aplicación?

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:



### ! Importante

**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()
```



Tip

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   |



Tip

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)

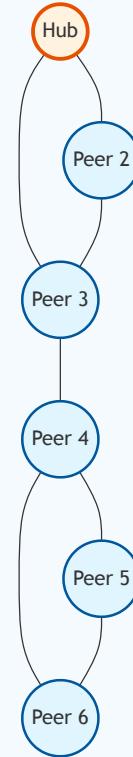
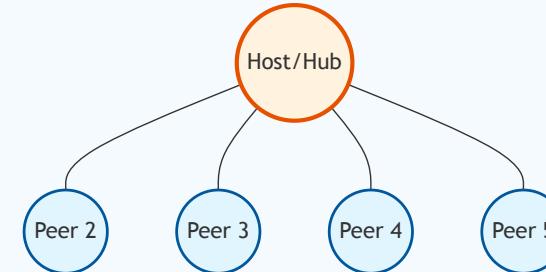
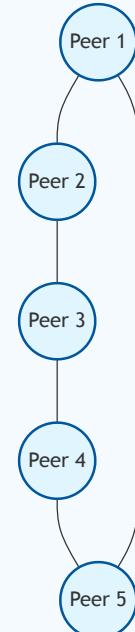
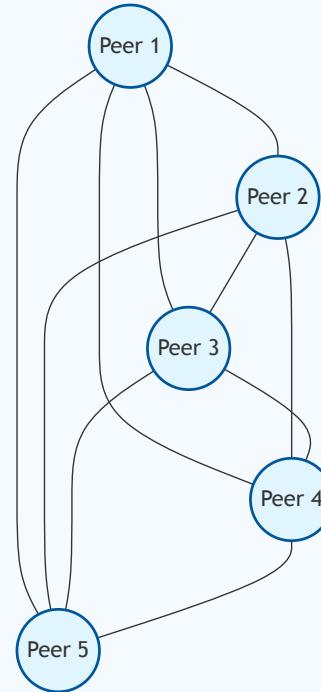
### Aplicaciones comunes:

- 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>

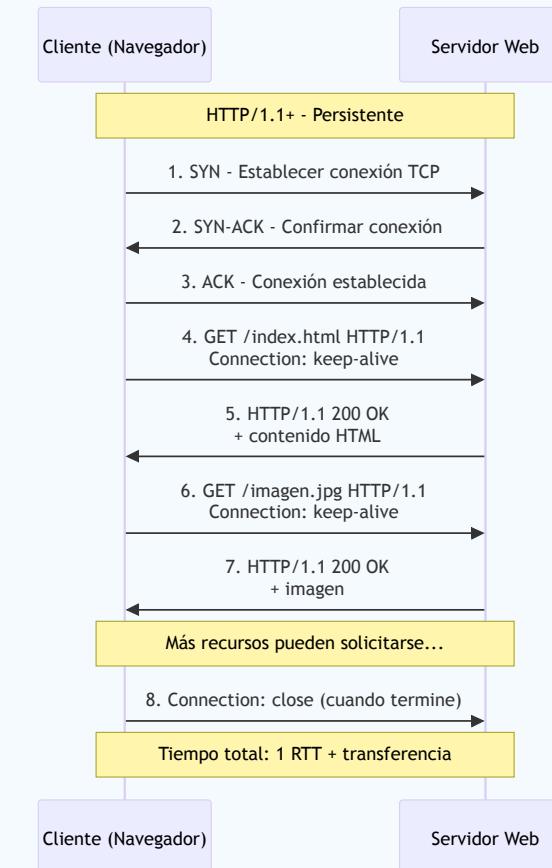
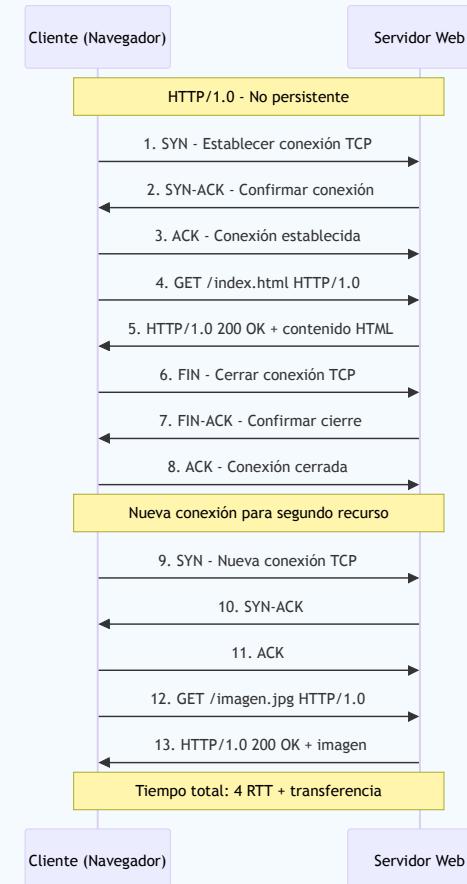
### Verbos HTTP:

- **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

### Tipos:

- **Propias:** De la web navegada
- **Terceros:** Servicios externos
- **Permanentes:** Sin expiración
- **Sesión:** Expiran al cerrar

### Usos principales:

- Mantener sesiones
- Personalización
- Análisis de uso
- Publicidad dirigida

### 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:

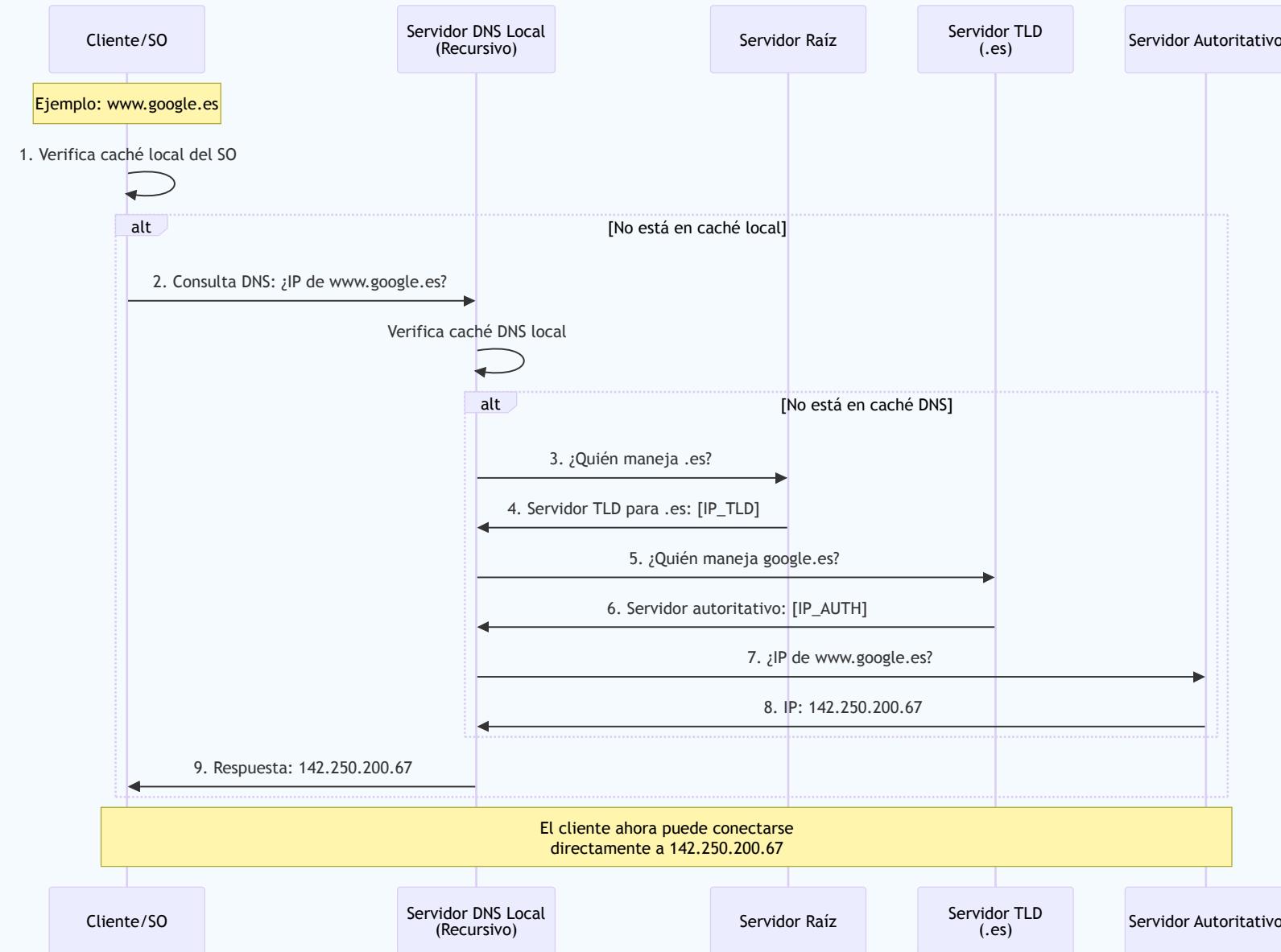
1. **Servidores raíz:** 13 lógicos (A-M)
2. **Servidores TLD:** .com, .org, .es
3. **Servidores autoritativos:** Info definitiva
4. **Servidores locales:** Recursivos/resolvers

### 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

### Ubicación típica:

### 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

# 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