

# Telecomunicaciones y Sistemas Distribuidos

Estas son las notas de los cursos "*Telecomunicaciones y Sistemas Distribuidos*" y "*Redes y Telecomunicaciones*" de la Licenciatura y el Profesorado en Ciencias de la Computación, respectivamente, dictadas por el Departamento de Computación - FCEFQyN, de la Universidad Nacional de Río Cuarto.

Estas notas contienen una guía principal de los temas estudiados en el curso y no pretender contener todos los detalles de cada tema abordado. También se incluyen las presentaciones correspondientes a cada clase.

Estas notas incluyen los temas de la primera parte del curso y se recomienda el libro en línea Computer Networks: A System Approach, para profundizar en algunos detalles.

En la segunda parte de estas notas se abordan temas de sistemas distribuidos, es decir, algoritmos y protocolos para implementar sistemas sin una coordinación central, permitiendo lograr sistemas tolerantes a fallas y con mayor escalabilidad.

## Equipo docente

- Marcelo Arroyo
- Gastón Scilingo

---

Próximo >

**Introducción**

# Introducción

Este curso incluye el estudio de los siguientes temas:

- Conceptos básicos de telecomunicaciones
  - Dispositivos y medios físicos de comunicación
- Redes de computadoras
- Protocolos de comunicación
- Software (diseño, APIs y frameworks)
- Sistemas distribuidos vs centralizados

## Un poco de historia

Las computadoras de la década de 1950 generalmente conectaban dispositivos remotos como las terminales o consolas. La comunicación digital se basaba en la transmisión de caracteres (bytes). Para eso se desarrollaron dispositivos y protocolos de comunicación en serie [RS-232](#) y paralelos como el [parallel port](#) de la IBM-PC.

En 1960 se comenzó a investigar y experimentar con la transmisión de datos en grupos o *paquetes*. La idea es que cada paquete podía transmitirse como una unidad independiente y eventualmente seguir diferentes rutas para alcanzar su destino. Esta técnica se conoce como *packet switching* y se diferencia con la otra forma de transmitir datos como una secuencia de bits de longitud desconocida a priori (*data stream*).

En 1972 se desarrolla [ARPANET](#), una red en USA que interconectaba computadoras de entidades académicas y gubernamentales. En 1983 migra sus protocolos a la suite [TCP/IP](#), conocida como la familia de protocolos de Internet.

En 1972 se desarrolla [Ethernet](#), un protocolo a nivel de enlace y físico para conectar computadoras en una red local.

Con la aparición de ARPANET se desarrollan aplicaciones en red como

- Correo electrónico (SMTP, POP, IMAP, ...)
- Login remoto (telnet, ...)

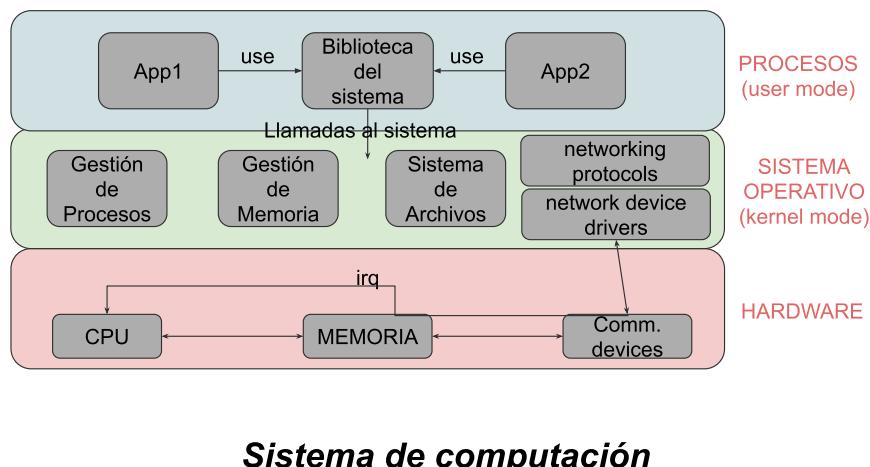
- Transferencia de archivos (FTP, ...)
- otras...

En 1990 Tim Berners-Lee desarrolla la World Wide Web.

## Hardware y software de red

Los sistemas operativos modernos generalmente incluyen soporte (*device drivers*) para dispositivos de comunicaciones y software de implementación de protocolos de comunicación.

La siguiente figura muestra la arquitectura de un sistema.



*Figura 1: Arquitectura del sistema.*

Un *Protocolo de comunicación* describe:

1. Los *roles* de los nodos (*peers*)
2. El formato de los *mensajes*
3. La *coreografía* de la comunicación

Por ejemplo, un protocolo del tipo *request/reply* define roles de *clientes*, quienes inician la comunicación mediante el envío de un requerimiento al *servidor*. El servidor es un proceso que está esperando pasivamente requerimientos. Ante la recepción de un requerimiento el servidor, lo procesa y envía al cliente un mensaje de respuesta.

En este caso, la *coreografía* es simple: Una comunicación es iniciada por un cliente y éste debe quedar a la espera de una respuesta del servidor. Luego puede iniciar otro ciclo de comunicación.

Los protocolos de comunicación se especifican en *Comments for Requests (RFCs)*, los cuales son documentos de texto administrados por la [IETF](#). Cada documento tiene asociado su identificador (número).

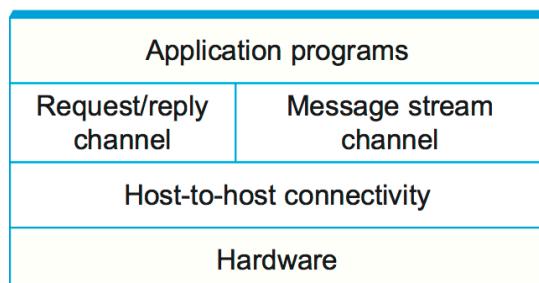
## Diseño de protocolos

Los protocolos pueden clasificarse en dos grandes grupos:

- *Orientados a conexión*: Antes de intercambiar mensajes las partes deben establecer una conexión. Una *conexión* determina las direcciones de origen y destino. Conceptualmente es similar a una llamada telefónica. Cada mensaje se asocia a la conexión.
- *Orientado a datagramas*: Los mensajes se envían de manera independiente. Conceptualmente es similar al correo postal. Cada mensaje incluye su dirección de origen y destino.

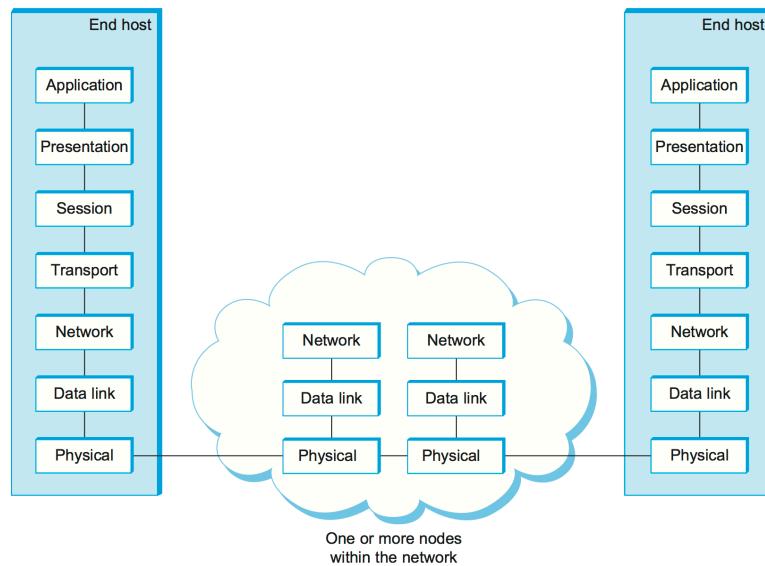
Uno de los objetivos principales de un subsistema de redes es su *heterogeneidad* de los participantes los cuales pueden tener diferentes tipos de software y hardware.

Esto motiva que el sistema debe diseñarse basado en *capas* de servicios. Esto permite independizar subsistemas y *componer* un conjunto de protocolos para el desarrollo de diferentes tipos de aplicaciones.

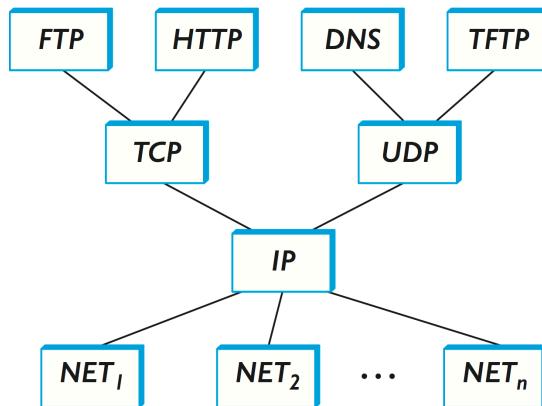


**Figura 2: Diseño en capas.**

El modelo **OSI** propone el siguiente diseño en 7 capas, como se muestra en la [figura 3](#). Cada capa ofrece una API de servicios a las capas superiores y puede usar las APIs de las inferiores.

**Figura 3: Modelo OSI.**

Los protocolos de Internet (familia TCP/IP) se basan en un modelo simplificado del modelo OSI, como se muestra en la [figura 4](#).

**Figura 4: Modelo TCP/IP (4 capas).**

## Protocolo de comunicación

Cada protocolo opera en alguna capa del modelo OSI. Un protocolo define los siguientes elementos:

1. Identificación de las partes intervenientes en una comunicación.
2. Formato de mensajes y presentación (o representación).

3. Coreografía o reglas de la comunicación. Incluyendo los *roles* de los participantes.
4. Mecanismos de detección o corrección de errores de comunicación
5. Mapeo de direcciones con otros protocolos.
6. Mecanismos de ruteo o multiplexado.
7. Mecanismos de detección y prevención de congestión.

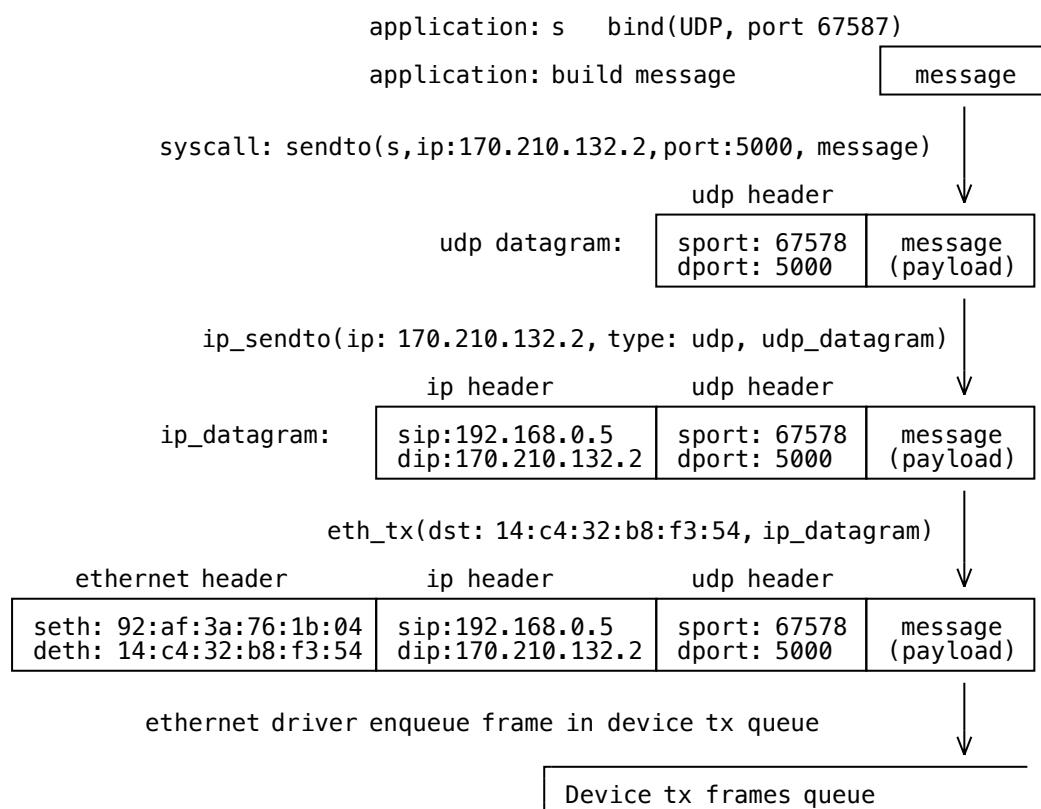
El formato de los mensajes puede estar basado en texto (strings) como en [HTTP](#), binario como en IP o TCP, o ambos como en [File Transfer Protocol\(FTP\)](#).

Cada protocolo debe diseñarse de tal forma para minimizar el acoplamiento con los demás protocolos. Su implementación debe definir una interfaz de servicios a las capas superiores y usar los servicios provistos por las capas inferiores.

Las reglas de la comunicación definen el orden en que deben ser enviados los mensajes y los roles de las entidades que intervienen.

A modo de ejemplo, un protocolo del tipo *requerimiento-respuesta* (como *HTTP*) define *roles*, un *cliente* y un *servidor*. El cliente envía un mensaje de solicitud de un servicio y el servidor responde. El servidor inicia en forma pasiva. La comunicación es siempre iniciada por el cliente.

## Encapsulamiento y multiplexado de mensajes



### Figura 5: Encapsulado en una transmisión.

La figura anterior muestra cómo se van añadiendo las cabeceras correspondientes a cada protocolo que interviene en una transmisión de un mensaje. En éste caso se trata de una aplicación que envía un mensaje a un destino determinado (host con dirección IP 170.210.132.2 y a un proceso dentro de ese host asociado al puerto 5000). Esto muestra que en un protocolo de red (como IP) sus entidades son *hosts* (dispositivos) donde cada uno se identifica por su *dirección de red* mientras que en un protocolo de transporte (como *UDP*) sus entidades son *procesos* dentro de hosts, identificados por *puertos* (números de 16 bits).

A medida que una capa solicita un servicio de transmisión a una capa inferior, ésta última adiciona su propio encabezado e incluye lo anterior como su *payload* (datos).

Cuando el host destino del mensaje recibe el frame, lo recibirá por un dispositivo de red (Ethernet, USB o otro) y deberá realizar el proceso inverso para entregarlo (*deliver*) al proceso (destinatario) correspondiente.

En la recepción, cada capa verifica su header, controla errores y determina a qué capa superior se lo debe entregar. Cada header tiene un campo *protocol* o *type* que identifica el *payload type* y determina a qué protocolo de la capa superior se debe entregar.

Por ejemplo, UDP determina a qué proceso (aplicación) debe entregarle el mensaje usando el número de puerto al que está asociado (*bind*). Esto se conoce como *multiplexado* ya que todos los paquetes recibidos comúnmente encolados por el protocolo de red IP deberán *rutearse* al proceso destinatario correspondiente dentro del host. En este caso, UDP lo puede implementar usando una *tabla de bindings* con tuplas de la forma (*process\_id, port*).

Un *port* es un concepto usado por los protocolos de transporte como UDP o TCP para identificar a un proceso en un *host* de forma independiente de la plataforma.

## Métricas para comunicaciones

**Tasa de transmisión (data rate):** Número de bits transmitidos por unidad de tiempo.

Ejemplo: 10Mbps = 10000000 bits por segundo.

**Ancho de banda (bandwidth):** Rango de frecuencias usado en un medio (cable, aire, ...)

A veces estos términos se usan de manera invertida. El *ancho de banda* es una propiedad física del medio de transmisión, en cambio la *tasa de transmisión* puede aumentarse/disminuirse con un mismo ancho de banda, usando diferentes técnicas de modulación o cambiando la duración de un bit (*tiempo de bit*).

**Latencia:** Tiempo que demora un mensaje desde su transmisión hasta su arribo en el receptor.  $Latencia = T_p + T_x + T_q$ , donde

$T_p = distance/speed$ : Es el *tiempo de propagación de la señal*

$T_x = size(msg)/bandwidth$ : Tiempo de transmisión del mensaje

$T_q$  es el tiempo de *encolado (store and forward)* por los *routers intermedios*.

Otra medida de la calidad de una comunicación es la *variación de la latencia de paquetes o jitter*, como se muestra en la siguiente figura.

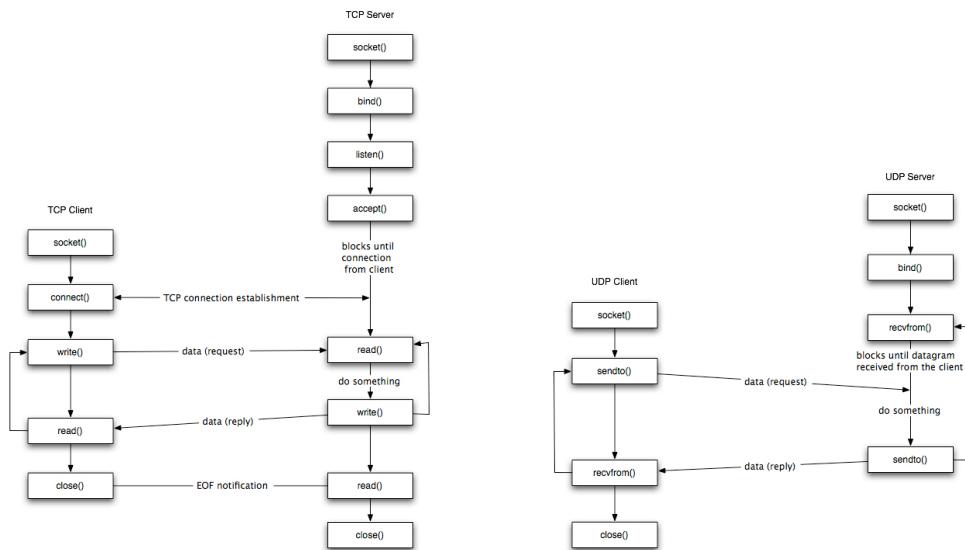


*Figura 6: Variación en el arribo de paquetes en el receptor.*

## Software

Cada protocolo provee una interface de programación (API). Una de las primeras APIs provistas que aún es ampliamente utilizada es Berkeley sockets, introducida como *syscalls* en 4.2BSD Unix en 1983.

Soporta varios protocolos, incluyendo la familia de internet y soporta protocolos orientados a conexión y orientado a datagramas.



**Figura 7: a) Orientado a conexión. b) Orientado a datagramas.**

A continuación se muestra código de ejemplo de dos programas (cliente y servidor) usando la API *sockets*. Los programas usan el protocolo **TCP** (orientado a conexión).

clike

```
/*
 * file: tcp-client.c
 * compile: gcc -o tcp-client tcp-client.c
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 8000
#define MAX_LINE 256

int main(int argc, char * argv[])
{
    FILE *fp;
    struct hostent *hp;
    struct sockaddr_in sin; // server internet address
    char *host;
```

```
char buf[MAX_LINE];
int s;
int len;

if (argc==2) {
    host = argv[1];
}
else {
    fprintf(stderr, "usage: simple-talk host\n");
    exit(1);
}

/* translate host name into peer's IP address */
hp = gethostbyname(host);
if (!hp) {
    fprintf(stderr, "simple-talk: unknown host: %s\n", host);
    exit(1);
}

/* build server address data structure */
bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET;
bcopy(hp->h_addr, (char *)&sin.sin_addr, hp->h_length);
sin.sin_port = htons(SERVER_PORT);

/* active open */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simple-talk: socket");
    exit(1);
}
/* connect to server */
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
    perror("simple-talk: connect");
    close(s);
    exit(1);
}
/* main loop: get and send lines of text */
while (fgets(buf, sizeof(buf), stdin)) {
    len = strlen(buf);
    buf[len] = 0;
    send(s, buf, len, 0);
    recv(s, buf, len, 0);
}
```

```

        printf("recv: %s", buf);
    }
}

```

### *Listado 1: Cliente TCP.*

A continuación se muestra el código del servidor.

```

clike

/* file: tcp-server.c
 * compile: gcc -o tcp-server tcp-server.c
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#define SERVER_PORT 8000
#define MAX_PENDING 5
#define MAX_LINE 256

/* string to uppercase */
void toupperstr(char *str)
{
    while (*str != 0 && *str != '\n') {
        *str = toupper(*str);
        str++;
    }
}

int main()
{
    struct sockaddr_in sin;           // server internet address
    char buf[MAX_LINE];              // data buffer
    int buf_len, addr_len;
    int s, new_s;                    // socket descriptors

```

```

char addr_str[INET_ADDRSTRLEN];

/* build server address data structure */
bzero((char *)&sin, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(SERVER_PORT);

/* setup passive open */
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    perror("simple-talk: socket");
    exit(1);
}
if ((bind(s, (struct sockaddr *)&sin, sizeof(sin))) < 0) {
    perror("simple-talk: bind");
    exit(1);
}
listen(s, MAX_PENDING);

/* wait for connection, then receive and print text */
while(1) {
    if ((new_s = accept(s, (struct sockaddr *)&sin, &addr_len)) < 0)
        perror("simple-talk: accept");
    exit(1);
    inet_ntop(AF_INET, &sin.sin_addr, addr_str, sizeof(addr_str))
    printf("Connection: IP: %s port: %d\n", addr_str, ntohs(sin.
    while (buf_len = recv(new_s, buf, sizeof(buf), 0)) {
        printf("server. Received: %s", buf);
        toupperstr(buf);
        send(new_s, buf, buf_len, 0),
    }
    close(new_s);
}
}

```

### *Listado 2: Servidor TCP iterativo.*

En éste caso el *socket new\_s* representa el *socket conectado a un cliente*, mientras que el *socket s* está asociado (*binded*) sólo al extremo del servidor (*half-end*).

Cada mensaje enviado incluye la información *de la conexión al que pertenece*.

Este servidor es *iterativo*, es decir que está en un ciclo (*infinito*) que:

1. Espera por una conexión de un cliente.
2. Espera mensajes, procesa y responde.
3. Cierra la conexión.

Para soportar requerimientos concurrentes (desde diferentes clientes), es posible crear *procesos hijos* o *threads* asignándolos a cada nueva conexión o requerimiento. Un servidor de este tipo se denomina *servidor concurrente*.

En UNIX un *socket* es una abstracción de un *archivo*. En el caso de una comunicación orientada a conexión, pueden usarse las llamadas al sistema `read(s,buf,count)` y `write(s,buf,count)` en lugar de `recv()` y `send()`.

Así es posible reusar funciones que leen/escriben de/a archivos para que lean/escriban de/a sockets.

---

< Anterior

**Inicio**

Próximo >

**Enlaces directos**

# Enlaces directos

En este capítulo se analizan los medios físicos utilizados en enlaces directos (de computadora a computadora u otros dispositivos) y técnicas de transmisión de señales.

Para lograr esto hay que resolver al menos los siguientes problemas:

1. *Conversión analógica-digital*
2. *Codificación* de los bits o símbolos a transmitir/recibir
3. *Transmisión de las señales*
4. *Empaquetado (framing)* de secuencias de bits
5. *Detección de errores* en la recepción
6. *Compresión de datos*. Los veremos en el capítulo [presentación](#).
7. *Acceso al medio*, en el caso de usar un medio compartido como un cable (bus) o aire

Actualmente muchas de estas funciones se implementan en una combinación de software y dispositivos de hardware de comunicaciones (chips Ethernet, wifi, RS-232, USB, ...).

## Conversión analógica-digital

Una señal de entrada analógica  $f(t)$ , por ejemplo una señal generada por un micrófono, puede convertirse en una digital o discreta, denotada como  $d[i]$  donde el dominio y rango de  $d$  son discretos.

El proceso consiste en los siguientes pasos:

1. *Muestreo (sampling)*: Captura de las amplitudes de la señal en ciertos puntos en el tiempo. El intervalo de tiempo entre las muestras tomadas determina la *frecuencia de muestreo*.
2. *Cuantización o discretización*: Discretización de los valores obtenidos en el paso anterior. Comúnmente se mapea un conjunto finito de intervalos continuos a un conjunto finito de valores discretos.

Esto resulta en una señal digital o *función escalonada*  $d[i]$ .

## Codificación

Nuestro interés se centra en la comunicación de *señales digitales*, es decir la transmisión de un conjunto finito de símbolos, como por ejemplo el conjunto 0, 1 (binario).

En primer lugar hay que determinar cómo se representa o *codifica* una secuencia de bits a transmitir.

Los bits se deben representar de forma tal que sean distinguibles por un receptor y deben tener una cierta duración en el tiempo que se conoce como *tiempo de bit*. Es común que en un circuito electrónico se use un reloj (oscilador) usado por el transmisor (*Tx*) y el receptor (*Rx*). Cada *pulso* (*tick*) del reloj indica el comienzo del próximo bit.

En la transmisión de datos digitales por pulsos (señales cuadradas), para distinguir los símbolos digitales transmitidos se usa comúnmente *modulación por amplitud*. Por ejemplo: Un 1 se representa con un *pulso* de un voltaje (*amplitud*) de 5V y un 0 por un *pulso de 0V o -5V*.

La *duración del pulso* determina la *frecuencia de muestreo* y se mide en *Hertz: pulsos o ciclos por segundo*.

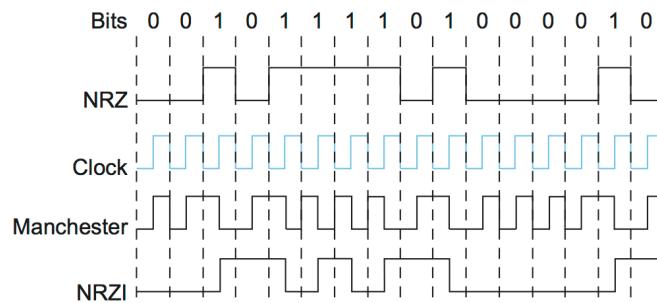
Si el transmisor y receptor tienen relojes independientes, como en el caso de las redes, surge el problema de *sincronización*. Los relojes no son perfectos y tienden a *desincronizarse* luego de un breve tiempo.

Una solución posible es el *framing*: Transmisión en *grupos de bits* de una longitud determinada, lo que permitirá la *re-sincronización* entre dos frames consecutivos.

Es común el uso de *preámbulos* o patrones de bits que representan el símbolo de comienzo de un frame lo que permite que el receptor inicie el proceso de *muestreo* del frame.

También es común el uso de *gaps* o *intervalos* entre *frames* para permitir que el receptor reinicie su proceso de muestreo. Esto facilita la sincronización con el emisor.

Otra forma es evitar secuencias largas del mismo bit transmitido, haciendo que las *transiciones* sirvan como puntos de re-sincronización. Esto motiva algunas de las codificaciones mostradas en la siguiente figura.



**Figura 2.1: Algunas posibles codificaciones.**

La *codificación de Manchester* por ejemplo, aplica un *o exclusivo (XOR)* del valor del reloj con el bit de datos.

La transmisión de *pulsos* es común en circuitos electrónicos y algunos medios de transmisión por cables de cortas distancias como por ejemplo Ethernet y USB.

## Transmisión de señales electro-magnéticas

La transmisión de datos digitales por pulsos tiene el inconveniente que éstas señales sufren *atenuaciones* importantes cuando se desea transmitir a distancias mayores, como lo es en el caso de los sistemas telefónicos o redes de datos.

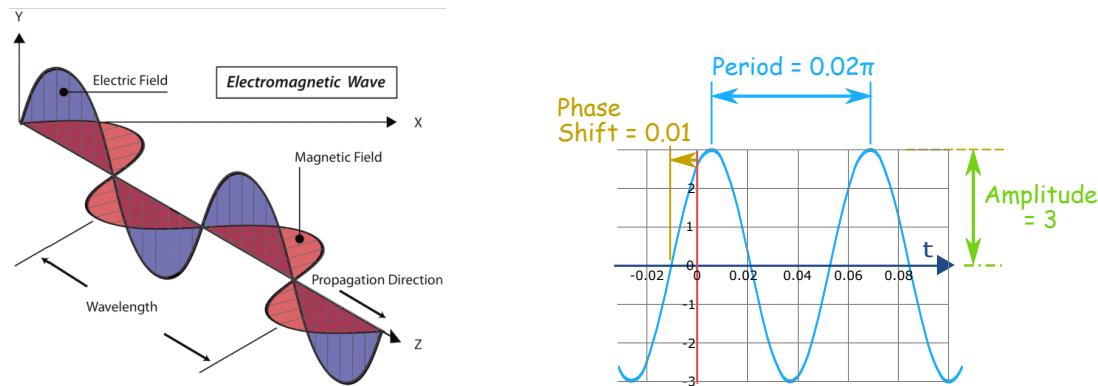
Además, ese tipo de transmisión se denomina **baseband** ya que permite transmitir sólo una secuencia de datos.

La transmisión de señales analógicas en forma de ondas electromagnéticas permiten codificar (o *modular*) señales digitales y pueden usarse también en medios de comunicación como el aire o el vacío.

También es posible transmitir por un medio diferentes ondas electromagnéticas a distintas frecuencias en paralelo sin que se interfieran. Esta técnica de conoce como *broadband* o *multiplexado de frecuencias*.

Una **señal electro-magnética (EM)** se genera al hacer variar el sentido de la corriente **periódicamente** en un material conductor (antena).

El proceso es reversible, es decir que un receptor puede generar corriente desde la EM. La [figura 2.2](#) muestra las características de una señal.



**Figura 2.2: Señales electromagnéticas.**

Matemáticamente, en una función de la forma

$$y(t) = a \times g(p(t + s)) + d$$

siendo  $g$  una función periódica (seno, coseno, etc), los parámetros  $a, p, s$  y  $d$  determinan:

- $a$ : La amplitud de la señal.
- $p$ : El período  $\frac{2\pi}{p}$  es el tiempo que la función completa un ciclo.
- $f$ : La frecuencia. Es la inversa del período:  $f = \frac{1}{p}$ .
- $\lambda$ : *Longitud de onda*. Es la distancia entre dos *crestas* (o *valles*) consecutivos.
- $s$ : El desplazamiento (*phase shift*) en el dominio.
- $d$ : El desplazamiento vertical.

La *velocidad de propagación* de una onda en el espacio (o vacío):

$$v = \frac{\lambda}{T} = \frac{\lambda}{p} = \lambda \times f$$

Así,  $\lambda = \frac{v}{f}$

Esta velocidad depende del medio de transmisión y otras condiciones como la temperatura.

Tipo de señal	Velocidad de propagación ( $\approx$ )
Luz en el vacío	299.792.458 m/s
Luz en el aire	299,708 km/s
Sonido en el aire (20°)	343 m/s

Tipo de señal	Velocidad de propagación ( $\approx$ )
Cable coaxial	66% de la velocidad de la luz.

**Tabla 1:** Velocidades de propagación en diferentes medios.

El **ancho de banda (bandwidth)** de un medio es el **rango de frecuencias** que puede transmitir sin que las señales sufran **atenuaciones severas**.

Se debe notar que el ancho de banda es una propiedad del medio físico.

En la práctica se limita el ancho de banda mediante la aplicación de *filtros*, dependiendo de la aplicación. Por ejemplo, en una línea telefónica analógica se utiliza un ancho de banda de 3000Hz, adecuado para transmitir un buen rango de tonos de voz en forma analógica a largas distancias por cable.

Es común que a mayores frecuencias las señales sufran de mayor **atenuación** (pérdida de potencia) debido a la generación de mas ruido en el espacio. Por esto en la práctica se usan comunicaciones a altas frecuencias para conectar dispositivos en distancias cortas.

Jean-Baptiste Fourier demostró que una función periódica  $g(t)$  con período  $T$  se puede construir desde la suma de términos (serie) dados por

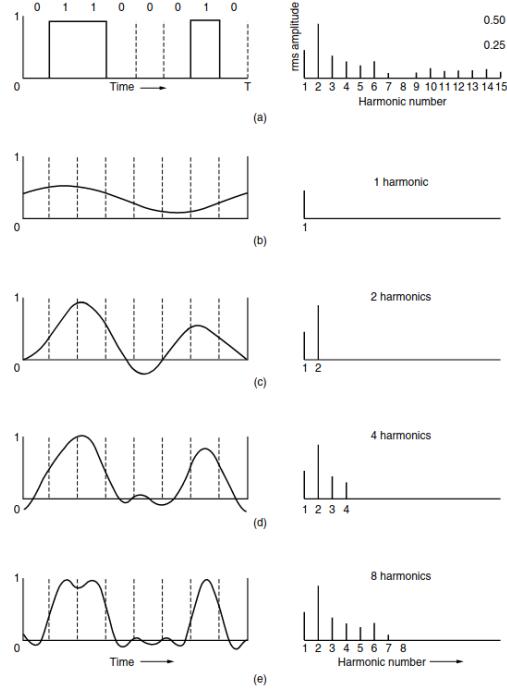
$$g(t) = \frac{1}{2}c + \sum_{n=1}^{\infty} a_n \sin(2\pi nft) + \sum_{n=1}^{\infty} b_n \cos(2\pi nft)$$

donde  $f = 1/T$  es la **frecuencia**,  $c$  es una constante y los coeficientes  $a_n$  y  $b_n$  de cada término son las **amplitudes** de la **n-ésima armónica**.

La transformada de Fourier comúnmente se emplea para descomponer una señal en sus componentes (armónicas). De cada componente puede extraerse su frecuencia, amplitud y su fase (ángulo o desplazamiento sobre el eje temporal) inicial.

En la siguiente figura se considera la transmisión digital de 8 bits.

## Enlaces directos



La figuras b), c), d) y e), muestran la reconstrucción de la señal usando 1 a 8 armónicas. Se puede apreciar que si se transmiten 8 armónicas la señal puede ser reconstruida por el receptor con una buena aproximación, haciendo innecesario transmitir la señal a mayor frecuencia (notar que cada armónica tiene una frecuencia múltiplo de la frecuencia original  $f$ ).

Por ejemplo, una línea telefónica de voz estándar tiene un ancho de banda de 3000Hz debido los filtros utilizados para transmisión de voz analógica. Si se usa para transmitir datos digitales a una cierta tasa de  $b$  bits por segundo, el tiempo para transmitir 8 bits es  $8/b$  segundos y la frecuencia de la primera armónica (la fundamental) es de  $b/8$ Hz.

En éste caso, la *armónica de mayor frecuencia* será de  $3000/(b/8)=24000/b$ . Esto indica que a una tasa  $b=19200$  bps ya no pueden pasar más de una armónica, lo cual daría un límite para  $b$ .

La tabla siguiente describe el número de armónicas que podrán transmitirse efectivamente para diferentes tasas de transmisión.

Bps	T (ms)	Primera armónica (Hz)	Armónicas
300	26.67	37.5	80
600	13.33	75	40
1200	6.67	150	20
2400	3.33	300	10
4800	1.67	600	5

Bps	T (ms)	Primera armónica (Hz)	Armónicas
9600	0.83	1200	2
19200	0.42	2400	1

**Tabla 2: Tasa de transmisión y armónicas transmitidas.**

Se puede notar que para transmitir a 9600 bps sólo se transmitirán 2 armónicas (como en el caso (c) de la figura anterior), dificultando la reconstrucción de la señal original en el receptor.

## Tasa de transmisión

**La tasa de transmisión (data rate) es el número de bits transmitidos por segundo.**

Es común que en comunicaciones digitales se use este término como sinónimo de *ancho de banda* aunque ya vimos que éste último es el *rango de frecuencias permitido por el medio*.

Estos dos valores están directamente relacionados.

La tasa de transmisión depende del ancho de banda ( $B$ ) del medio y el número de valores discretos ( $V$ ) codificados.

Es común que, por razones técnicas, en un medio de transmisión se empleen filtros para limitar las frecuencias permitidas y eliminar ruido.

En 1924, Nyquist estableció que en un canal filtrado sin ruido con un ancho de banda  $B$ , la señal puede reconstruirse (en el receptor) tomando exactamente  $2B$  muestras por segundo.

Esto significa que realizar el muestreo a frecuencias mayores no contribuye en nada ya que los componentes de las demás frecuencias se han filtrado.

Así el teorema de Nyquist define:

**La tasa máxima de transmisión =  $2B \log_2 V$**

Por ejemplo, un canal (sin ruido) con un ancho de banda  $B = 3Khz$  puede transmitir 6000bps usando señales digitales binarias ( $V = 2$ ).

La tabla de la sección anterior muestra que a tasas de transmisión bajas se pueden reconstruir las señales con mayor fidelidad (pasan más armónicas). Para aumentar la tasa de transmisión se debe aumentar el ancho de banda o usar codificaciones y modulaciones apropiadas para ampliar el conjunto de valores discretos  $V$ .

## Ruido

La ecuación anterior asume un canal *libre de ruido*. En la práctica, en un canal siempre hay *ruido*, es decir, otras señales electromagnéticas generados por la radiación solar, motores eléctricos y otras fuentes. El ruido distorsiona las señales, afectando a la tasa máxima de transmisión.

La *cantidad de ruido en un canal* =  $\frac{S}{N}$ , donde  $S$  es la potencia de la señal transmitida y  $N$  es la potencia de las señales de ruido. Estas potencias se expresan en **Decibeles**:  $Db = 10 \log_{10} \frac{S}{N}$ .

Así,  $\frac{S}{N} = 100 = 20Db$ ,  $\frac{S}{N} = 1000 = 30Db$ , ...

**Shannon postuló que la *capacidad máxima de un canal*** =  $B \log_2(1 + \frac{S}{N})$ .

Al comparar la capacidad del canal con la tasa de transmisión es posible determinar el número de valores discretos eficaces  $V$ .

$$V = \sqrt{1 + \frac{S}{N}}$$

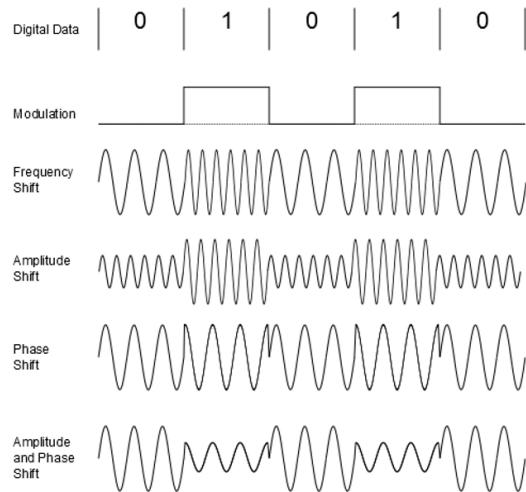
Esto determina un límite físico a la tasa máxima de transmisión en un canal con ruido.

## Modulación

Para transmitir datos sobre ondas electromagnéticas, se usa comúnmente una *señal portadora*, la cual se modifica su forma en intervalos en el tiempo para representar los diferentes valores que se desean transmitir.

En el caso de las comunicaciones digitales la señal portadora se modifica en base a la señal digital (cuadrada) de entrada.

Como ya se describió, una señal digital se genera como una función con dominio y rangos discretos. En comunicaciones digitales, el dominio (tiempo discreto) determina el *tiempo de bit*. El rango determina el conjunto de valores posibles a transmitir. El receptor debe reconstruir los componentes de la señal recibida para *reconocer* los valores digitales *codificados* en la señal.



**Figura 2.3: Modulación de datos digitales.**

En la [Figura 2.1](#) se muestran las técnicas básicas de modulación. A partir de una señal de entrada y una *señal analógica portadora (carrier)*, éstas se combinan para producir la señal que *codifica* los datos.

En comunicaciones digitales la señal de entrada es una función discreta (o señal cuadrada).

Es posible *codificar* información en una señal en el *tiempo de bit* (intervalo de tiempo) correspondiente modificando algunos de sus parámetros.

- Una señal de **modulación por frecuencia (FM)** cambia la frecuencia (usa un oscilador variable modificando  $p$ ).
- La **modulación por amplitud (AM)** cambia la *amplitud* (variando el voltaje de la señal).
- La **modulación por desplazamiento de fase (PSM)** desplaza la onda en una cierta cantidad de grados.

Por supuesto, es posible combinar estas técnicas. Una técnica muy usada es la modulación PQSK, que combina desplazamiento de fase (0, 90, 180 y 270 grados) con modulación por amplitud y permite codificar 4 valores discretos por cada tiempo de bit (genera 4 formas diferentes de la señal) lo que es equivalente a transmitir 2 bits por unidad de tiempo.

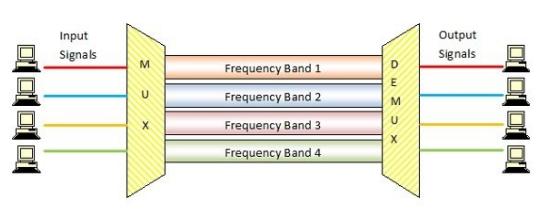
Algunas técnicas de modulación y codificación actuales permiten transmitir hasta 8 valores discretos por tiempo de bit.

## Multiplexado

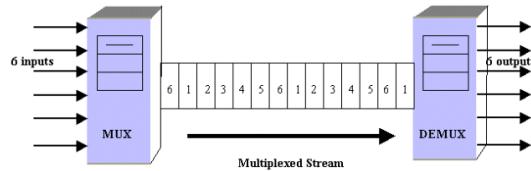
El *multiplexado* permite múltiples comunicaciones sobre un mismo medio compartido. Un receptor deberá *separar* o *demultiplexar* las diferentes comunicaciones (*streams*) recibidas.

Las técnicas utilizadas se pueden clasificar en:

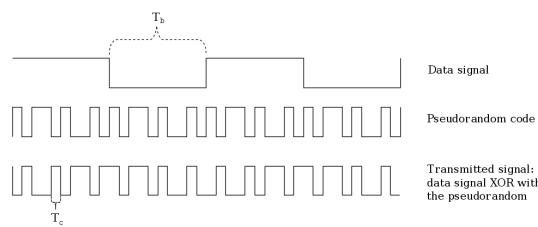
1. **División de frecuencias (FDM):** El ancho de banda del canal se divide en subcanales evitando que se interfieran entre sí. Entre cada canal generalmente existe un *gap* o separación que hace que ocurra un pequeño desperdicio del ancho de banda total. Técnicas de modulación como Orthogonal Frequency Division Modulation (OFDM).
2. **División de tiempo (TDM):** Cada dispositivo utiliza la totalidad del canal por turnos y por un intervalo de tiempo determinado, generalmente con una estrategia *round-robin*.
3. **División por códigos (CDM):** La idea es que cada participante genera dos señales digitales superpuestas: La de datos, a una frecuencia  $f_d$  y otra de alta frecuencia  $f_c$  (el código, asignado a cada comunicación). La señal transmitida es el *XOR* entre las dos señales (con frecuencia  $f_c$ ), como se muestra en la Figura 2.2 c). Así varias secuencias de datos pueden transmitirse simultáneamente. Un receptor puede *extraer* la secuencia de una comunicación en particular, haciendo nuevamente el XOR entre la señal recibida y el código correspondiente. Además de permitir el multiplexado, este tipo de multiplexado también permite definir un mecanismo de acceso al medio (como en *CDMA*). Una técnica comúnmente utilizada es la asignación de códigos *ortogonales* entre sí (vistos como vectores, su producto interior es cero).



a)



b)



c)

**Figura 2.4: FDM a), TDM b) y CDM c).**

Una analogía con las conversaciones entre grupos de personas es la siguiente: *FDM* es como diferentes personas hablando simultáneamente lo hacen con diferentes tonos de voz, *TDM* es análogo a que hablen por turnos y *CDM* serían cuando diferentes grupos hablan simultáneamente en el mismo tono pero en diferentes idiomas. En *CDM* cada receptor percibe a las otras conversaciones en idiomas que desconoce como *ruido*.

Por supuesto, estas técnicas se pueden combinar como es el caso de las técnicas usadas en el sistema de telefonía celular, donde se utilizan división de tiempo y de frecuencias para soportar varias comunicaciones simultáneas compartiendo bandas (canales) entre diferentes dispositivos en una celda.

## Introducción

## Enlace de datos

# Enlace de datos

Los protocolos a nivel de enlace de datos (capa 2 del modelo OSI) deben resolver los problemas de *framing* (*empaquetado*), gestión del medio (canal de comunicación) y *detección de errores*. El modelo OSI denomina las unidades de transmisión en esta capa como *frames*.

Los datos a transmitir comúnmente se dividen en *frames* para resolver dos problemas principales:

1. *Control de errores*: Los medios físicos son propensos a interferencias, ruido y otros problemas que pueden afectar a la comunicación.
2. *Sincronización*: Es común que el transmisor (*tx*) y el receptor (*rx*) se deban sincronizar en el tiempo para que el receptor pueda tomar las muestras de la señal y reconstruirla adecuadamente.

Es posible clasificar los protocolos en base a cómo empaquetan los datos transmitidos.

- *Orientados a bytes*: Un frame consiste en una secuencia de bytes.
- *Orientados a bits*: Los datos en el frame se interpretan como una secuencia de bits.

Un protocolo en esta capa generalmente interactúa directamente con uno o más dispositivos de hardware de comunicaciones.

Una interface de red es un dispositivo de hardware/firmware y se denomina *Network Interface Controller (NIC)*.

Algunos protocolos, como *PPP*, son independientes del medio físico y *NICs*. Otros protocolos, como *ATM* y *Ethernet*, también especifican los medios físicos, detalles del tipo de señales, la modulación y técnicas de multiplexado soportados. En estos últimos protocolos podríamos decir que también incluyen la especificación de la capa física.

## Point-To-Point (PPP) protocol

Este protocolo ([RFC 1661](#)) se usa comúnmente para la conexión de terminales a mainframes y enlaces punto a punto en redes WAN. Es *orientado a bytes* y soporta frames de longitud fija o variable.

Comúnmente se usa sobre enlaces físicos como cables serie del tipo RS232, de telefonía, líneas dedicadas, fibra óptica y otros. Requiere enlaces *full-duplex* (comunicación bidireccional simultánea).

Se usa en forma de líneas de *dial-up*, es decir un extremo (peer) *llama* a otro y éste último acepta o rechaza la conexión. Tiene las siguientes fases:

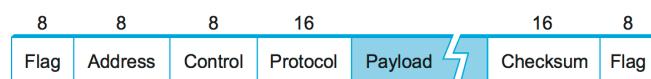
1. *Link dead*: Los peers están desconectados
2. *Link establishment*: Negociación usando LCP (descripto abajo).
3. *Network Layer phase*: Transporte de datos desde/hacia la capa de red.
4. *Link termination*: Luego que algún peer decida cortar la conexión (*hung-up*) o al ocurrir un error.

PPP contiene otros protocolos auxiliares. El [Link Control Protocol \(LCP\)](#) se usa durante el *establecimiento de la conexión* para negociar los parámetros de la comunicación antes de comenzar la transmisión de datos. Los mensajes LCP se encapsulan en frames PPP.

Algunos de los parámetros son el tamaño de los frames (fijos o variables), identificación de los peers (direcciones de red), tasa máxima de transmisión, tamaño máximo del payload (*Maximum Transmission Unit* o *MTU*), compresión de datos y autenticación de las partes.

Algunos de los protocolos de autenticación usados son ([PAP](#) y [CHAP](#)).

La siguiente figura muestra un frame PPP.



**Figura 3.1: Formato de un frame PPP.**

Cada frame tiene delimitadores de inicio y fin ( **flag** ) que es el valor **0x7E** . Se usa como carácter de escape el valor **0x7D** . Si alguno de esos valores aparecen en un byte de datos se reemplaza por el carácter de escape **0x7D** seguido por el **byte  $\oplus 0x20$** .

El campo **protocol** identifica el protocolo de la capa superior (LCP, IP, etc).

El campo `checksum` se usa para detectar errores de transmisión y se calcula como un CRC sobre los campos `address`, `control`, `protocol` y `payload`.

La RFC 2516 especifica cómo usar frames PPP encapsulados en frames Ethernet. PPPoE es usado comúnmente por proveedores de servicios de Internet (ISPs) para autenticar usuarios y negociar las características de la conexión (como la tasa máxima de conexión) con el servidor del proveedor.

## Protocolos orientados a bits

Un protocolo orientado a bits interpreta su *payload* como una secuencia de bits. Para determinar los límites se usan dos enfoques:

1. Un campo que contenga la cantidad de bits del payload
2. Usar delimitadores (marcas) de inicio y fin

A modo de ejemplo, el *High-Level Data Link Control (HDLC)* usa los mismos delimitadores de PPP. Para evitar que un dato arbitrario no produzca la marca de fin, cada 5 bits consecutivos con valor 1 en el payload se inserta un cero. El receptor luego de cada 5 1s consecutivos, si recibe un cero, lo descarta. Si el siguiente es un 1, o es la marca de fin y el siguiente debe ser un 0. En el caso que el siguiente bit sea un uno tiene que ser producto de un error.

## Ethernet

La familia de tecnologías y protocolos *Ethernet* (estándar IEEE 802.3) es muy amplia, flexible y mantiene compatibilidad hacia atrás. Generalmente se utiliza en LANs.

Cada *Network Interface Controller (NIC)* Ethernet tiene una dirección única de 48 bits. Una dirección (conocida como *Medium Address Control (MAC)*) Ethernet se denota como una secuencia de 6 valores de dos dígitos hexadecimales separados por dos puntos ( : ). Las direcciones Ethernet se asignan según el Internet Assigned Numbers Authority (IANA).

En Ethernet original (10Base5 y 10Base2) usaba cable coaxial y los nodos se conectaban a cada segmento formando un *medio compartido* logrando una *topología de bus*.

Cada versión define la tasa de transmisión soportada. Ethernet es un protocolo de comunicación con *flujo de control* basado en una tasa de transmisión fija, desde 10Mbps en 10Base5 y 10Base2, 100Mbps en 100Base-TX, hasta 1Gbps en 1000BaseX sobre fibra óptica.

La norma *10BaseT* reemplaza el coaxial por cables de *pares trenzados (UTP)*, más flexibles y de menor costo. En *10BaseT* cada nodo se conecta inicialmente a un *hub* que actuaba como un *repetidor* de los frames transmitidos a los demás nodos. Esto resulta en una *topología de estrella*. *10BaseT* permite comunicación **full duplex** (bidireccional simultánea).

Estas dos tecnologías *comparten* el medio físico de comunicación por lo que el protocolo necesita un mecanismo de *arbitraje* en el acceso al medio. El algoritmo usado es ([Carrier Sense-Multiple Access with Collision Detection \(CSMA-CD\)](#)).

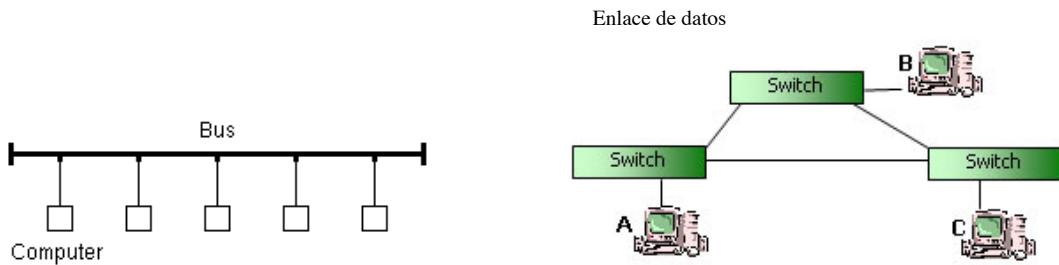
*CSMA-CD* se basa en que el dispositivo tiene que *esperar* hasta detectar el canal libre (señal portadora) para luego iniciar la transmisión de un frame. Esto no impide *colisiones* ya que dos estaciones pueden iniciar su transmisión simultáneamente. En este caso, las estaciones transmisoras, detectan la colisión (comparando lo que están transmitiendo con lo que están recibiendo por el canal) y abortan la transmisión, reintentando luego de una espera aleatoria (algoritmo *backoff*).

Notar que esto define un *sistema distribuido*, ya que no requiere de una coordinación centralizada.

Actualmente se utilizan *switchs* que permiten conexiones *punto a punto* entre el switch y cada estación, formando una topología de estrella.

A diferencia de un *hub*, un *switch* retransmite un frame sólo al puertos de la estación *destino*, analizando la dirección de destino de la cabecera del frame. De esta forma se evitan colisiones y se permiten comunicaciones simultáneas sobre subconjuntos disjuntos de estaciones.

Los *switches* pueden interconectarse entre sí para formar generalmente una topología de árbol. Un *switch* moderno también hace de *puente (bridge)* transformando frames de diferentes versiones de Ethernet, permitiendo la interconexión de dispositivos heterogéneos como NICs de 10Mbits con otros de 100Mbits.



*Figura 3.2: Topologías de bus y uso de switchs.*

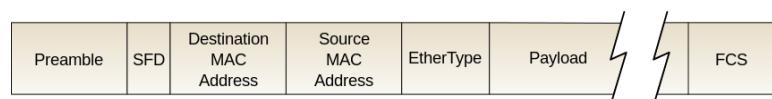


*Figura 3.3: Foto de un switch (puertos RJ-45).*

Un *switch* o *bridge* usa la técnica de *store and forward* almacenando en memoria temporalmente los frames, para analizar sus cabeceras y retransmitirlos por los *puertos* correspondientes.

*¿Cómo sabe un switch a qué puerto tiene conectada una interface con una dirección MAC dada para enviar un frame con ese destino?* Bien, en principio no sabe, pero va asociando pares de la forma (*MAC, port*) a medida que recibe frames desde el puerto correspondiente. De esta manera un switch en el inicio se comporta como un *hub* ya que retransmite cada frame por cada puerto aún no asociado a una dirección Ethernet. El switch va *aprendiendo* qué MAC está conectado a cada port a medida que recibe frames.

La [figura 3.4](#) muestra el formato de un *frame Ethernet*.



*Figura 3.4: Formato del frame Ethernet.*

El *preámbulo* es un patrón de bits que facilita la *sincronización* de los NICs. El campo *SDF* (*start delimiter frame*) indica el inicio del frame.

Las direcciones de origen y destino permiten determinar los dispositivos que intervienen en la comunicación. Una interfaz que recibe un frame con destino no se corresponde con su MAC, simplemente lo descarta.

La dirección **FF:FF:FF:FF:FF:FF** está reservada para *broadcast*, es decir especifica que el frame tiene como destino a todas las demás interfaces conectadas en la LAN.

El campo *EtherType* puede usarse con dos propósitos: Si su valor es menor o igual a 1500, indica la longitud del *payload* (datos). Si es mayor a 1500, indica el tipo del *payload*, es decir, qué tipo de *paquete* contiene. Por ejemplo, si contiene un paquete o *datagrama IP*, su valor es *0x0800*.

Los números de protocolos (*payload type*) también son asignados por la [Internet Assigned Numbers Authority](#).

Cuando el campo *EtherType* no representa el tamaño del *payload*, el final del frame puede detectarse o por un símbolo representando el *fin del data-stream* (por ejemplo, un [código 8b/10b](#) en Gigabit Ethernet) o por la pérdida de la señal de portadora sobre la que se modulan los datos. En éste último caso se define un intervalo entre frames (*inter-frame gap*).

El *payload* corresponde a los datos transmitidos de longitud entre 42 y 1500 bytes. El *frame check sequence (FCS)* es un *cyclic redundancy check (CRC)* para detección de errores como el que se describe en la sección [CRC](#).

## Asynchronous Transfer Mode (ATM)

[ATM](#) fue diseñado como un protocolo flexible para proveer servicios de transmisión de datos para diversos tipos de aplicaciones, desde aplicaciones de VoIP (voz sobre IP) hasta aplicaciones no interactivas.

Un *frame ATM* es de longitud fija de 53 bytes. Permite establecer *circuitos virtuales* entre los extremos, ofreciendo un servicio *orientado a conexión*.

Ha sido utilizado ampliamente en redes de telefonía para la transmisión de datos digitales como *Digital Line Subscriber (DSL)*. El protocolo permite el uso de enlaces físicos de diferentes tipos como cables telefónicos o fibra óptica.

Muchos proveedores de servicios de internet (ISPs) usan ([PPPoE](#)) sobre ATM. Esto es, PPP *encapsulado* en paquetes Ethernet para aprovechar las funciones de negociación de parámetros de enlace como compresión de datos, limitación de tasa de transmisión en base al plan del suscriptor y autenticación de usuarios.

Comúnmente cada *modem DSL* de usuario se conecta a un equipo tipo [DSLAM](#) del ISP por enlaces (cable telefónico, coaxial, fibra óptica, etc) usando ATM o Ethernet. El *modem DSL* también opera como un *switch L2* para la red local del usuarios y *router (L3)* o *gateway* que permite rutear paquetes desde/hacia el router del ISP. Algunos ISPs encapsulan PPPoE sobre ATM (PPPoEoA) en los enlaces entre el *modem/router* del usuario y el *DSLAM*.

El tamaño del frame (ATM *cell*) permite la transmisión de pequeños mensajes, lo cual es común en aplicaciones de transmisión de audio o mensajes de control, permitiendo minimizar el *jitter* o *fluctuación* entre los tiempos (o separación) entre frames.

## Formato de frames ATM

Una *celda* ATM tiene una cabecera (*header*) de 5 bytes y un *payload* o cuerpo de 48 bytes. Los principales campos de la cabecera son:

- *Virtual Path Identifier (VPI)*: 1 byte
- *Virtual Channel Identifier (VCI)*: 2 bytes
- *Payload Type (PT)*: 1 byte
- *Header Error Control (HEC)*: 1 byte CRC (polinomio  $x^8 + x^2 + x + 1$ )

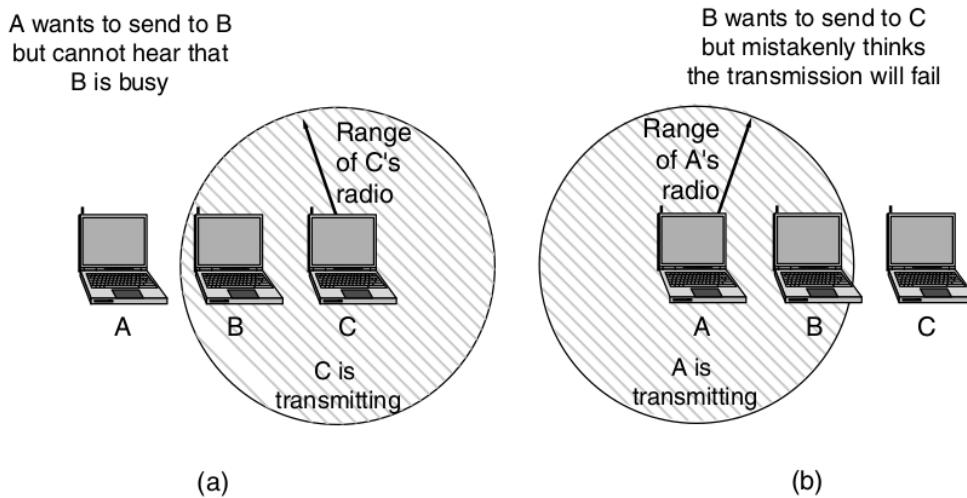
El par de valores (*VPI,VCI*) se usa para determinar el *circuito virtual* a seguir entre los extremos. Un switch ATM usa estos valores para determinar el próximo dispositivo a retransmitir la celda. Cada switch utiliza una función de *labeling* (*rotulado*) implementada como una tabla con entradas de la forma  $(vpi_{in}, vci_{in}, vpi_{out}, vci_{out}, vlc)$  que mapea una terna  $(vpi_{in}, vci_{in}, vcl)$  al puerto de salida representado por  $(vpi_{out}, vci_{out})$  para continuar con el próximo salto perteneciente al *virtual circuit link (VCL)*.

## Wifi (IEEE 802.11)

Basado en IEEE 802, permite comunicación inalámbrica (aire). Usa ***Carrier Sense Multiple Access with Collision Avoidance (CSMA-CA)***. La idea es similar a ***SCMA*** pero se transmite todo el frame. En una colisión los receptores detectarán frames inválidos. Se usan *bandas libres* de frecuencias (centrales) como 2.4 o 5Ghz.

El protocolo en realidad no evita colisiones, principalmente cuando dos estaciones inician su transmisión por estar fuera de alcance entre sí (cada una detecta el canal libre). Este problema se conoce como el de la *estación oculta*, y se muestra en la [figura 3.5 a\)](#).

Otro problema a resolver es el de *estación expuesta*, como se muestra en la [figura 3.5 b\)](#), en la cual **B** podría transmitir a **C** sin problemas ya que C no recibiría interferencias.



**Figura 3.5: a) Estación oculta. b) Estación expuesta.**

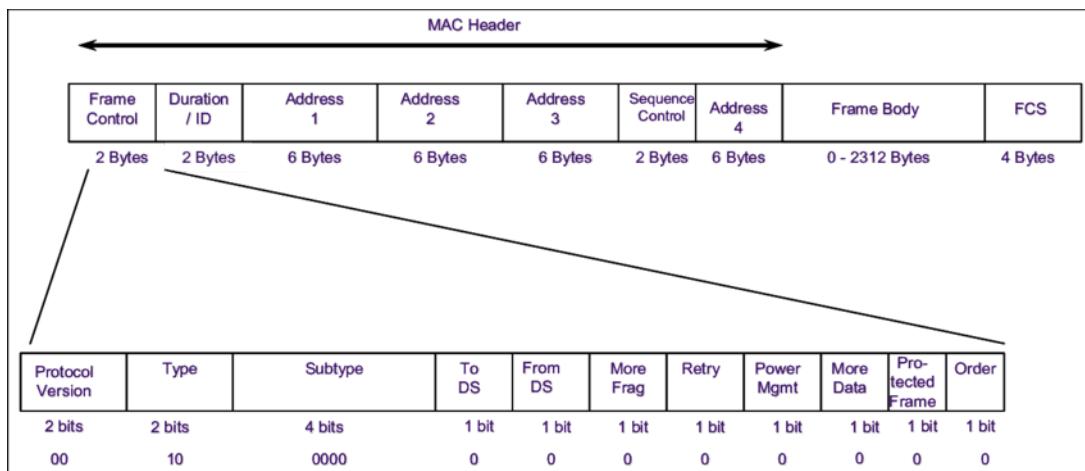
Para evitar estos casos, una estación que desea transmitir, envía un mensaje **RTS** (*request to send*) indicando el destino y la duración del uso del canal. El receptor, responde con **CTS** (*clear to send*). El emisor comienza a transmitir luego de la recepción del **CTS**. El receptor envía mensajes de *reconocimiento de recepción de cada frame* (**ACK**).

Si el *timer* disparado por el transmisor expira, se asume una colisión u error y se reinicia el intento de transmisión.

Esto implementa un protocolo distribuido de acceso al canal, permitiendo el *modo ad-hoc*, también llamado *Distributed Coordination Function*, es decir sin uso de *access point (AP)* actuando como coordinador central.

Los protocolos 802.11 fue evolucionando en versiones *a,...,g*. Esta última utiliza **OFDM** y usa *canales* con un ancho de banda de 20Mhz alcanzando una tasa de transmisión de 54Mbps. La última versión, 802.11n puede alcanzar tasas de transmisión de hasta 450Mbps, usando más ancho de banda usando hasta 4 antenas.

La siguiente figura muestra el formato de un frame 802.11.



**Figura 3.6: Formato de un frame 802.11.**

La *dirección 1* contiene la dirección de destino, la *dirección 2* la de origen y la *dirección 3* generalmente contiene la dirección del *default gateway* (que puede ser el *access point*).

En **modo infraestructura** (la mas común actualmente), se usan **access points (AP)**, los cuales hacen de *switch* y *bridge* (L2) y *router* (L3), permitiendo su conexión a una red cableada Ethernet. Un AP representa una red local, con su identificador (*SSID*) y canales utilizados. Periódicamente transmite un *beacon* con el *SSID* mas información sobre su acceso (parámetros de seguridad) y actúa como un coordinador central de la red (*Point Coordination Control*).

Un AP permite que una estación se *una* a la red (con posible autenticación) usando un *protocolo de asociación*. También existe la operación de *salida* o *disasociación*. Este modo provee una *topología estrella*. Un AP puede interconectarse con otros APs y/o switches.

En este modo puede usarse un esquema de seguridad. Actualmente se utiliza **Wi-fi Protected Access(WPA)** que brinda autenticación con contraseñas para asociación y canales seguros (cifrado de datos).

Otras tecnologías inalámbricas ampliamente usadas son Bluetooth para distancias cortas, redes de telefonía celular (3, 4 y 5G) y LoRa, usada para largas distancias y bajas tasas de transmisión.

Bluetooth usa frecuencias similares que 802.11 (2.4Ghz, con un ancho de banda de 83Mhz) por lo que puede causar interferencias si se usan simultáneamente.

Para evitar esto ambas tecnologías usan multiplexado en canales y *saltos periódicos*. Bluetooth divide el ancho de banda en 79 canales, cada uno de 1Mhz y usa *Frecuency-hopping Spread Spectrum*, saltando de canal 1600 veces por segundo. Wi-Fi divide el canal en 11 (o 14) canales donde cada transmisión ocupa 5 canales. Wi-Fi usa una técnica de modulación conocida como Direct-Sequence Spread Spectrum que permitiendo la coexistencia con otras redes Wi-Fi y Bluetooth donde el receptor puede *extraer* la señal requerida. Esta técnica permite implementar *Code-Division Multiple Access* como se analizó en el capítulo de enlaces directos.

## Interconexión con switches

Generalmente se interconectan varias redes Ethernet mediante switches con enlaces redundantes para proveer tolerancia a fallas.

Esto hace que posiblemente se formen *ciclos*, los cuales debe evitarse a nivel lógico para que las tablas de switching de cada switch se mantengan coherentes.

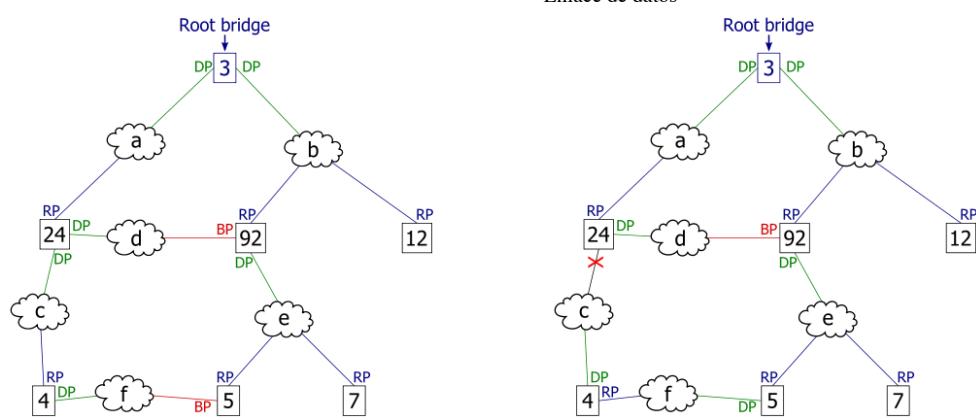
Para ello, los switches corren el Spanning Tree Protocol (STP) que forma un árbol de switches, designando un *root bridge*.

Cada switch se configura con un ID, el cual se concatena a su dirección MAC. STP selecciona como *root bridge* al valor menor. El árbol se arma seleccionando los *caminos* más cortos en base al ancho de banda de cada enlace.

STP designa a cada puerto que conecta a otro switch con una de las siguientes categorías:

- **Blocking (BP):** Desactivado para prevenir *loops*.
- **Designed (DP):** Puerto (el de menor ID del switch) que conecta a la LAN.
- **Root (RP):** Puerto que conecta en un *path* al *root*.

La siguiente imagen muestra en a) el árbol creado por STP en una red de ejemplo y en la parte b) la reconfiguración obtenida ante la falla del enlace entre el switch 24 y la red c.



*Figura 3.7: a) Spanning tree. b) Reconfiguración luego de una falla.*

## Maximum Transmission Unit (MTU)

Un protocolo de enlace de datos si opera con frames de tamaño máximo restringe el tamaño máximo del *payload* para los protocolos de las capas superiores. Esto comúnmente se conoce como *Maximum Transmission Unit (MTU)*.

El MTU indica a un protocolo de red, como por ejemplo IP, que si debe enviar un payload de mayor tamaño que el MTU deberá *fragmentarlo*, es decir enviar una secuencia de paquetes más pequeños. La *fragmentación* puede afectar al rendimiento del sistema de comunicaciones, ya que se agregan intervalos de inter-frames, latencias (tiempo de preparación para el envío de un paquete), etc.

## Virtual LAN

Una *vLAN* es una red a nivel de enlace que permite dividir una red física en varias redes lógicas. Un bridge vLAN rutea frames entre las redes lógicas. El estándar Ethernet con soporte vLAN es el [802.1ad](#) que extiende el formato del frame conteniendo un *tag* o identificador de una vlan. Este formato también es aplicable en el estándar 802.3 (Ethernet por cable).

## Hubs, switches, bridges y routers

Todos estos elementos de hardware de red permiten interconectar otros dispositivos, como hosts y otros dispositivos de interconexión para interconectar redes.

La diferencia de nombres se determina en base en qué capa del modelo OSI operan, como se muestra en la siguiente tabla.

Dispositivo	Capa en el modelo OSI
Hub	Capa física (1). Reenvía cada frame (señal) en todos los demás puertos (broadcast)
Switch	Capa de enlace (2). Rutea tráfico sólo a los puertos de destino.
Bridge	Capa de enlace (2). Conforma una red a partir de dos o más <i>segmentos</i> o redes.
Router	Capa de red (3).

Cabe aclarar que muchos dispositivos implementan servicios en todas esas capas, por lo que es común denominarlos *routers* o *gateways*.

---

< Anterior

Enlaces directos

Próximo >

Infraestructura

# Infraestructura

La interconectividad lograda en la actualidad se basa en diferentes tipos de tecnologías para formar una gran red de redes entre sí a nivel mundial.

Aquí se describen algunas tecnologías usadas para acceder a Internet desde nuestros hogares, oficinas de trabajo o inclusive desde teléfonos móviles (celulares).

Un *Internet Service Provider (ISP)* provee servicios de interconexión a Internet. Estas organizaciones generalmente tienen una red propietaria que a su vez está interconectada a una red regional (un país, por ejemplo) principal o *backbone* que permite la interconexión entre ISPs. Estos a su vez utilizan los servicios de otras empresas que proveen servicios de telecomunicaciones internacional, contando con una infraestructura de enlaces intercontinentales basados en comunicaciones satelitales, cables o fibra óptica.

Actualmente la mayoría de las comunicaciones globales atraviesan los enlaces intercontinentales de cableado submarino (de fibras ópticas), como se muestran en el siguiente [mapa](#). En [nic.ar](#) se pueden encontrar mas detalles de la interconexión de Argentina a Internet.

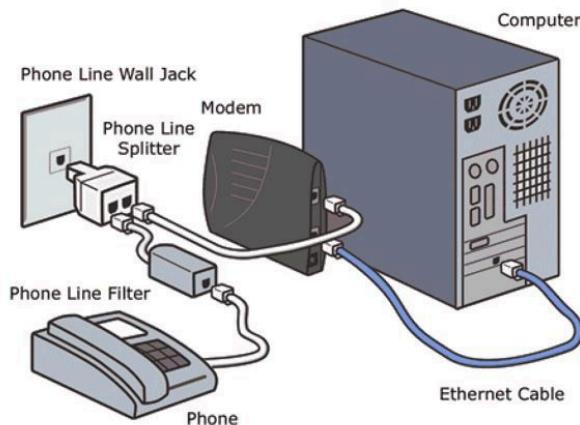
Cada ISP brinda el servicio de conectividad a sus clientes mediante algunas de las siguientes tecnologías:

- **Digital Subscriber Line (DSL):** Uso de líneas telefónicas (cables UTP de cobre).
- **Cable modem:** Basada en la infraestructura generalmente usada en los proveedores de TV por cable.
- **Wi-fi:** Enlaces punto a punto usando antenas direccionales de medio o largo alcance.
- **Fibra óptica:** Líneas de fibra óptica.
- **Telefonía celular:** Servicios de datos usando redes inalámbricas 3G, 4G o 5G (de rápido desarrollo).

## DSL

La tecnología *Digital Subscriber Line (DSL)* es usada principalmente por las empresas de telefonía fija ya que permite reusar la infraestructura de cableados existentes, agregando *repetidores* para filtrar y amplificar las señales por tramos en la red. Cada línea de cada cliente, llega a la central y se conectan a un sistema [DSLAM](#) que multiplexa las comunicaciones de la intranet en los enlaces de alto ancho de banda del *backbone* del ISP.

En el cliente, la línea telefónica se conecta a un *modem DSL* que codifica y modula las señales digitales de la red local en la red local. Este modem también actúa de *switch Ethernet* y/o *wi-fi* para interconectar los dispositivos de la red local, tal como se muestra en la siguiente figura.



El servicio generalmente es asimétrico (*ADSL*) por lo que la tasa de transmisión desde el cliente (subida) es menor que la tasa de recepción o descarga.

Las señales digitales se transmiten en un rango de frecuencias más alto que el usado en telefonía tradicional, lo que permite la comunicación de voz y datos simultáneamente. Para efectivamente separar las frecuencias de voz y datos se usan filtros, como se puede apreciar en la figura de arriba.

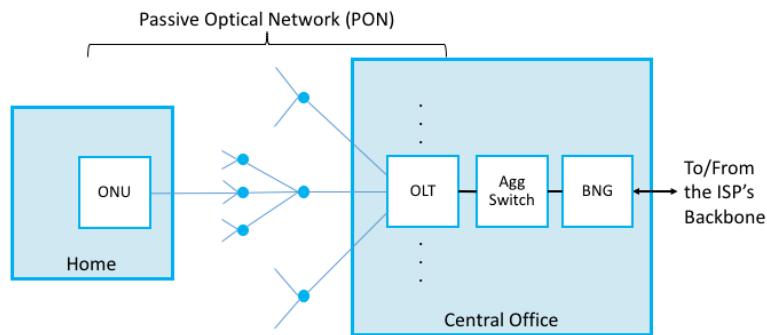
Las versiones modernas como VDSL pueden alcanzar tasas de transmisión de hasta 54Mbps, requiriendo segmentos cortos (hasta 300 metros) entre la conexión del cliente hasta el repetidor.

Comúnmente, en los enlaces entre cada *modem* y la central del ISP se utiliza [PPPoE](#), que permite *encapsular* frames *PPP* dentro de frames *Ethernet*. Esto permite que el ISP pueda utilizar los servicios de control de tasa de transmisión (dependiente del plan de la suscripción del usuario) y autenticación provistos por *PPP*. Es común que entre los enlaces del *modem/router* del ISP y la central (*DSLAM*) se use ATM para la transmisión de frames PPPoE.

Como ya se analizó en el capítulo de enlaces de datos, el uso de *PPPoE* afecta al tamaño máximo del frame o *maximum transmission unit (MTU)*, ya que habrá que informar a la capa de red que en este caso la MTU ya no es de 1500 bytes sino de 1492 (la cabecera de un frame PPPoE consume 8 bytes).

## Fibra óptica

El servicio de enlaces por fibra óptica al cliente se basa en las tecnologías conocidas como *passive optical network (PON)*. El ISP en su oficina central cuenta con múltiples *terminales ópticas (OLT)* que a su vez se conectan a su *backbone* mediante *switches*. De cada *OLT* se distribuyen enlaces de fibra que se divide en ramas por medio del uso de *splitters* formando un árbol hasta cada cliente que cuenta con una *optical network unit (ONU)*, tal como se muestra en la siguiente figura.



Los *splitters* utilizados son *pasivos* (de ahí el nombre), ya que simplemente son *repetidores* de las señales ópticas (no realizan *store and forward* de frames).

La comunicación es bidireccional en cada cable (*pelo*) de fibra ya que se usan dos frecuencias diferentes (*wavelengths*) en las señales ópticas de subida y bajada.

Los frames se multiplexan usando *división de tiempo*. Cada *ONU* filtra los frames sólo dirigidos a su unidad.

## Telefonía celular

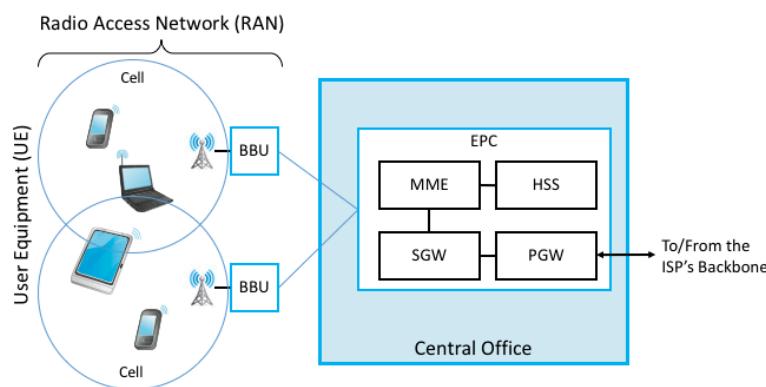
Una red de telefonía celular o móvil se basa en un conjunto de nodos interconectados a una oficina central (generalmente por enlaces de fibra óptica). Estos nodos se conocen como *celdas* o *estaciones base (Broadband Base Unit o BBU)*.

Cada celda contiene equipos de computación y comunicación por radio con un alcance que varía entre los 5 y 20km. Los equipos móviles se denominan *user equipment (UE)*.

Las celdas vecinas usan diferentes rangos de frecuencias para evitar interferencias.

La oficina central implementa una *Evolved Packet Core (EPC)* que brinda servicios de *rastreo* de *UEs*, conocida como *Mobility Management Entity (MME)*, consultas de usuarios suscritos (*Home Subscriber Server o HSS*) y un par de *gateways de sesiones (SWG)* y de *paquetes (PGW)* que mantienen las *sesiones* (comunicaciones) de usuarios y procesan los paquetes entre la red celular e Internet.

La siguiente figura muestra una posible configuración.



Cada UE es *manejada* por una celda. Para establecer una comunicación entre dos UEs se requiere determinar un *circuito virtual* entre los dos nodos extremos (los que manejan cada UE), generalmente usando nodos y oficinas intermedias. Cuando una UE se mueve desde una celda a otra, la celda que logra mejor intensidad de señal *captura* la UE. Si una comunicación estaba en curso, la red debe transferir la sesión de una celda a otra.

La red celular está interconectada a la red telefónica permitiendo comunicaciones entre teléfonos móviles y fijos.

Actualmente, las diferentes versiones de tecnología (3G, 4G, 5G) se especifican en las normas LTE (*Long-Term Evolution*). LTE describe principalmente cómo se utiliza el espectro radio-eléctrico usado por los UEs.

LTE usa una estrategia basada en *reserva de slots* y utiliza *Orthogonal Frecuency-Division Multiple Access (OFDMA)* para datos de *bajada* y *Single Carrier Frecuency-Division Multiple Access (SC-FDMA)* de subida. Este último esquema permite reducir el consumo energético de los UEs.

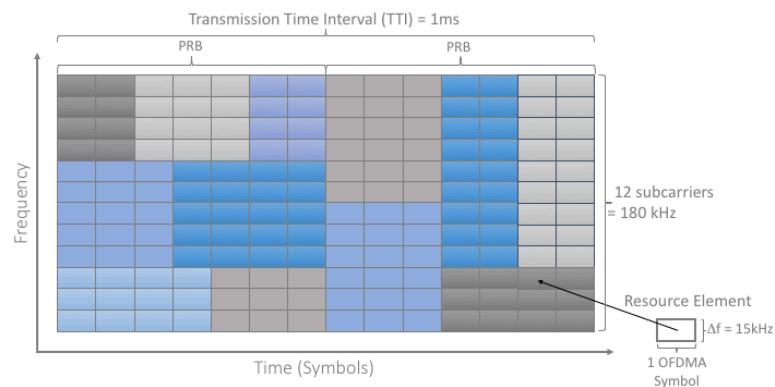
En cada comunicación, la celda asigna a un UE una frecuencia de subida y una de bajada, permitiendo comunicaciones *full-duplex*.

En un canal de subida las comunicaciones de múltiples usuarios se *comparten* usando *TDM*. Un canal de bajada (usando OFDMA) se multiplexa usando *FDM* y *TDM*.

En OFDMA (4G-LTE) se divide un canal en 12 frecuencias ortogonales moduladas independientemente. Cada banda (subportadora) es de 15khz.

La codificación y modulación usadas (*Quadrature Amplitud Modulation*) permiten representar el conjunto de valores discretos o símbolos (4 bits en 16-QAM, 6 bits en 64-QAM).

La siguiente figura muestra una representación bidimensional del esquema usado en OFDMA.



Un *resource element* representa un símbolo transmitido. Los datos se transmiten en bloques de 1ms de duración.

La *asignación de resource elements o slots* a cada UE lo realiza un *planificador (scheduler)*. Este es un problema complejo de optimización y cada operador generalmente utiliza sus propios algoritmos.

En la figura, cada color representa comunicaciones de diferentes UEs.

Las comunicaciones de voz se digitalizan, comprimen y se cifran en cada UE, por lo que la red sólo se basa en comunicación de datos digitales.

Las diferencias entre 3G, 4G y 5G son principalmente los rangos del espectro utilizados y algunas diferencias en las técnicas de codificación, modulación y multiplexado utilizados.

5G, flexibiliza el manejo en base a las distintas bandas (espectro) utilizadas, permitiendo usar diferentes formas de ondas. Las frecuencias bajas pueden usarse para canales de bajo ancho de banda como transmisión de mensajería de texto. Las señales por debajo de 1GHz se utilizan para comunicaciones de largo alcance, las bandas entre 1 y 6GHz ofrecen mayor ancho de banda, principalmente para acceso a Internet o videoconferencia y las frecuencias por encima de los 24GHz ofrecen un gran ancho de banda en distancias cortas (de pocos metros).

## Satélites

El uso de satélites permite interconectar nodos separados por grandes distancias, ya que hacen de *repetidores en gran altura*, visibles desde antenas direccionales terrestres.

Los satélites geoestacionarios se ubican en órbitas cercanas a los 36000km sobre el Ecuador (latitud 0). Giran a una velocidad de 3.07km por segundo para lograr la misma velocidad angular que la superficie terrestre. Así su posición relativa desde un punto en la superficie terrestre se mantiene fija. Esto facilita las estaciones terrenas ya que sus antenas no necesitan movilidad.

Estos tipos de satélites son ampliamente usados en servicios de *broadcasting* como radio y TV. Uno de los principales problemas en el uso de satélites es la *latencia* debido a los *tiempos de propagación de las señales*.

A menudo se utiliza una *constelación* de satélites en órbitas mas bajas. En este escenario, se usan estaciones terrestres con antenas omnidireccionales o direccionales móviles y los satélites forman una red de retransmisores.

Actualmente el mayor porcentaje de las comunicaciones globales se realizan por medio de los tendidos de cables (de fibra óptica) intercontinentales.

---

< Anterior

Enlace de datos

Próximo >

Errores

# Control de errores

Como los canales (medios) de comunicación generalmente no son 100% confiables, es necesario utilizar técnicas de transmisión de datos que incluyan mecanismos para detectar y eventualmente *corregir* errores en un receptor.

Estas técnicas se basan en las *teorías de la información y codificación* desarrolladas por Richard Hamming, Claude Shannon y otros.

Todos los esquemas de detección y corrección de errores requieren que se incluya a los mensajes *información redundante* para que el receptor pueda realizar alguna *verificación de consistencia*.

El diseño de un código corrector de errores requiere definir el conjunto de símbolos válidos. Cada símbolo es de  $n + m$  bits, donde los  $n$  bits son de datos y los  $m$  bits son *datos redundantes*.

**Bit de paridad:** Es el código de detección de errores simple. Se basa en el uso de 1 bit de redundancia a una palabra de  $n$  bits. En *paridad par* (*ímpar*), cada palabra (codeword) *válida* contiene un número par (*ímpar*) de bits.

El desafío es diseñar un código (conjunto de símbolos válidos o *codewords*) que permita *detectar* un símbolo válido o no y en este último caso determinar su cercanía a un código válido permitiendo su corrección.

La *distancia de Hamming entre dos codewords (hd)* es el número de bits en que difieren.

La *distancia de Hamming de un código*  $HD(c) = \forall w_i, w_j \in c | min(hd(w_i, w_j))$ .

Para *detectar*  $d$  errores, se requiere  $HD(c) = d + 1$  (así con  $d$  errores no hay forma de ir a un código válido a otro adyacente).

Para *corregir*  $d$  errores, se requiere  $HD(c) = 2d + 1$ . Esto permite que un código con  $d$  errores se mantenga más cercano a un codeword (válido) que a otro.

Ejemplo: Sea  $c = 000000000, 000001111, 111110000, 111111111$ .

$HD(c) = 5$ , por lo tanto puede corregir errores dobles (cambios en hasta 2 bits).

## Códigos detectores

El uso de códigos simples como el bit de paridad, comúnmente se usa para palabras cortas, comúnmente de longitud de 8 bits. Su uso es común en memorias RAM y en comunicaciones orientadas a bytes, como en comunicaciones serie RS-232.

## Bit de paridad de dos dimensiones

Este esquema estructura la secuencia de datos en una matriz de  $N \times M$ . Adiciona bits de paridad en cada fila y en cada columna. Así se adicionan  $m + n + 1$  bits a los datos a transmitir. Este esquema permite detectar errores en cadena o ráfagas (*bursts*).

Para detectar errores en secuencias de bits mas largas, se han desarrollado otros códigos. A continuación se describen algunos de ellos.

## Checksum

Este algoritmo se usa generalmente en protocolos de red o en capas superiores. Se utiliza en varios protocolos de internet (familia TCP/IP). La idea es muy simple. A cada mensaje se le anexa el resultado de una suma de las *palabras* (por ejemplo, de 16 bits).

La operación utilizada es generalmente la *suma en complemento a uno*.

En aritmética de complemento a uno, un número negativo  $-x$  se representa por su complemento, es decir cada uno de sus bits invertidos. La suma requiere que si existe un acarreo en el bit más significativo, éste debe sumarse al resultado.

Por ejemplo: La suma de  $-5 + -3 = 1010 + 1100 = 0111$ .

El receptor recomputa el checksum del mensaje y lo compara con el campo que contiene el checksum computado por el transmisor. Si son diferentes, ocurrió un error.

Se debe notar que el método no es muy robusto. Un mensaje con palabras erróneas que se *complementen*, es decir, una que incremente su valor y otra que lo decremente, el checksum coincidirá y no se detectará el error. Sin embargo, generalmente funciona en la práctica ya que su aplicación es independiente de la longitud del mensaje.

## CRC

El chequeo de redundancia cíclica (CRC) se basa en interpretar un mensaje de  $n + 1$  bits como un *polinomio de grado n*. Por ejemplo, el mensaje  $x = 10100101$ , se interpreta  $M(x) = x^7 + x^5 + x^2 + 1$ .

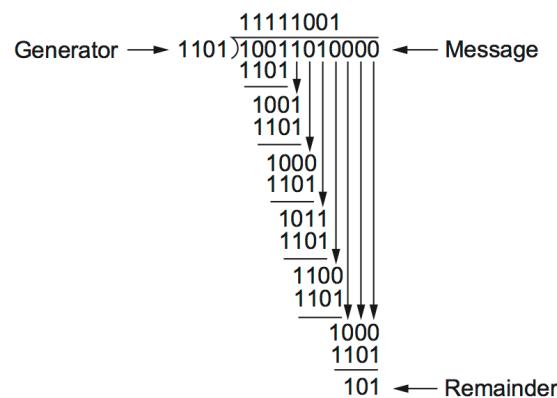
Para calcular el CRC, ambos extremos ( $Tx$  y  $Rx$ ) deben acordar un *generador* (divisor)  $C(x)$  de grado  $k$ . Por ejemplo, el generador 1101 (de grado 3) corresponde a  $C(x) = x^3 + x^2 + 1$ .

El mensaje a transmitir incluyendo el CRC se computa de la siguiente manera:

1.  $T(x) = M(x) \times x^k$ , es decir, anexar  $k$  ceros al final del mensaje.
2.  $R(x) = T(x) \bmod C(x)$  (obtener el resto)
3.  $P(x) = T(x) - R(x)$  (es decir,  $P(x) = T(x) \oplus R(x)$ )

Es fácil ver que  $P(x)$  es divisible por  $C(x)$ . Así el mensaje transmitido ( $P(x)$ ) puede ser verificado por el receptor haciendo  $P(x) \bmod C(x) = 0$ . Si el resto no da cero, ha ocurrido un error en la transmisión.

La siguiente figura muestra la operación del paso 2.



**Notar que en la división módulo 2, la resta realizada en cada paso es equivalente a una operación XOR.**

Un generador de grado  $k$  puede detectar los siguientes tipos de errores:

- Errores simples: Si  $x^k$  y  $x^0$  son unos.
- Errores dobles: Si  $G(x)$  tiene al menos tres términos (no cero).
- Número impar de errores: Si contiene  $x + 1$
- Errores de ráfaga (*burst* o consecutivos): Para ráfagas de longitud de hasta  $k$  bits.

Es común que algunos protocolos a nivel de enlace usen CRC como mecanismo de detección de errores. En particular, Ethernet usa *CRC-32* con el generador  $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 = 0x104C11DB7$ .

## Códigos correctores

Los códigos correctores se conocen como *Forward Error Correction*. Como vimos anteriormente, estos códigos requieren mayor redundancia.

Se pueden clasificar por la forma en cómo operan: por *bloques* o *lineales* (sobre *streams o secuencias* de bits).

## Códigos de Hamming

En 1950, Hamming propuso el siguiente diseño de un código corrector de  $m$  *check bits* ocupando las posiciones potencias de 2 y  $m$  *data bits* en las demás posiciones. Esto hace que califique como un *block code*. Las posiciones se cuentan de izquierda a derecha comenzando desde 1.

Cada *check bit* interviene en la paridad de las posiciones de bit cuya representación binaria lo incluye como se muestra en la siguiente tabla de ejemplo (hasta 15 posiciones). Las columnas  $p$  corresponden a *bits de paridad* y las  $d$  corresponden a bits de datos.

Posición	1	2	3	4	5	6	7	8	9	10	11	12
Codificación	p	p	d	p	d	d	d	p	d	d	d	d
$p_1$	X		X		X		X		X		X	
$p_2$		X	X			X	X			X	X	
$p_4$				X	X	X	X					X
$p_8$								X	X	X	X	X

Por ejemplo, el bit de paridad  $p_2$  se calcula en base a las posiciones 2 (10b), 3 (11b), 6 (110b), 7 (111b), 10 (1010b), 11 (1011b), 14 (1110b) y 15 (1111b), cuyos valores en que su representación binaria el bit en la posición 2 (contando de derecha a izquierda) es 1.

Si el *XOR* de todo el mensaje recibido da cero, el mensaje es correcto, sino hay al menos un error.

En caso de error, el receptor *recalcula* los bits de chequeo. Si el error es en un bit, el valor de la secuencia de *bits checks* como valor numérico (con  $p_0$  como bit menos significativo), indica la posición del bit cambiado. El receptor podrá *intercambiar (flip)* su valor para *corregir* el mensaje.

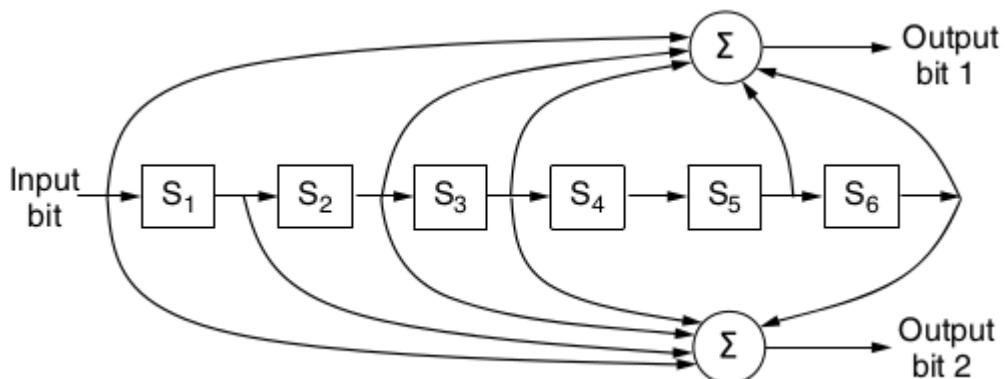
Esta codificación da una distancia de Hamming = 3.

**Ejemplo:** Dado el mensaje 0110 se codificará de la siguiente manera: **--0-110** donde los guiones representan los bits de paridad. El transmisor calcula el mensaje a transmitir: **1100110**. El bit de paridad en la posición 1 (el de más a la izquierda) es la paridad de los bits 3,5 y 7. Suponiendo que el receptor recibe la secuencia **1100010** con errores (cambió el bit de la posición 5), al recalcular los bits de paridad determinará que los bits de paridad que no coinciden son  $p_1$  (da 0) y  $p_4$  (da 1). La suma de estas posiciones indica la posición errónea ( $1+4=5$ ).

## Códigos convolucionales

Son códigos lineales y se usan ampliamente en comunicaciones por radiofrecuencias, como en wi-fi, telefonía celular y comunicaciones satelitales.

La [Figura 4.1](#) muestra un codificador desarrollado por la NASA para la misión Voyager (1977) y actualmente usado en wi-fi 802.11.



### *Figura 4.1: Codificador convolucional.*

La operación se basa en que por cada bit del flujo de entrada, se generan dos bits de salida en base a la *suma (xor)* del bit de entrada con los *bits de estado* almacenados en un *shift register* (en este caso de 6 bits).

Luego de cada *output*, el bit de entrada se *inserta por izquierda* en el *shift register*.

Estos codificadores se caracterizan como  $n/o, k$ , donde  $n$  es la tasa de bits de entrada,  $o$  es la tasa de bits de salida y  $k$  es la *longitud de la restricción*. El codificador de la figura 4.1 es  $1/2, 7$  ya que por cada bit de entrada genera dos de salida y depende de 7 bits en la secuencia de entrada.

Estos códigos pueden formalizarse como una familia de funciones  $c(i, s_k) = o$  donde  $i$  es el *input bit*,  $o$  son los *output bits* y  $s_k$  es el *estado* actual del *shift-register* (que se va actualizando).

La decodificación se basa en algoritmos que determinan la secuencia más cercana a la transmitida tomando la secuencia de bits recibidos y *emulando* todos los posibles estados del codificador. Para más información ver el [Viterbi decoder](#). Estos algoritmos pueden implementarse directamente en circuitos de hardware.

## Códigos Reed-Solomon

Se basan en la idea que un polinomio de grado  $n$  está determinado por  $n + 1$  puntos. Cualquier punto extra es *redundante* y puede ser usado para corregir errores. Por ejemplo, una línea puede representarse por dos puntos. Si se envían 4 puntos sobre la recta, al ocurrir un error, la recta puede reconstruirse a partir de los otros 3.

Un código [Reed-Solomon](#) se basa en la teoría de campos finitos y opera sobre bloques de datos, agregando  $t$  símbolos de chequeo a los bloques, puede *detectar t errores y corregir t/2 errores*.

También se conoce como *erasure codes* ya que permiten que un mensaje de longitud  $n = k + r$ , donde  $k$  es la longitud del mensaje original y  $r$  son los bits de redundancia pueda reconstruirse desde un subconjunto de  $n$  símbolos.

Estos códigos se usan comúnmente en dispositivos de almacenamiento como Redundant Array of Inexpensive Discs (RAID), CDs y DVDs, códigos QR. También lo usan algunas implementaciones de *Digital Subscriber Line (DSL)* para la transmisión de datos digitales sobre líneas telefónicas.

Un código de redundancia cíclica califica como un código Reed-Salomon pero éstos últimos incluye otros no cílicos.

El esquema usado en RAID se basa en que un bloque se almacena en forma redundante en dos discos, sean  $A$  y  $B$ . En un tercer disco se puede almacenar  $C = A \oplus B$ . Así, ante la falla de uno se puede reconstruir otro gracias a las propiedades del operador XOR ( $\oplus$ ) ta que  $C \oplus A = B$  y  $C \oplus B = A$ .

---

< Anterior

Infraestructura

Próximo >

Internetworking

# Internetworking

La interconexión de diferentes redes generalmente *heterogéneas*, es decir, cada una con sus propios protocolos y tecnologías de comunicación, requiere la definición de protocolos que permitan abstraer las diferencias en las tecnologías y topologías de cada subred.

Los dispositivos de interconexión de redes toman diferentes nombres en base al nivel de la capa de red del modelo OSI en que operan. Si operan en la capa 2 (enlace de datos) generalmente se denominan *switches* y si operan en capa 3 (*red*) se denominan *routers*. Para independizarnos de la capa, los llamaremos *gateways*.

Las técnicas de ruteo pueden clasificarse en tres estrategias:

- Ruteo basado en *datagramas*
- Establecimiento de *circuitos virtuales*
- Ruteo de origen

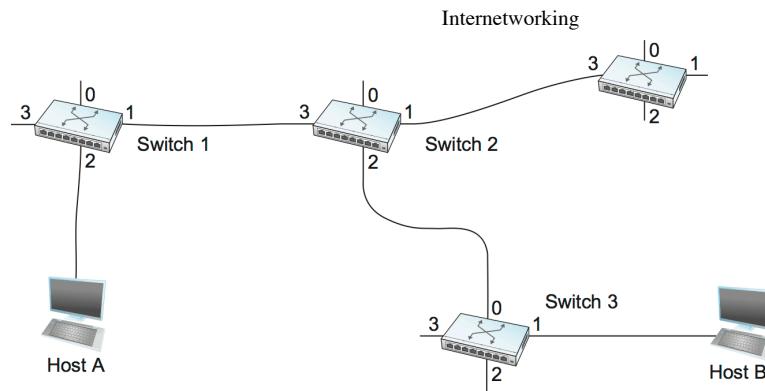
La primera estrategia se basa en que la decisión de ruteo se basa en la dirección de destino del paquete. Cada *gateway* se basa en una *tabla de ruteo (forwarding table)*, definida estáticamente o dinámicamente que indica por qué puerto/dispositivo tiene que reenviar el paquete.

El problema de cómo se definen esas tablas se conoce como *routing*.

La segunda estrategia se basa en la idea de un *servicio orientado a conexión*. Si dos nodos deben intercambiar datos, se debe establecer un *circuito virtual* entre ellos, incluyendo los *gateways* intermedios.

Un circuito virtual queda representado en forma distribuida por una entrada en una tabla de cada gateway. Cada entrada está definida por la tupla *<vc-id, in-port,out-port>*.

Una vez establecido un circuito virtual entre dos nodos, cada paquete incluye un *identificador del circuito virtual*, el cual será usado por cada *gateway* para determinar a qué otro debe reenviarlo. La siguiente figura muestra un ejemplo de los circuitos virtuales establecidos.



**Figura 7.1: Ejemplo de circuitos virtuales.**

El establecimiento de los circuitos virtuales generalmente se hace en forma automática. El protocolo incluye mensajes de *signaling* para definir un circuito entre dos nodos (*hosts*). Esto requiere que inicialmente los gateways *separan* alcanzar al nodo destino, por ejemplo, usando una *tabla de ruteo*.

Las redes basadas en Asynchronous Transfer Mode (ATM), actualmente en desuso, se basa en el uso de circuitos virtuales.

La última estrategia, conocida como *source routing*, se basa en que el emisor incluye el *camino* o la secuencia de gateways intermedios a usar para alcanzar el destino. Este esquema requiere que cada nodo conozca la *topología de la red* y hace que la *cabecera (header)* de un paquete sea de tamaño variable.

El protocolo IP incluye una opción para soportar esta estrategia. La aplicación *Unix-to-Unix Copy (uucp)* de base en *source routing*.

## Switchs Ethernet

Ethernet permite interconectar diferentes segmentos mediante *switches* enlazados entre sí. Esto define un protocolo de interconexión de redes de *nivel 2 (L2)*, ya que se define en un protocolo de enlace de datos (capa 2 del modelo OSI).

En grandes organizaciones, la interconexión de muchas redes Ethernet hace que su rendimiento decaiga ya que los paquetes de *broadcast* (con destino a todos los demás nodos) y *multicast* (con destino a un grupo de nodos) generalmente se retransmiten por toda la red. El protocolo de *ruteo* (o *forwarding*) de frames Ethernet usados en switches, permite que cada uno *construya dinámicamente* una *tabla de ruteo* que mapea las direcciones de destino de cada nodo/switch con cada *puerto* del switch.

Este protocolo requiere resolver el problema de *circularidad* que haría que algunos frames circularan indefinidamente por la red al no encontrar un nodo con la dirección de destino.

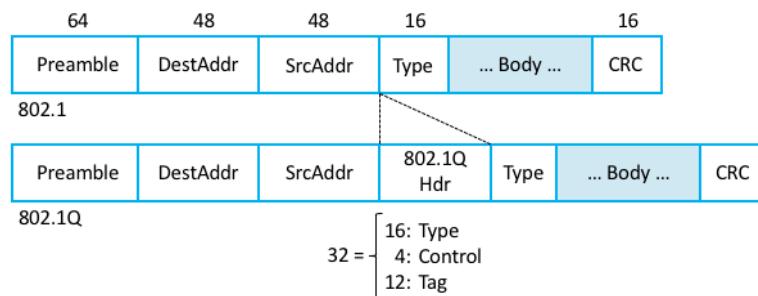
Para eso, cada *switch* adquiere un *rol* por medio de *identificadores*, formando una *topología virtual de árbol* (el switch con menor identificador es el raíz) construyendo un *spanning tree*.

## VLANs

Una *virtual lan (VLAN)* es una tecnología para definir *redes virtuales* sobre redes a nivel de enlace (L2), ampliamente usada en redes Ethernet.

Permite que una LAN (posiblemente extendida) pueda *particionarse* en diferentes *redes lógicas*. A cada VLAN se le asigna un identificador único y se usa para el *reenvío de frames*. Un frame se reenvía de un segmento a otro si ambos segmentos tienen el mismo identificador.

La versión del protocolo Ethernet 802.1Q soporta el manejo de VLANs, extendiendo el formato del frame como se muestra en la siguiente figura.



**Figura 7.2: Formato de frame Ethernet 802.1Q.**

## Internet

Internet es actualmente la *red de redes* y es obviamente la red mundial más amplia del mundo. Interconecta subredes de diferentes tecnologías utilizando la familia de protocolos TCP/IP, desarrollados principalmente por Vinton Cerf y Robert Khan en la década de 1970 en un proyecto financiado por DARPA.

Esta familia de protocolos se basa en un diseño mas simple que el modelo OSI, organizado en 4 capas. La capa inferior, denominada *capa de enlace* cubre la capa física y la de enlace de datos del modelo OSI.

La capa de red está representada por el protocolo IP y un conjunto de protocolos auxiliares que se describen en el próximo capítulo. Esta capa provee un servicio orientado a *datagramas* no confiable, es decir no garantiza la entrega de paquetes a destino ni el orden.

La capa de transporte incluye los protocolos UDP y TCP. El primero simplemente extiende IP para el multiplexado de paquetes con los procesos de cada host. TCP ofrece un servicio orientado a conexión confiable, es decir, garantizando la entrega ordenada de paquetes al receptor con retransmisiones de paquetes perdidos o con errores.

Las capas de sesión y presentación quedan como responsabilidad de las aplicaciones.

---

< Anterior

## Errores

Próximo >

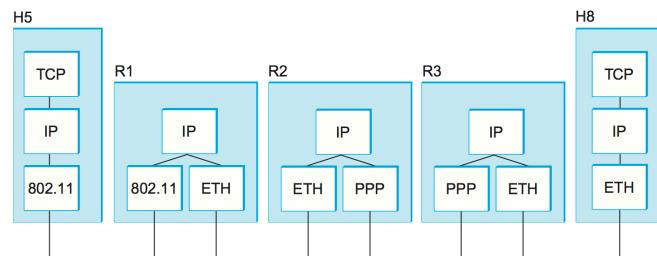
## Internet Protocol

# Internet Protocol

El protocolo **IP** permite la interconexión de redes heterogéneas (*internet o redes de redes*) proveyendo una abstracción independiente de la tecnología de comunicación subyacente.

Los paquetes generados en una capa se *encapsulan* en un paquete de la capa inferior, tal como se muestra en la siguiente figura. Esto permite abstraer la tecnología usada en la capa de enlace de datos y permite la inter-operabilidad entre redes diferentes. Los *gateways (routers)* intermedios retransmiten (*store-and-forward*) los paquetes IP encapsulados en los frames de cada tecnología usada en cada enlace hasta llegar al destino.

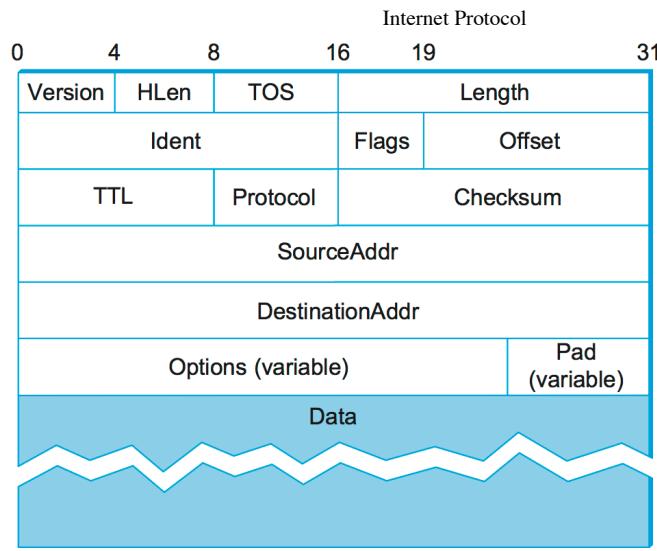
Este último, analizando la dirección de destino lo entrega al módulo que implementa el protocolo en la capa superior; en este ejemplo, TCP.



Es un protocolo simple que permite *escalabilidad* y se basa en los siguientes principios:

- Describe un esquema de *direccionamiento de hosts y gateways*.
- Provee un servicio de *datagramas de mejor esfuerzo* en la entrega de paquetes, es decir que no ofrece garantías (servicio *no confiable*).

El formato de un paquete IP (*IP datagram*) se muestra en la siguiente figura.



**Figura 8.2: Formato de un datagrama IP (versión 4).**

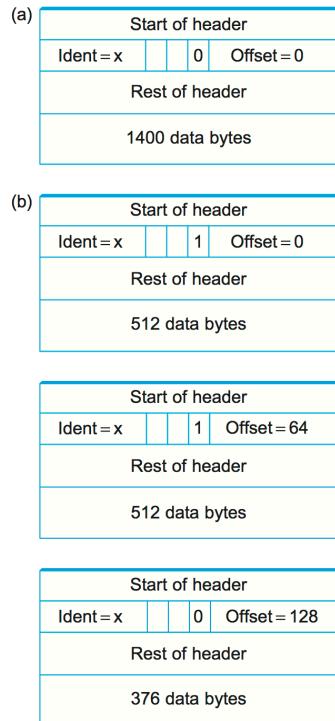
Un datagrama IP contiene las direcciones *origen* y *destino*. En la versión 4 (IPv4) las direcciones son de 32 bits. El campo `hlen` especifica la longitud del *header* (de longitud variable) en palabras de 32 bits. El campo `TOS` indica el *tipo de servicio* requerido por la aplicación. El campo `Length` define el tamaño en bytes del datagrama, incluyendo el encabezado.

Los campos `Ident`, `Flags` y `Offset` se utilizan para la *fragmentación* y *reensamblaje* de datagramas.

Esto ocurre cuando el protocolo de enlace usa frames de menor tamaño que un datagrama IP. Por ejemplo, Ethernet tiene un tamaño máximo (*Maximum Transmission Unit o MTU*) de su *payload* de 1500 bytes, sin embargo en un datagrama IP puede ser de hasta  $2^{16} = 65536$  bytes.

Cuando un datagrama tiene tamaño mayor que la MTU del protocolo de enlace, IP debe particionar y enviar el datagrama en *fragmentos*. Cada fragmento tiene el mismo `Ident`. El `Offset` indica el orden del fragmento en bloques de 8 bytes. En cada fragmento, excepto el último, se setea el bit `M` en el campo `Flags` indicando al receptor que restan fragmentos.

La siguiente figura muestra la fragmentación de un datagrama IP con un tamaño de 1420 bytes (20 bytes del header + payload de 1400 bytes) en un enlace con una MTU=532 bytes.



*Figura 8.3: a) Datagrama IP original. b) Fragmentos.*

El receptor debe *reensamblar* el datagrama IP aunque los fragmentos arriben fuera de orden, por lo que deberá memorizar los fragmentos para luego rearmar el paquete IP antes de entregarlo (*delivery*) al protocolo de la capa superior.

**MTU discovery:** Proceso para descubrir el menor MTU en un *camino* desde el emisor al receptor para evitar la fragmentación.

Ejercicio: ¿Cómo se puede hacer con el comando **ping** ?

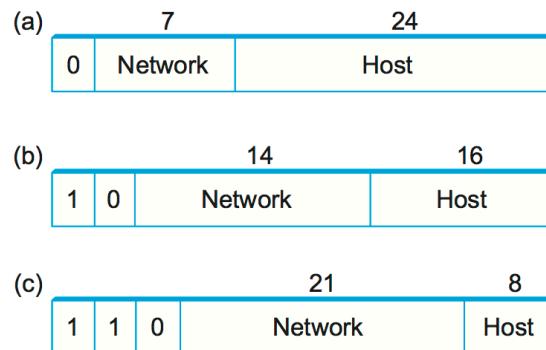
## Direcciones IP

El direccionamiento IP se basa en un esquema jerárquico. Cada dirección se divide lógicamente en dos partes: El *identificador de red* y el *identificador de host* dentro de esa red. Las direcciones en IPv4 son de 32 bits y se denotan usando los valores decimales de cada uno de sus cuatro bytes separados por puntos. Por ejemplo: 192.168.0.15.

Una dirección IP se asigna a una interface (NIC) del host.

Un *router* es un host que pertenece a más de una red. Cada una de sus *interfaces* se conecta a una de las redes y se les asignan direcciones con la parte de red de manera acorde.

Inicialmente, las direcciones IP se dividieron en *espacios (clases)*, como se muestra en la siguiente figura, indicando el número de bits para el identificador de red y host.



**Figura 8.4: Clases de direcciones IPv4. a) A, b) B y c) C.**

En cada clase reserva un rango de *direcciones privadas* (class A: 10.0.0.0-10.255.255.255, B: 172.16.0.0-172.31.255.255 y C: 192.168.0.0-192.168.255.255), las cuales no son *ruteadas* y permiten *reusarse* en diferentes subredes. Esto permite que no todos los hosts y routers internos (en *intranets*) usen *direcciones públicas*, permitiendo mitigar la escasez actual de direcciones IPV4.

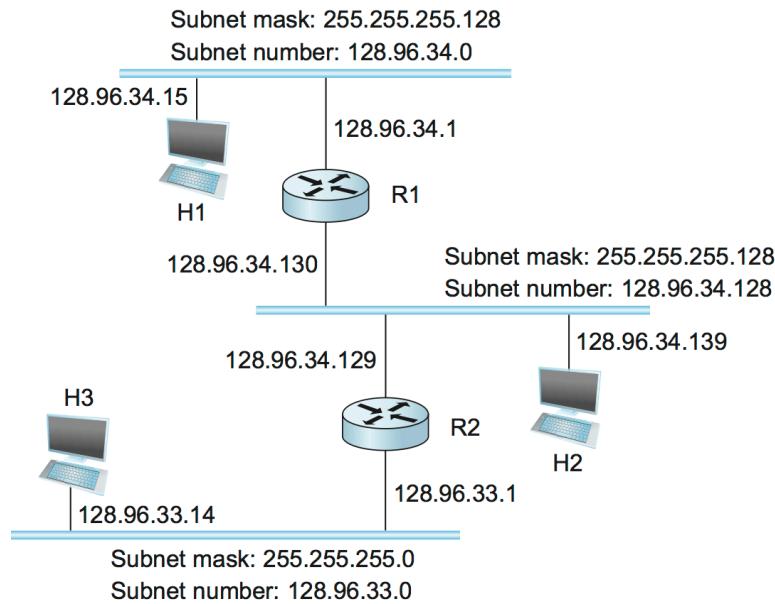
**Masquerading:** Proceso que hace que un *gateway (router)* reenvíe datagramas con destino a Internet, recibidos de hosts internos con direcciones privadas re-escribiendo la dirección de origen con su dirección pública, permitiendo su ruteo. Esto funciona a nivel de transporte ya que se usa el número de puerto del paquete saliente para registrar (en una tabla) el envío.

Al recibir la respuesta (con IP destino/port del gateway) éste busca en la tabla par (*internal-ip, port*) registrado y reescribe la dirección de destino con *internal-ip*, lo que hará que se retransmita al emisor original.

El *puerto* se encuentra en la cabecera del protocolo de transporte TCP o UDP, los cuales se analizan en el capítulo de [transporte](#).

Este es un caso particular de un proceso más general denominado *Network Address Translation (NAT)*.

En cada clase es posible configurar subredes, tal como se muestra en la siguiente figura. Esto se realiza extendiendo el número de bits usados para representar identificadores de *subredes*. En la [figura 8.5](#) se muestran subredes dentro de la clase C, habiendo extendido 1 bit la parte de identificación de red para dividir la red 128.96.34 en dos subredes (128.96.34.0 y 128.96.34.128).



*Figura 8.5: Ejemplo de subnetting.*

La máscara de subred se denota indicando los bits que participan en el identificador de red en uno y los de host en cero. Por ejemplo la máscara de subred 255.255.255.128 indica que se usan 25 bits para el identificador de red.

El esquema basado en clases ha mostrado ser demasiado rígido y no siempre permite la asignación mas adecuada de direcciones IP, por lo que actualmente una dirección IP se puede dividir en las dos partes de manera arbitraria, permitiendo definir a los administradores de una red el número de bits a usar para la identificación de sus subredes. Este esquema se conoce como *classless interdomain routing (CIDR)*.

En CIDR, las redes se denotan de la forma *network-id/x*, donde *x* es el número de bits de la parte de red. Por ejemplo, la notación **192.168.34/24** denota la red 192.168.34 de clase C y la notación **192.4.16/20** denota las redes desde 192.4.16 a 192.4.31.

Esto permite especificar en forma compacta un conjunto de redes y reduce el número de entradas en las tablas de *ruteo*.

## Forwarding

Cada router debe determinar en el arribo de un datagrama si debe ser reenviado por otra interface o entregado al módulo del protocolo de transporte local. En el caso que deba reenviarse, el proceso se denomina *forwarding* y está implementado tanto por routers como en cada host.

Desde la tabla de *ruteo* (configurada a nivel de red), se extraen las entradas correspondientes a las interfaces del router en una *tabla de forwarding* para poder determinar eficientemente el camino que debe seguir un paquete recibido.

La tabla de *forwarding* tiene entradas como se muestra a continuación.

Net	Mask	Gateway	Ifce
128.96.34.0	255.255.255.0	0.0.0.0	eth0
128.96.34.128	255.255.255.128	0.0.0.0	wlan0
0.0.0.0	0.0.0.0	210.134.17.5	eth1

**Tabla 8.1: Ejemplo de una tabla de forwarding.**

La tabla muestra un ejemplo de un gateway que se conecta a un ISP por el enlace `eth1` e interconecta dos subredes mediante las interfaces `eth0` y `wlan0`. La última entrada se conoce como la ruta por omisión o *default gateway*.

A continuación se muestra un esquema del algoritmo.

```
python

def ip_forward(datagram):
    s = datagram.src_ddr
    d = datagram.dst_addr
    if datagram.ttl == 0:
        return icmp_send(s, TIME_EXCEEDED)
    for (dst, mask, gw, ifce) in forwarding_table:
        m = d & mask
        if m == dst:
            if gw != 0:
                # send to gateway
                if not data_link_send(dev=ifce, dst=ARP(gw), data=da
                    # send ICMP error message to sender
                    return icmp_send(s, NETWORK_UNREACHABLE)
            elif (ifce.ip & mask) != 0:
```

```

# local delivery
if not data_link_send(dev=ifce, dst=ARP(d), data=dat
    # send ICMP error message to sender
    return icmp_send(s, HOST_UNREACHABLE)
else:
    return OK
# no match, send ICMP error to sender
return icmp_send(s, NO_ROUTE_TO_HOST)

```

### *Listado 8.1: Algoritmo simplificado de forwarding.*

El algoritmo de forwarding busca una entrada en la tabla hasta encontrar una entrada cuyo prefijo de la dirección de destino coincida con el valor de la primer columna. Luego se solicita al protocolo de enlace de datos (L2) correspondiente el envío del paquete. El ejemplo muestra el uso de un protocolo auxiliar, el *Address Resolution Protocol (ARP)* para obtener la dirección MAC asociada a la IP del gateway de destino del próximo salto (hop). Más abajo se describe ARP más en detalle.

Si el campo **TTL** (*time to live*) alcanzó el valor cero, se descarta por haber alcanzado el máximo número de saltos (re-envíos). Esto evita que algún datagrama continúe reenviándose indefinidamente por no encontrar su destino.

Se debe notar que en el caso que si la tabla de forwarding incluye el *default gateway* siempre se logrará un *matching* con esta entrada.

En el caso de *CIDR* el algoritmo debe modificarse (o mantener la tabla ordenada apropiadamente) para que se seleccione la entrada de la tabla con matching del prefijo más largo.

## Entrega local

En la recepción de un paquete un host o gateway determina si debe consumirlo o reenviarlo.

En el caso que la dirección de destino del paquete coincida con la dirección IP asignada a algunas de sus interfaces de red, se debe entregar al módulo de transporte indicado en el campo **Protocol** del datagrama.

En otro caso, si se debe enviar a otro host de su misma red local y en el caso que ésta sea del tipo Ethernet (o similar) se debe encontrar la dirección MAC correspondiente al host con dirección de destino en el datagrama IP.

Este proceso se delega en el ***Address Resolution Protocol (ARP)***, definido en la [RFC 826](#) el cual puede entenderse como una implementación de la siguiente función:

$$\text{ARP}(ip\_address) \rightarrow \text{data\_link\_address}$$

ARP implementa esta función enviando un mensaje ARP de tipo "*who has this ip-address?*" a todos los hosts de la red local en un datagrama IP de *broadcast* (dirección de destino 255.255.255.255). Esto hará que la capa de enlace lo encapsule en un *broadcast frame* (destino 0xff:0xff:0xff,0xff,0xff,0xff) el cual será recibido por todos los hosts de la red local.

El host que tenga esa IP asociada a una interfaz, responde con un mensaje que contiene la dirección de enlace (ej.: Ethernet MAC) correspondiente. Así, ahora el emisor puede encapsular el datagrama IP en un frame de enlace de datos con la dirección MAC de destino correspondiente.

ARP usa una tabla caché para *recordar* las resoluciones previas.

Este protocolo no se utiliza en el caso de enlaces del tipo PPP, ya que no hace falta determinar la dirección de destino.

Otro protocolo auxiliar, el ***Reverse Address Resolution Protocol (RARP)***, descripto en la [RFC 903](#) realiza la función inversa, es decir  $\text{RARP}(\text{DataLinkAddress}) \rightarrow \text{IPAddress}$ . RARP se usa en el protocolo [\*\*bootp\*\*](#) de *booting* remoto en dispositivos sin disco.

## Configuración de Hosts

Un host requiere una configuración de red para cada una de sus interfaces. Cada interface se asocia con una dirección IP y una máscara de subred. Además se requiere al menos una tabla de forwarding mínima (al menos conteniendo la entrada de un *default gateway*).

Generalmente en un host el sistema operativo crea una *interface virtual* (por software) llamado *localhost* o *loopback*. Esta interface se asocia con la dirección IP 127.0.0.1 y se usa para la comunicación entre procesos locales del host.

Para evitar recordar direcciones IP asociadas a cada host, se pueden asignar *nombres*. En los sistemas tipo UNIX, este mapping puede definirse en el archivo `/etc/hosts`.

En redes grandes como Internet, es necesario usar el *Domain Name Service (DNS)* que implementa la función  $DNS(domain) \rightarrow ip\_address$ . Este protocolo a nivel de aplicación se analiza mas adelante en estas notas.

Resumiendo, la configuración de una interfaz en un host consiste en:

1. Dirección IP.
2. Máscara de red.
3. La dirección IP de al menos un servidor de nombres (DNS). Generalmente se definen dos, uno como DNS primario y otro secundario (en Linux, ver el archivo `/etc/resolv.conf`).

Además hay que configurar la *tabla de ruteo* que describe cómo se alcanzan otras redes asociadas a cada interfaz como se describe arriba.

Cada sistema operativo ofrece comandos para configurar la red.

- Configuración de interfaces: `ifconfig` en sistemas tipo UNIX, `ipconfig` en MS-Windows.
- Configuración de rutas: El comando `route` comúnmente se usa para agregar, modificar o eliminar rutas.

En sistemas GNU-Linux, el comando `ip` (hacer `man ip`) permite configurar prácticamente todos los elementos de una red IP. Los archivos que contienen la configuración de red son los siguientes:

Archivo	Descripción
<code>/etc/hosts</code>	Mapping de direcciones IP a host names (dominios)
<code>/etc/resolv.conf</code>	IPs de servidores de nombres (primario y secundario)
<code>/etc/network/interfaces</code>	Configuración estática de interfaces
<code>/etc/network/if-*</code>	Scripts para manejo de interfaces ( <code>up/down, ...</code> )

La configuración de un host puede ser manual, en archivos de configuración dependientes del sistema operativo, o automática.

Para este último caso, existe el **Direct Host Configuration Protocol (DHCP)**.

DHCP requiere un *servidor* y *clientes*. Comúnmente un *gateway* (router) implementa un servidor DHCP. Cada host implementa un cliente DHCP. Los switchs/routers basados en Ethernet/wi-fi usados comúnmente proveen el servicio DHCP.

Un host en su inicio del subsistema de red envía un mensaje **DHCPDISCOVER** broadcast (`dst_ip=255.255.255.255, src_ip=0.0.0.0`). El host/gateway que esté corriendo el servidor DHCP responde con un mensaje **DHCPOFFER** el cual comúnmente contiene la dirección IP ofrecida y la máscara de subred, la IP del *gateway* e IPs de los DNS.

El cliente si acepta el mensaje responde al servidor con un mensaje **DHCPREQUEST** indicando que aceptó la configuración recibida del server. Esto permite seleccionar otros mensajes **DHCPOFFER** recibido por otros posibles DHCP servers. Finalmente el servidor responde al cliente con un mensaje final **DHCPPACK** que incluye el *lease time* (*tiempo de validez*) y alguna otra información que el cliente pudiese haber requerido.

Un servidor DHCP usa una base de datos (archivo de configuración) que describe los rangos de direcciones IP a asignar a los hosts y mantiene en su estado las direcciones asignadas.

Es posible *fijar* la asignación de una dirección IP a un host determinado usando su MAC.

## Notificaciones de errores

En todo protocolo pueden ocurrir errores y deben ser notificados. Por ejemplo, en el **algoritmo de forwarding** se debe notificar un error si un datagrama agotó su *tiempo de vida* o no se encuentra el destino o ruta de reenvío.

Para esto se usa un protocolo auxiliar, el **Internet Control Message Protocol (ICMP)** definido en la [RFC 792](#) el cual define un conjunto de tipos de mensajes de error y control de ICMP.

Los mensajes de control más importantes son:

- **Redirección:** Indica al host emisor que existe una *mejor ruta* por medio de otro router, indicando su dirección IP.

- **Echo:** Los mensajes *echo-request* y *echo-reply* se utilizan para verificar alcanzabilidad. Usados en las aplicaciones `ping` y `traceroute`.

Algunos de los mensajes de error son

- **Destination unreachable:** Indica que el host, red o número de puerto es inalcanzable.
- **Time exceeded:** El campo `TTL` (*time to live*) llegó a cero.
- **Packet too big:** El paquete tiene una longitud mayor que el que puede manejar la red por la que tiene que transitar. Esto permite al emisor determinar la MTU a utilizar.

## IPv6

La nueva versión de IP es la 6 y se conoce como IPv6. Este protocolo no es compatible con IPv4 y pretende las siguientes mejoras:

1. **Espacio de direcciones mayor:** Las direcciones son de 128 bits y se usa una notación en 8 grupos de 4 dígitos hexadecimales separados por `:`. Los grupos consecutivos en cero pueden omitirse usando como separador `::`. Cada dispositivo puede ser alcanzado globalmente.

Una dirección también se separa lógicamente en dos partes. Al menos 64 bits más significativos para el *identificador de red* y el resto representan el *identificador de host* en esa red.

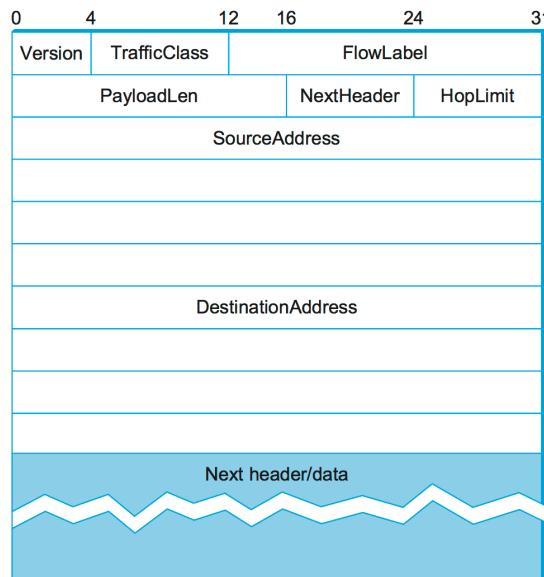
2. Simplifica las comunicaciones *multicast* (a todos los hosts conectados al enlace correspondiente de la LAN) utilizando la dirección `ff02::1` y utiliza *grupos* de hosts.

3. **Configuración automática:** Cada host utiliza el protocolo [Neighbor Discovery Protocol](#) Este protocolo opera en la capa de *enlace de datos*. Define mensajes para obtener de un router local el *prefijo de red*. Cada host define el *host id* desde la dirección MAC de la interface u otros mecanismos.

4. **Encabezado simplificado:** Provee mayor flexibilidad mediante las *extensiones* opcionales. La [figura 8.6](#) describe el *encabezado* de un paquete IPv6.

5. Mayor soporte para **calidad de servicios**: Su cabecera contiene información sobre el *tipo de servicio requerido* en el campo `Traffic class`.

6. Soporte para **flujos de paquetes**: Secuencias de paquetes relacionados que representan diferentes *flujos* como por ejemplo, sonido, video y datos en aplicaciones de streaming multimedia o videoconferencia.



**Figura 8.6: Formato de la cabecera de un paquete IPv6.**

La siguiente tabla muestra los *prefijos* definidos para representar diferentes tipos de direcciones:

Prefijo	Uso
::	No especificado
::0001	Loopback interface
FFxx: ...	Direcciones multicast
FE2x: ...	Multicast (en la red local)
otra	Direcciones de hosts (unicast)

Se debe notar que la cabecera no contiene un *checksum*, como IPv4. La integridad de los datos se traslada a los protocolos en capas superiores.

El campo **next header** permite formar una lista enlazada de *extensiones*. Entre las extensiones pre-definidas están las que brindan soporte para el uso de protocolos de seguridad como **IPSec**, extensiones para fragmentación, o *ruteo basado en el origen*, entre otras.

El ruteo es jerárquico, similar a IPv4 con CIDR. Los *registros* en el sistema DNS para direcciones de hosts/routers se denotan con el tipo **AAAA** (a diferencia de **A** para IPv4).

# Asignación de IPs y dominios

El IANA es la organización encargada de asignar rangos de direcciones IP en el mundo. La distribución se hace por regiones. Cada ISP gestiona rangos de direcciones ante *Internet registry* (locales o regionales). Cada usuario obtiene sus direcciones IP desde su ISP.

Un ISP puede asignar direcciones públicas a sus usuarios (a la interfaz externa de su modem/router), aunque generalmente las asignan dinámicamente (en base a las que están en desuso actualmente), por lo cual las direcciones pueden cambiar en el tiempo. Un usuario que desee una IP pública estática generalmente deberá adquirir un plan especial, comúnmente a un mayor costo.

Para simplificar el acceso a las aplicaciones (servicios) en red, en lugar de hacerlo utilizando su IP (la cual podría cambiar), se utilizan nombres. Un **dominio**, determina la IP de un host o un servicio. El nombrado de hosts o servicios es jerárquico y se describe en detalle en el capítulo de aplicaciones. Por ejemplo, el dominio `dc.exa.unrc.edu.ar` determina la IP de un host del departamento de computación de la facultad de ciencias exactas de la Universidad Nacional de Río Cuarto, la cual es una organización educativa en Argentina.

Las aplicaciones usan el servicio **Domain Name System (DNS)**, el cual es un sistema de base de datos distribuido global que dado un dominio resuelve a la IP correspondiente.

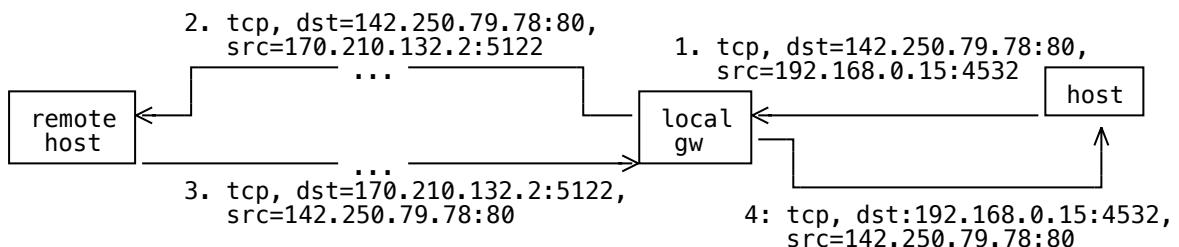
*¿Cómo podría implementar un servidor en mi organización u hogar con direcciones dinámicas?*

Es posible utilizando un servicio de DNS dinámico (existen muchos gratuitos), el cual permite asociar una dirección con un dominio, donde el mismo servidor (mediante un simple script) del usuario *actualiza* ese par en el DNS ante un cambio de direcciones.

## Network Address Translation (NAT)

NAT se usa ampliamente con IPv4 ya que las direcciones públicas ya se han asignado en su totalidad y es común que una *intranet* (red interna hogareña o de una organización) use direcciones *privadas*. El problema con el uso de direcciones privadas es cómo hacer que un datagrama enviado desde un host interno a través de un gateway con destino a un servicio, logre recibir la respuesta, ya que el servidor no podría enviar un datagrama a una IP privada fuera de su red.

La solución se basa en que el gateway (router) de salida *reescriba* la dirección origen (par *ip:port*) del host emisor con su propia dirección IP externa (pública), posiblemente con un nuevo port generado. Se usa una *NAT table* para recordar la traducción para que luego, al recibir una respuesta desde el destino, el router pueda *reenviar* el datagrama al host interno, como se muestra en la siguiente figura.



local gateway NAT table			
Prot	Destination	Source	Internal
...			
tcp	142.250.79.78:80	170.210.132.2:5422	192.168.0.15:4532
...			

Este tipo de NAT se conoce como *masquerading* ya que el gateway *enmascara* el paquete enviado como si lo hubiese emitido él. Se hace *source nat (snat)* en el envío de paquetes y *destination nat (dnat)* en cada respuesta.

En una respuesta, un router puede utilizar alguna de las siguientes implementaciones. Sea *NAT* la tabla NAT del router.

1. **One-to-One NAT:** También conocida como *full cone NAT*. El router al recibir un paquete *pkt*, existe *i*, tal que: *pkt.protocol==NAT[i].prot* y *NAT[i].source==pkt.dst\_ip:dst\_port*. Luego, reescribe *pkt.dst\_ip:dst\_port=NAT[i].internal* reenviándolo al host interno.
2. **Symmetric NAT:** El router reenvía al host interno un paquete *pkt* recibido sólo si existe *i*, tal que: *pkt.protocol==NAT[i].prot*, *pkt.dest\_ip:dest\_port==NAT[i].source* y

$pkt.src\_ip:src\_port==NAT[i].destination$ . Esto significa que el router debe crear una nueva entrada NAT que relaciona cada dirección interna con cada destino remoto diferente. El router reenvía un paquete de un host externo a un host interno sólo si éste le envió un mensaje previamente.

Esta funcionalidad comúnmente está habilitada en cualquier router provisto por el ISP. *Full cone NAT* permite comunicaciones entre hosts (*peers* con direcciones IP privadas) detrás de routers. *Symmetric NAT* no hace posible esta comunicación entre *peers* detrás de routers con NAT. Al menos uno de ellos deberá tener una dirección IP pública.

## Port forwarding

La técnica de *port forwarding* es una técnica de *destination nat* que permite que una aplicación (servicio) interno (con IPs privadas) pueda alcanzarse desde el exterior.

En éste caso se debe crear una entrada en una tabla en el gateway (router) local para que *reenvíe* los paquetes entrantes (con destino *public-ip:out-port*) al host interno correspondiente (*internal-ip:in-port*).

Comúnmente los routers provistos por los ISPs proveen una interfaz de administración (comúnmente web) accesible con un navegador web a la IP interna del router (comúnmente 192.168.0.1 o 192.168.1.1) como se muestra en la siguiente figura.

Aplicaciones>Port Forwarding

Nombre de la Aplicación	Custom settings	
Puerto WAN	<input type="text"/>	<input type="text"/>
Puerto LAN	<input type="text"/>	<input type="text"/>
Cliente Interno	Custom settings	<input type="text"/>
Protocolo	TCP	
Lista de Conexión de Banda Ancha - WAN	1_INTERNET_R_VID_33	
Descripción	<input type="text"/>	
<input type="button" value="Agregar"/>		

Nombre de la Aplicación	Conexión WAN	Puerto WAN	Puerto LAN	Nombre del Dispositivo	Cliente Interno	Protocolo	Descripción	E
Customer settings	1_INTERNET_R_VID_33	8080~8080	8000~8000	marcelo-linux-desktop	192.168.1.13	TCP	My web server	A

En la imagen se puede observar que es posible acceder a un servicio (web) desde el exterior mediante la IP externa pública (WAN) al puerto 8080. El router redirige cada paquete al host interno 192.168.1.13 (privada) al puerto 8000 en el que la aplicación está escuchando.

## Aplicaciones

NAT es útil para resolver varios problemas como

- **Masquerading:** Usado para mitigar el problema de la falta de direcciones IPv4 públicas.
- **Balance de carga:** Un router puede redirigir paquetes de clientes a diferentes servidores permitiendo distribuir el procesamiento de los requerimientos.
- Establecer conexiones desde servidores a clientes detrás de routers NAT. Aplicaciones como File Transfer Protocol (FTP) y Session Initiation Protocol (SIP) requieren esta capacidad.
- **Port forwarding** como se explica en la sección anterior.
- **Packet filtering:** Filtrado de paquetes con destino a puertos específicos de servicios. Reglas de este tipo comúnmente se usan en *firewalls*.

---

< Anterior

Internetworking

Próximo >

Ruteo

# Ruteo

Los protocolos de *internetworking* atacan el problema de abstraer la heterogeneidad de las tecnologías de cada red.

Otro problema a resolver es la *escalabilidad*. El algoritmo de *forwarding* usado en IP es muy simple y eficiente y se basa en una tabla dada. En pequeñas redes esas tablas pueden definirse estáticamente (ver los comandos `ip route` en GNU-Linux). En grandes redes, como Internet, es necesario que los *routers* descubran automáticamente las rutas a ciertos destinos y sus características, con el objetivo de lograr encontrar la *mejor ruta* para que los paquetes alcancen su destino.

La diferencia entre una *tabla de forwarding* y una *tabla de ruteo* es que la primera contiene información específica útil para realizar el *reenvío* de paquetes, mientras que la segunda generalmente contiene una descripción (parcial) de *alcanzabilidad* de redes y describe una *visión local* de una parte de la red.

El problema de la construcción de *tablas de ruteo* dinámica y automáticamente, se denomina *ruteo* y en éste capítulo se analizan diferentes algoritmos y aplicaciones distribuidas (protocolos) para resolver este problema.

## Redes y grafos

Un *grafo*  $G = \langle N, E \rangle$ , donde  $N$  es un *conjunto de nodos* (o vértices) y  $E$  es un *conjunto de arcos* se usa comúnmente para representar una red. Los *nodos* representan *hosts* y *gateways* y los arcos representan *enlaces*. Los *arcos* son *bidireccionales* y pueden tener atributos asociados como por ejemplo su *costo*, *distancia* u otras propiedades.

Los algoritmos para *descubrir* la topología (parcial) de la red se basan en la construcción de grafos y pueden clasificarse en dos grandes grupos que se describen a continuación.

## Vector de distancias

La idea detrás de la estrategia conocida como *distance-vector algorithm* se basa en que cada nodo conoce las distancias (initialmente en 1) con sus vecinos conectados directamente y con costo infinito a los demás nodos.

Periódicamente, cada nodo intercambia mensajes con sus vecinos, intercambiando sus tablas y recalculando los caminos seleccionando los mas cortos a cada destino usando el algoritmo Bellman-Ford.

Cada nodo mantiene una tabla (vector) de costos para alcanzar a los demás.

Cuando un nodo  $n_i$  recibe el vector de costos de un vecino, actualiza su vector  $v[j] = \min(c + 1, v[j])$ , para cada destino  $j$ .

Un nodo también puede enviar el mensaje a sus vecinos ante un evento de detección de falla en la comunicación por alguno de sus enlaces estableciendo el costo de llegar a su vecino es *infinito*.

Un problema de este algoritmo es que ante la caída de un nodo o un enlace, puede darse el caso que otros nodos no actualizarán esta información ya que tienen un camino de menor costo.

Supongamos un conjunto de nodos conectados de la forma  .

Luego de algunas rondas de intercambios de tablas, el nodo **B** alcanza a **A** con costo 1, **C** con costo 2 y **D** con costo 3. Aquí usamos como medida de costo el *número de saltos*.

Ahora supongamos que **A** falla. **B** no recibe actualizaciones de **A** en un lapso de tiempo preestablecido y actualiza el costo de alcanzar **A** en  $\infty$ . **B** recibe una actualización de **C** . El costo de **C** a **A** es  $2 < \infty$ , por lo que **B** actualiza su costo para alcanzar **A** en  $2+1=3$ . **B** no sabe que el costo que recibió de **C** lo incluye en el camino.

Luego cuando **B** notifica a **C** , como **C** sabe que el costo de alcanzar a **A** se computó desde **B** , decide actualizar su costo a **A** en  $3+1$ . Este proceso se repetirá y los costos crecerán, por lo que se conoce como *contando hasta el infinito*.

Este problema puede solucionarse limitando el valor *infinito* a un valor (pequeño)  $n$ , lo que representaría el *número máximo de saltos*. La otra estrategia es enviar las rutas calculadas a un vecino cuyos resultados no se han derivado de las rutas recibidas de él. Esta técnica se conoce como *horizonte dividido*.

## Routing Information Protocol (RIP)

RIP se ha sido ampliamente usado en Internet. Los routers intercambian información típicamente cada 30 segundos. Un paquete RIP incluye una tabla de triples  $<address, mask, distance>$  y es prácticamente una implementación directa del algoritmo descripto arriba.

## Link state

La principal diferencia de esta idea con la anterior es que la información de cada nodo se envía a todos los demás. La técnica de *broadcast* utilizada es la de *inundación (flooding)* de paquetes. Cada nodo actualiza su tabla de ruteo con los caminos más cortos a partir de la información recibida de los demás.

La información se propaga ya que de cada nodo reenvía la tabla recibida de los demás vecinos, excepto del que la recibió (*flooding*). Para que este proceso sea confiable, es necesario que los mensajes incluyan:

- El *id* del nodo que creó el paquete.
- La lista de sus vecinos (enlaces con sus costos) directos.
- Un número de secuencia
- Un tiempo de vida (número máximo de saltos)

Los dos primeros items permiten hacer el cálculo de las rutas mínimas. El tercero permite *filtrar* paquetes que son *copias* de otros recibidos por otros vecinos. El tercero permite identificar paquetes con *nueva* información y descartar mensajes *viejos*. El *tiempo de vida* permite *eliminar* mensajes que pudieran aún estar circulando por la red y limitar el *alcance* de la red conocida por cada nodo, manteniendo las tablas con tamaños manejables.

## Open Shortest Path First Protocol (OSPF)

Es uno de los protocolos más usados en Internet y se basa principalmente en el algoritmo de *link-state*.

Provee algunas características adicionales como:

- *Autenticación de mensajes*: Basado en diferentes técnicas criptográficas.
- *Particionado jerárquico* de la red: Permite la definición de *areas* (dominios).

- *Balance de carga:* Permite *distribuir* tráfico a destinos alcanzables por diferentes rutas (con el mismo costo).

## Métricas utilizadas

La noción de *costo* depende del tipo de servicio requerido. Inicialmente, en ARPANET, una métrica utilizada fue la longitud de la cola de paquetes a ser enviados en cada enlace. Posteriormente se introdujo como métrica el *delay*, teniendo en cuenta la *latencia* y el *ancho de banda* del enlace.

$$\text{Delay} = (\text{ArrivalTime} - \text{DepartTime}) + \text{TransmissionTime} + \text{Latency}$$

Los tiempos *ArrivalTime* y *DepartTime* son los tiempos de arribo y retransmisión en el router, respectivamente. Los tiempos de transmisión y latencia se extraen desde las propiedades del enlace utilizado.

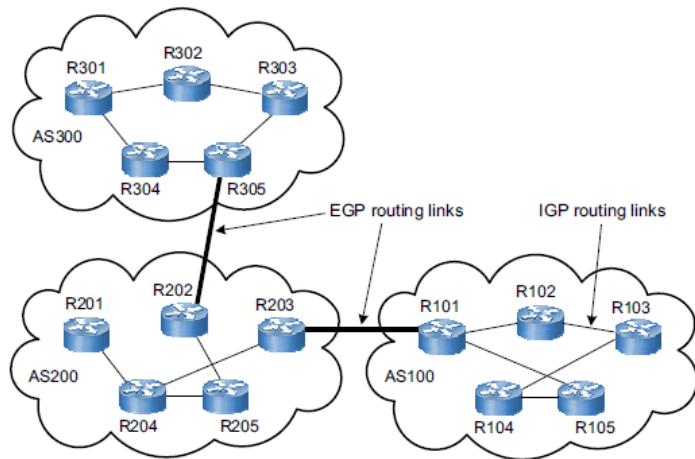
El problema con ésta métrica es que no tiene en cuenta la *carga* o utilización del enlace. Las métricas utilizadas actualmente incluyen la utilización, tasa o capacidad de transmisión y tipo de enlace. Para mayores detalles, ver [Metrics](#).

## Ruteo interdominios

Para lograr escalabilidad en Internet, cada organismo (compañía, instituciones educativas, gobierno, ISPs, etc) constituye un *dominio* o área y configura su red (WAN) como un *sistema autónomo (AS)*, en el cual puede utilizar tecnologías y sus propios protocolos de ruteo, como RIP u OSPF. Dentro de un AS el ruteo se denomina *intradominio*.

Los ASs se conectan a un *backbone* regional, nacional o internacional. Esto deriva en un sistema jerárquico de ruteo. Cada AS se conecta a otros ASs mediante *external border gateways* utilizando el [Border Gateway Protocol \(BGP\)](#).

*BGP* califica como un protocolo de ruteo de *path vector* ya que los routers intercambian mensajes de rutas (*advertisements*) y toman decisiones de ruteo en base a *paths*, es decir secuencias de direcciones de ASs a redes de destino.



**Figura 9.1: Ejemplo de ASs usando BGP.**

BGP soporta métricas de costos en base a *políticas* definidas por cada AS, como por ejemplo, factores económicos, tipo de tráfico o servicios, contratos con otros ASs, etc.

BGP utiliza TCP y la versión actual se describe en la [RFC 4271](#).

Actualmente muchos ASs corren BGP también para el ruteo interno. Para esto se utiliza una versión de BGP conocida como *interior BGP (iBGP)*, el cual selecciona el subconjunto de las *mejores rutas* a los *routers exteriores*, reduciendo el número de rutas almacenadas.

A modo de ejemplo, la [Red Interuniversitaria Nacional \(RIU\)](#) es responsable del dominio [edu.ar](#) y consiste de una red de ASs (Universidades y otros organismos) que utilizan BGP. Es posible hacer consultas BGP en su servicio [looking.glass](#).

## Ruteo multicast

Los mecanismos de ruteo analizados hasta ahora se denominan *unicast* ya que los paquetes tienen un único destino.

Un paquete *multicast* tiene como destino a un grupo de hosts. Un paquete *broadcast* tiene destino a todos los nodos de la red y se utiliza principalmente en redes locales.

Algunos protocolos a nivel de enlace, como Ethernet, tienen soporte para *multicast* en hardware. Los protocolos a nivel de red generalmente también deben brindar el servicio de *multicast* ya que muchas aplicaciones lo requieren como por ejemplo las de video-conferencia.

Algunas aplicaciones usan el modelo *one-to-many*, como por ejemplo, IPTV, radio por internet o video-broadcasting, como por ejemplo *youtube streaming*.

Otras aplicaciones, como las de video-conferencia, se basan en el modelo *many-to-many*.

Para implementar envío *multicast* se utilizan direcciones reservadas para este propósito.

En redes locales de múltiple acceso, como Ethernet, su implementación es simple. La interface *se configura* para aceptar una o más direcciones de diferentes *grupos multicast* y el hardware *acepta* los paquetes contenido esas direcciones de destino.

En redes IP es necesario implementar algoritmos en software. En el protocolo IPv4 se utilizan direcciones *multicast* de clase D, en el rango 224.0.0.0 a 239.255.255.255. IPv6 también reserva un amplio rango de direcciones *multicast*.

Una dirección *multicast* define un *grupo* de nodos. Cada nodo participante del grupo envía un paquete IGMP con un mensaje *join* a su router inmediato. Los protocolos [Internet Group Management Protocol \(IGMP\)](#) y [Multicast Listener Discovery](#), se usan para el intercambio de mensajes entre *hosts* y *routers* para establecer *grupos multicast* en IPv4 y IPv6, respectivamente.

La idea del ruteo multicast se basa en la construcción de *árboles de routers*, los cuales definen sus propias *tablas de ruteo multicast*.

Analizaremos uno de los protocolos mas usados, el [Protocol Independent Multicast \(PIM\)](#).

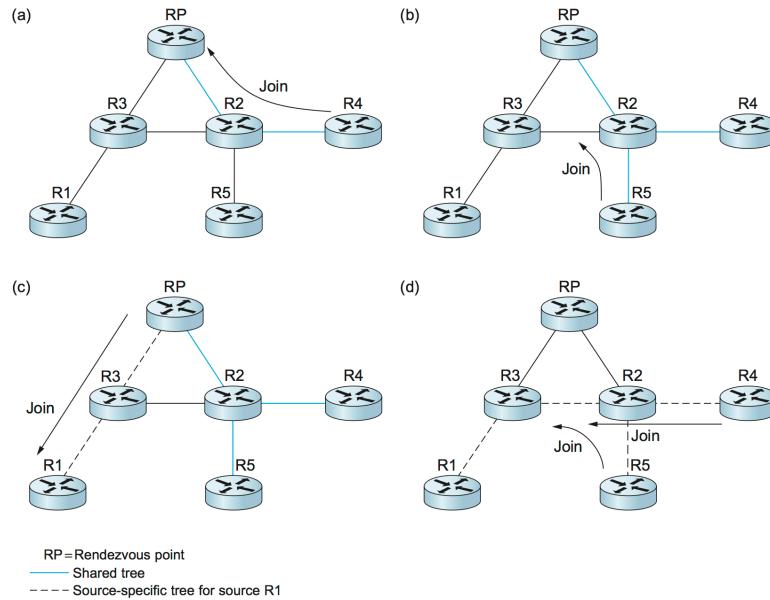
## PIM

PIM tiene varias versiones. La versión PIM-SM (*sparse mode*) es uno de los más usados. PIM-SM asigna a cada *grupo* un router que asume el rol de *rendezvous point (RP)*. Un RP actúa como la raíz del árbol de routers para ese grupo particular. El RP puede configurarse estáticamente o usando el protocolo *BootP*.

El árbol se va formando cuando un host envía un mensaje de *join G* a su router inmediato y éste lo reenvía al *RP*, como se muestra en la [figura 9.2 a\)](#).

Este mensaje crea una entrada  $(*, G)$  en la *tabla multicast* en cada router intermedio, lo cual significa que recibe y debe reenviar paquetes del grupo *G*.

Cada router que participó del ruteo del mensaje *join* queda unido al árbol del grupo, como lo muestran los arcos celestes en la figura 9.2 a) y 9.2 b).



**Figura 9.2: Funcionamiento de PIM.**

Supongamos que un host detrás del router *R1* envía un paquete a ese *grupo multicast*. El router *R1* *encapsula (tunnels)* el paquete original en un paquete *PIM Register*, con destino (*unicast*) al *RP*.

Cuando el *RP* recibe el paquete, lo *desencapsula* y al ver que es un paquete *multicast* para el grupo, lo reenvía a los routers correspondientes, (*R2*, en este caso) los cuales a su vez reenvían a los hosts del grupo.

Para evitar sucesivos encapsulamientos (lo cual afecta al rendimiento), el *RP* envía un mensaje *join* al router *emisor* (*R1*, en nuestro ejemplo). Este mensaje crea en cada router por donde pasó una entrada o *sender state* del tipo  $(S, G)$  el cual representa una ruta desde el *sender* al *RP*.

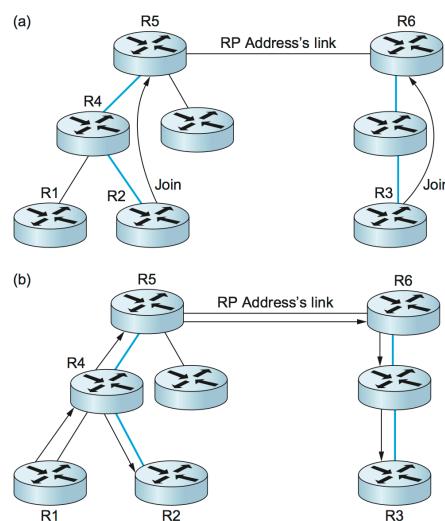
A partir de aquí, los demás routers del grupo pueden advertir al *sender router* un paquete *join*  $(S, G)$  indicando una posible ruta más corta que la ruta a través del *RP*.

Con estas nuevas entradas en las tablas multicast, en el ejemplo, *R1* y *R3* podrán reenviar próximos paquetes directamente a los demás routers del grupo vía *R2*.

El [Multicast Source Discovery Protocol \(MSDP\)](#), descripto en la [RFC 3618](#), utiliza múltiples instancias de PIM-SM para diferentes dominios, conectando los *RP* de cada dominio entre sí.

La versión *PIM-SSM* (*source specific multicast*) define un modelo *one-to-many* introduciendo el concepto de *channel*. Un *host* (no un router) envía el par (*Sender, Group*) en un mensaje *IGMP Report Message* a su router local, el cual lo reenvía en un mensaje *join* al *RP*. En este esquema sólo el *sender* puede emitir paquetes al grupo.

La versión *BIDIR-PIM* permite armar árboles bidireccionales, proveyendo el modelo *many-to-many*. Un router recibiendo un paquete de un host interno puede reenviar hacia los routers superiores (como en PIM-SM) y por las otras ramas de bajada en el árbol. Así, no es necesario el uso de *RPs*, sólo se requieren *direcciones multicast* representando a los grupos. Un mensaje *join G* enviado por un host sólo debe alcanzar al primer router que contenga una ruta al grupo con dirección multicast *G*.



**Figura 9.3: PIM bidireccional.**

En la [figura 9.3](#) a) se muestran los mensajes *join* enviados por *R2* y *R3* hasta sus correspondientes routers con el grupo *G* (routers a los que se les asignó la dirección multicast *G*), los routers *R5* y *R6*, respectivamente. Estos últimos routers actúan como las *raíces* de los múltiples árboles interconectados entre sí, tomando el rol de *RPs*.

La [figura 9.3](#) b) muestra cuando un host detrás de *R1* envía un paquete multicast al grupo *G* y cómo se reenvían (flechas) a los demás routers que participan en el grupo.

DIDIM-PIM incluye también en el árbol a los *senders*, así éstos pueden ser receptores de otros.

BIDIR-PIM es utilizable dentro de un dominio ya que los routers *RPs* deben configurarse apropiadamente, asignándoles las direcciones multicast asociadas a cada grupo.

## Ruteo con dispositivos móviles

Actualmente es común que los dispositivos sean móviles, como laptops o teléfonos con enlaces inalámbricos como wifi o 3/4/5G.

Un enfoque simple es manejar las redes con un protocolo del tipo *join/leave*, es decir basado en operaciones de inicios/fin de sesión o conexión en cada red. En este modelo, cada dispositivo al quedar fuera del alcance de una red el router (su default gateway) correspondiente lo desconecta. Al entrar en el alcance en otra red el dispositivo se *une* a la nueva red y es re-configurado para ser un nodo de la nueva red.

En este enfoque un dispositivo móvil va cambiando su dirección de red y ante cada movimiento desde una red a otra se produce una nueva asociación en la cual se deben proveer las credenciales de acceso a la nueva red.

Cabe aclarar que en este escenario el dispositivo se reconfigura en cada nueva asociación, obteniendo posiblemente una nueva dirección de red, default gateway, DNS y otros.

En algunas aplicaciones, como telefonía celular, el identificador de cada dispositivo debe preservar independientemente de su localización.

Una estrategia comúnmente usada, por ejemplo en la red de telefonía móvil, es que cada dispositivo (host) está asociado a su *home location* (localidad, dado en el prefijo del número). Cada dispositivo está asociado a su *home host agent*: Una especie de gateway fijo en la *home network* asociado al dispositivo.

El dispositivo debe enviar su localización (en qué red está) al *host agent* para que éste pueda reenviarle los paquetes cuando los reciba desde otro dispositivo y re establecer conexiones.

&lt; Anterior

## Internet Protocol

Próximo &gt;

## Protocolos de transporte

# Protocolos de transporte

Los protocolos de nivel de red tienen como objetivo la entrega de paquetes entre hosts. Las aplicaciones requieren establecer la comunicación entre *procesos* ejecutándose en diferentes hosts. Esto es realizado por los protocolos de la capa de transporte, también conocidos como *end-to-end protocols*.

Un protocolo de transporte puede ofrecer servicios orientados a datagramas u orientados a conexión.

En la familia TCP/IP los dos protocolos fundamentales son [User Datagram Protocol \(UDP\)](#) y [Transport Control Protocol \(TCP\)](#).

Ambos protocolos deben actuar como *multiplexores* de paquetes transmitidos desde procesos hacia la capa de red y viceversa.

Los paquetes de los protocolos de red incluyen un campo que identifica su payload, como el campo **protocol** en datagramas IP. Típicamente contiene el identificador del tipo de paquete (o protocolo) de la capa de transporte.

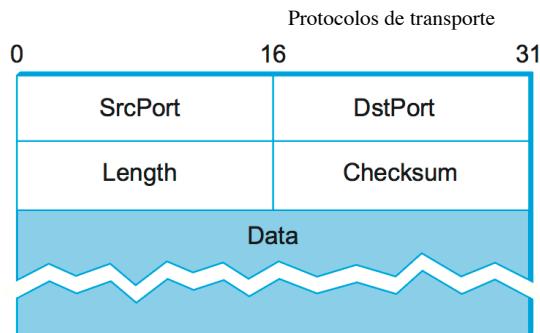
De esa forma, ante la recepción de un paquete, el protocolo de red *entrega* al módulo correspondiente de la capa superior.

Estos protocolos deben *abstraer* el *end-point* o proceso en un host. En TCP/IP, estos *end-points* se representan como *puertos (ports)*, que son valores de 16 bits.

## UDP

UDP actúa como un simple *multiplexor* de datagramas con los procesos de un host.

Un datagrama UDP relaciona proceso (port) del host (IP address) emisor con el proceso (port) del host receptor, tal como se muestra en la siguiente figura.



**Figura 10.1: Formato de un datagrama UDP.**

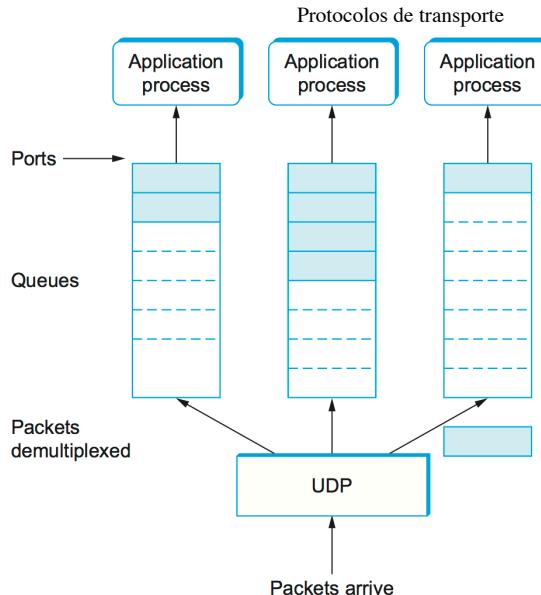
La cabecera sólo agrega los *puertos* de origen y destino, el tamaño del datagrama y un *checksum*, el cual se computa a partir del *payload*, la cabecera y del *pseudoheader* formado por las direcciones IP origen y destino, el número de protocolo y el campo *length*.

La operación de la API sockets `int bind(int s, struct sockaddr*, ...)` asocia al proceso invocante con el *end-point* `<address, port, protocol>`. En TCP/IP estos valores se corresponden con los valores de los campos `sin_family=AF_INET`, `sin_port` de la estructura `sockaddr_in`. El protocolo se selecciona desde el tipo de socket creado (ver `socket()` syscall).

Las aplicaciones del tipo cliente-servidor generalmente se basan en que los clientes envían los requerimientos a *well-known ports* los servidores. Por ejemplo, los servidores del sistema DNS usan el puerto 53. En los sistemas tipo UNIX, el archivo `/etc/services` lista los puertos reservados para las aplicaciones comunes tanto para UDP como para TCP.

Un *port mapper*, es una aplicación cliente-servidor que usa un *puerto conocido* (el 111) y los clientes le requieren el puerto asociado a un *servicio* (dado como una cadena de caracteres). El servidor *port mapper* responde con el número de puerto en el que escucha el servicio.

El módulo que implementa UDP principalmente se encarga del *demultiplexado* de datagramas, como se muestra en la siguiente figura.



**Figura 10.2: Demultiplexado de datagramas UDP.**

Cabe aclarar que una aplicación UDP usando el puerto  $p$  no tiene conflictos con una aplicación TCP asociada al mismo puerto.

Generalmente, los sistemas operativos permiten asignar un puerto con valor al 1025 a una aplicación (*bind*) si la aplicación corre con privilegios de *administrador* (*root* en sistemas tipo UNIX).

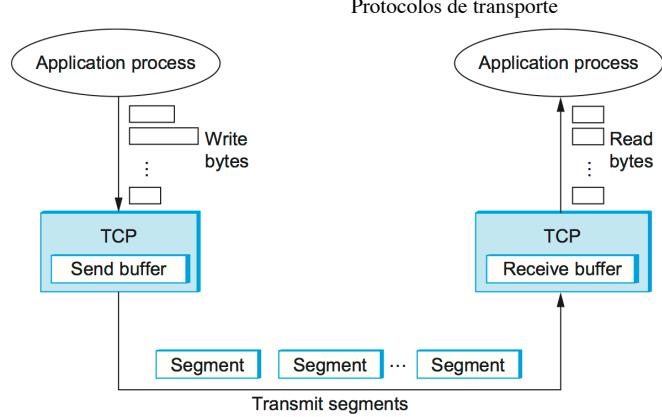
## TCP

El Transport Control Protocol provee un servicio orientado a conexión, confiable (retransmisión de paquetes faltantes o con errores), con entrega ordenada de secuencias de bytes (*byte-stream*).

También ofrece mecanismos de *control de congestión* para prevenir la sobrecarga de la red.

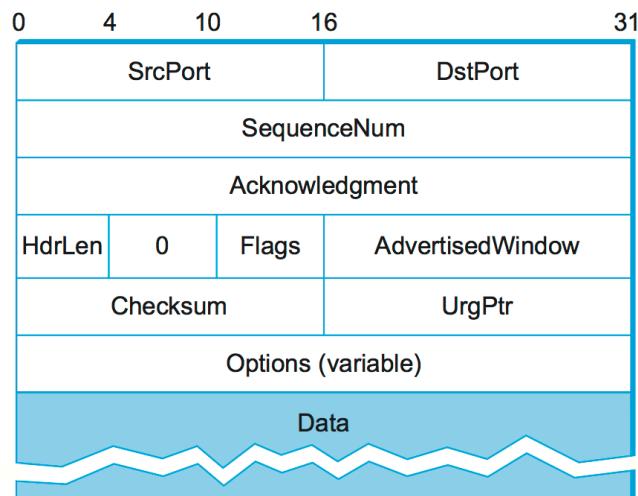
La aplicación (*peer*) debe primero establecer una *conexión* con el otro (*peer*) extremo. Una vez que la conexión se estableció, los *peers* pueden comenzar a intercambiar datos.

Un *peer* emisor envía una secuencia de bytes que son almacenados en *buffers* por el subsistema TCP, el cual arma *segmentos* a transmitir, como se muestra en la siguiente figura.



**Figura 10.3: Transmisión de segmentos TCP.**

Cada segmento tiene un encabezado que incluye los puertos de origen y destino, y un número de secuencia. Los campos *acknowledgment* y *advertised window* se usan para *reconocer* los segmentos recibidos y *notificar* al emisor sobre el tamaño del espacio libre en el buffer del receptor. Estos dos últimos campos permiten implementar el protocolo de *sliding window* (*ventana deslizante*) que *controla el flujo* entre los extremos. El *window size* indica el número de bytes que el emisor puede enviar sin esperar por un reconocimiento. Cuando TCP recibe un *Ack* con *window size=0*, *bloquea* al proceso emisor hasta recibir otro con *window size > 0*.

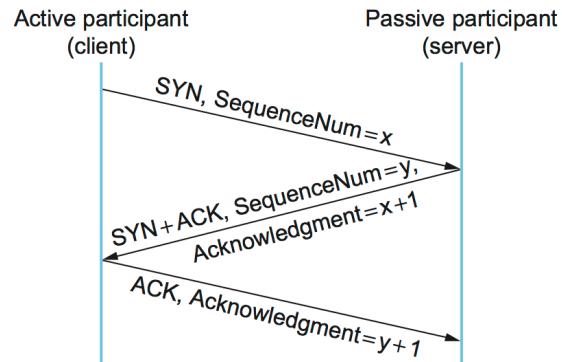


**Figura 10.4: Formato de un segmento TCP.**

El campo *HdrLen* contiene el tamaño del header en palabras de 32 bits ya que el header soporta *opciones*, lo que lo hace de tamaño variable. Los *Flags* se usan para implementar operaciones de control como el establecimiento y fin de la conexión (bits *SYN* y *FIN/RESET*). El flag *ACK* determina que el segmento contiene un reconocimiento. El flag *URG* determina que el segmento contiene *datos urgentes*, apuntados por el campo *UrgPtr*. El resto de los campos son análogos a UDP.

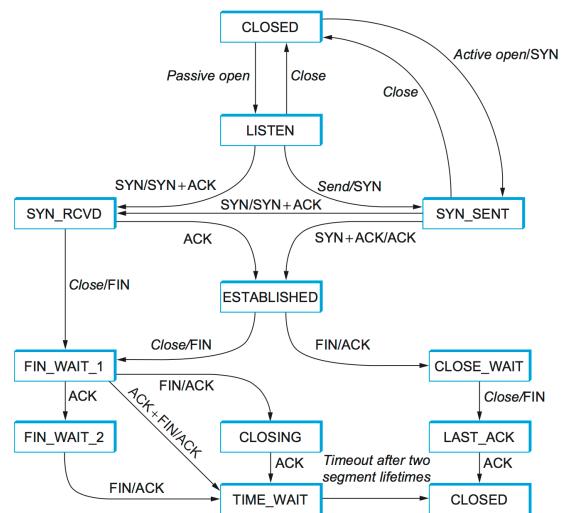
## Establecimiento de conexión

El establecimiento de conexión se describe en la siguiente figura. Se requieren tres mensajes para asegurar que ambas partes arriben al mismo estado de conexión.



*Figura 10.5: Handshake de conexión TCP.*

Luego de haberse establecido la conexión los peers pueden intercambiar datos. La siguiente figura muestra el sistema de transición de estados de TCP.



*Figura 10.6: Sistema de transición de estados TCP.*

## Transmisión y recepción

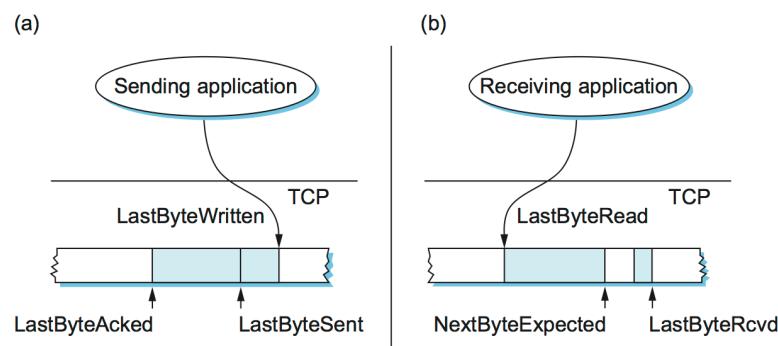
Para garantizar confiabilidad y secuencialidad de los datos transmitidos existen varios algoritmos/protocolos.

El más simple es *stop-and-wait*, el cual se basa en el uso de *reconocimientos (acks)* y *vencimientos (timeouts)*. En este protocolo, el emisor envía un paquete y *espera* por un cierto intervalo de tiempo por un *acknowlegment* del receptor para luego enviar el próximo. Si el reconocimiento no llega dentro del *timeout* establecido, el paquete se reenvía. Cada paquete incluye un número de secuencia (que puede ser 0 o 1) para relacionar cada paquete con su *ack*.

Este algoritmo, si bien es simple, está lejos de hacer una buena utilización de la capacidad del enlace.

TCP utiliza el *algoritmo de sliding window (ventana deslizante)*, el cual permite lograr una mejor utilización del canal y *control de flujo*.

El algoritmo se basa en el uso de *buffers* de emisión y recepción. El emisor en cada mensaje envía un número de secuencia `seq n` y mantiene la *ventana* de bytes transmitidos pero aún no reconocidos, tal como se muestra en la siguiente figura.



**Figura 10.7: Buffers del emisor y del receptor.**

El campo `AdvertisedWindow` (`aw`) de un segmento representa el número de bytes que un emisor puede enviar sin esperar por *acks*.

Cuando el emisor recibe un `ack n, aw`, mueve `LastByteAcked += n` y ahora puede enviar segmentos hasta `aw` bytes. Cuando envía un nuevo segmento con número de secuencia  $n$  se planifica su *timeout* (alarma). Si el *timeout* se dispara antes de recibir un *ack m* (con  $n \leq m \leq n + aw$ ), se reenvía el paquete. En otro caso, se cancela la alarma ya que el receptor recibió el segmento. Si el `aw` recibido es cero, el emisor debe *bloquear* al proceso.

El receptor también mantiene una ventana sobre su buffer de recepción. Si recibe un segmento con `seq = n + 1`, adelanta `NextByteExpected=n+1`, y eventualmente *despierta* al proceso receptor. Luego envía al emisor un `ack n+1, rws`, donde el valor de `rws` será el número de bytes libres en el buffer de recepción. Si recibe un segmento *fueras de orden*, lo almacena en el buffer en la posición correspondiente sin responder con un `ack`.

Este algoritmo establece un mecanismo de *control de flujo* gobernado por el receptor lo que previene que el emisor sobrecargue (*overflows*) el buffer de recepción.

Un factor a considerar es el valor del *timer* de cada segmento en el emisor. Este debería tener en cuenta el tiempo de ida y vuelta o *round-trip time (RTT)* de mensajes entre el emisor y el receptor.

Un *timer < RTT* generará retransmisiones aunque los segmentos hayan arribado efectivamente al receptor pero el *ack* aún estaba en tránsito. Por el contrario, un valor alto, causará demoras excesivas antes de retransmitir paquetes perdidos. TCP incluye algoritmos para determinar el *RTT* midiendo las diferencias entre los tiempos de envío de segmentos y de recepción de *acks* correspondientes, comúnmente teniendo en cuenta su *varianza*.

TCP usa un *algoritmo adaptativo* que permite encontrar valores adecuados del tamaño máximo de segmentos a transmitir usando una *ventana de congestión*. Esto hace que TCP incluya una estrategia de *control de congestión* que se analiza en el próximo capítulo.

## Protocolos *request/reply*

Estos tipos de protocolos de transporte han ganado mucha atención principalmente para implementación de *Remote Procedure Calls (RPC)*. La idea de *RPC* se basa en que una aplicación *invoca* a una operación a ejecutarse en un host remoto al estilo de invocaciones de una función, pasando valores como argumentos y recibiendo un resultado. El objetivo de *RPC* es proveer una abstracción al programador que haga transparente la ejecución (local o remota) de una función ocultando los detalles de las comunicaciones. Esto simplifica el diseño e implementación de aplicaciones distribuidas.

Un ejemplo de un protocolo *reply/request* es *HyperText Transport Protocol (HTTP)*, usado en la *world wide web*, comúnmente implementado sobre TCP.

Los análisis realizados indican que TCP puede que no sea el ideal para implementar estos protocolos, por lo que recientemente se han propuesto otros, específicos para este modelo, entre los que podemos mencionar [Stream Control Transmission Protocol \(SCTP\)](#) y [QUIC](#).

QUIC fue desarrollado por Google y aceptado para su estandarización por la [IETF](#). Actualmente ya está soportado en algunos *browsers* como Google Chrome y Mozilla Firefox. Básicamente tiene las siguientes características:

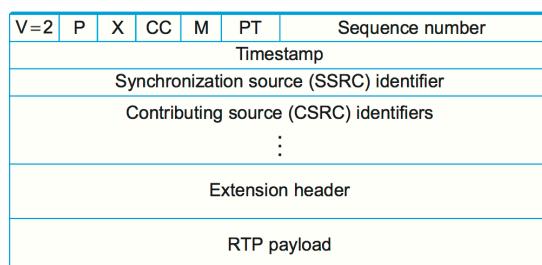
1. Simplifica el *handshake* de conexión incorporando los parámetros para establecer conexiones seguras.
2. Soporta *reconexiones* rápidas en dispositivos con múltiples interfaces de red como teléfonos celulares (wifi + 3/4G).
3. Se basa en UDP, o sea que es un protocolo de transporte sobre otro en la misma capa de abstracción. Esta técnica se conoce como *tunneling*.

## Transporte en tiempo real

Las aplicaciones multimedia, como VoIP (voz sobre IP), streaming de audio y video requieren protocolos de transporte que provean servicios como secuencialidad, *timing*, sincronización entre flujos (ejemplo, audio, video y subtítulos) y no necesariamente deben garantizar confiabilidad. Por ejemplo, no tiene sentido retransmitir un paquete de VoIP perdido fuera de secuencia, ya que el receptor podría escuchar *ecos* del pasado.

El [Real-time Transport Protocol \(RTP\)](#) fue diseñado para cubrir esta necesidad. Generalmente se usa con el *Real-time Control Transport Protocol (RCTP)* que permite intercambiar información de control por un canal alternativo en una sesión o flujo RTP.

Generalmente se utiliza RTP sobre UDP.



**Figura 10.8: Formato de un frame RTP.**

La cabecera de un frame RTP permite soportar secuencialidad e incluye un *timestamp* para el *timing* de la *reproducción* y sincronización entre flujos. Un receptor generalmente calcula la diferencia entre *timestamps* para determinar los tiempos de reproducción de los frames recibidos.

El campo *synchronization source identifier* permite identificar un *flujo* de datos, por ejemplo, un flujo de audio mp3. Esto permite implementar una *pseudo conexión* o flujo de paquetes relacionados.

Un flujo puede ser la transmisión de un *contenedor multimedia*, como MPEG, OGG u otros (AVI, QuickTime, ...).

Los *contributing source identifiers* se usan cuando diferentes flujos se *mezclan* o multiplexan en un único flujo, por ejemplo la voz en diferentes idiomas en un video. El mixer identifica cada origen para que el decodificador pueda demultiplexarlos. El número de *contributing sources* está dado en el campo **CC** de 4 bits.

El protocolo RCTP permite asociar a cada sesión funciones de control como

1. Información (feedback) sobre el rendimiento de la aplicación. Esto permite intercambiar parámetros como por ejemplo, resolución de video y frames por segundo dependiendo del rendimiento de la red.
2. Relacionar y sincronizar diferentes flujos del mismo emisor como por ejemplo audio y video, ya que valores de RTP como el *timestamp* de cada frame de diferentes flujos pueden tener diferentes escalas (*clocks*) y es necesario correlacionarlos.
3. Identificación (metadatos) de emisores que las aplicaciones pueden mostrar.

Generalmente, las aplicaciones usan el *port n* para RTP y el *n+1* para RCTP.

Para el acceso a un flujo generalmente se usa el *Real Time Streaming Protocol (RTSP)* que permite que una aplicación se conecte a un servidor RTSP (por TCP), usando un URL de la forma **rtsp://domain/resource**, el cual retorna una descripción del flujo en formato Session Description Protocol (SDP), luego el cliente requiere el flujo desde el servidor usando RTP.

RTSP permite el control del flujo con mensajes **PLAY** , **PAUSE** , **RECORD** (upload) y otros.

Estos protocolos son comunes en aplicaciones de streaming, cámaras IP y circuitos cerrados de TV (CCTV).

Otro protocolo alternativo a RTSP es el [\*\*Real Time Messaging Protocol \(RTMP\)\*\*](#), creado por Adobe para la transmisión de video en Flash y liberado en 2012.

RTMP usa TCP y permite crear *canales virtuales* independientes, por ejemplo, para audio, video y mensajes (chat). Existen varias extensiones para proveer seguridad y transmisión sobre túneles HTTP (RTMPT).

Actualmente es posible ver que Youtube usa RTMP para *live streaming* ofreciendo un URL como `rtmp://a.rtmp.youtube.com/live2` para recibir una presentación en vivo usando un *encoder* como [\*\*Open Broadcaster Software \(OBS\)\*\*](#) u otros.

Otro protocolo para transmisión en vivo es [\*\*HTTP Live Streaming \(HLS\)\*\*](#), desarrollado por Apple y liberado en 2009. Aprovecha las facilidades de reproducción de sonido y video de navegadores HTML5. Codifica video en formato [\*\*H.264\*\*](#), audio en formatos [\*\*AAC\*\*](#), [\*\*MP3\*\*](#) y otros. Usa [\*\*MPEG-2\*\*](#) o [\*\*MPEG-4\*\*](#) como *contenedores multimedia*.

El protocolo [\*\*Dynamic Adaptive Streaming over HTTP \(DASH\)\*\*](#), es ampliamente usado actualmente (por Youtube, Netflix y otros).

La ventaja de los protocolos sobre HTTP es que están soportados por los *browsers* modernos y se usan como clientes en aplicaciones web de streaming o video-conferencias.

En [\*\*Computer Networks: A System Approach. RTP\*\*](#) se describen en mayor detalle estos protocolos.

---

< Anterior

Ruteo

Próximo >

APIs

# Interfaces de programación

Un sistema operativo (SO) provee llamadas al sistema como operaciones básicas (primitivas) de comunicación.

Es común que un SO provea mecanismos de comunicación entre procesos locales, conocidos como *Interprocess communication (IPC)* como pipes, FIFOs, memoria compartida y colas de mensajes.

El subsistema de red provee llamadas al sistema para comunicación remota generalmente con soporte de diferentes familias de protocolos.

## Berkeley y POSIX sockets

La API de BSD sockets soporta *Unix* e *Internet Domain Sockets*. Los primeros permiten comunicación entre procesos locales, mientras que los segundos permiten comunicación de sistemas y procesos remotos usando la familia de protocolos *TCP/IP*.

Un *socket* es una abstracción de un *extremo local (endpoint)* de comunicación. Se representa como un *descriptor de archivo* por lo que soporta operaciones como `read(s, buf, count)`, `write(s, buf, count)` y `close(s)`.

La API de BSD sockets fue tomada por el estándar [POSIX](#) con algunas leves modificaciones.

La API se describe en C aunque cada lenguaje de programación define *bindings* a las operaciones de bajo nivel ofrecida por el SO.

La API ofrece varios archivos de cabecera como los que se listan en la siguiente tabla.

Archivo	Descripción
<code>sys/socket.h</code>	Funciones y estructuras de datos
<code>netinet/in.h</code>	Protocolos y direcciones IP y puertos UDP/TCP

Archivo	Descripción
arpa/inet.h	Funciones de manipulación de direcciones IP

Las funciones pueden estar implementadas directamente como llamadas al sistema o como funciones de biblioteca, dependiendo del SO.

Función	Descripción
socket()	Retorna un socket de un tipo dado
bind()	Asocia un socket a una dirección (IP/port)
listen()	El socket entra en un estado <i>listen</i>
connect()	Conexión a un <i>servidor</i> . Usado por un <i>cliente TCP</i>
accept()	Acepta una conexión. Usado por un <i>server TCP</i>
send()/write()	Envía datos por el socket en <i>connected state</i>
sendto()	Envía datos por el socket a una dirección dada (UDP)
recv() / read()	Recibe datos por el socket (conectado)
close()	Cierra el socket. En TCP termina la conexión
gethostbyname()	Obtiene la IPv4 dado el <i>nombre</i> o <i>dominio</i> de un host
gethostbyaddr()	Obtiene la IPv4 dada la dirección IP como un string
getaddrinfo()	Obtiene una lista de direcciones IPv4/v6 desde un nombre/dirección
select()	Espera por actividad en un arreglo de descriptores
poll()	Verifica el estado de descriptores en un arreglo

La función `int socket(domain, type, protocol)` crea un socket de un dominio, familia y protocolo específico dado.

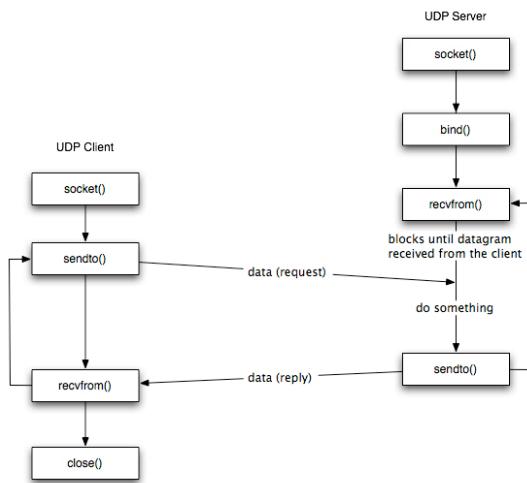
Los dominios (entre otros) pueden ser `AF_UNIX` , `AF_INET` o `AF_INET6` , para UNIX, IPv4 e IPv6 respectivamente.

El parámetro *type* identifica el tipo de servicio requerido como `SOCK_STREAM` , `SOCK_DGRAM` , `SOCK_SEQPACKET` o `SOCK_RAW` que representan servicios de transmisión de *streams* (como TCP), *datagramas* (IP/UDP), paquetes con garantía de secuencialidad o un servicio *plano* (IP, Ethernet, ...).

El último argumento es el protocolo específico como `TCP` o `UDP` . Si se omite se determina un protocolo *por omisión* (ej: `TCP` para `SOCK_STREAM` ).

## API con protocolos orientados a datagramas

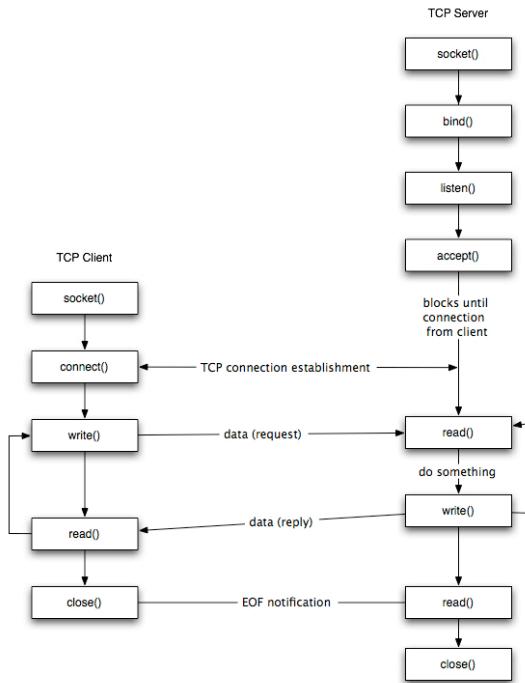
En un protocolo sin conexión, como UDP, el patrón de uso de la API de sockets se muestra en la siguiente figura.



Una de las partes (ej: el servidor) deberá estar esperando un mensaje (comúnmente bloqueado) ejecutando un `recvfrom()`. Otro proceso enviará un mensaje usando `sendto()`. La función `sendto(s, buf, len, dst_addr, dst_addr_len)` incluye la dirección de destino. La función `recvfrom()` recibe en un parámetro de salida la dirección del emisor del mensaje, comúnmente la IP/port.

## API con protocolos orientados a conexión

En un protocolo orientado a conexión, como TCP, el patrón de uso de la API de sockets se muestra en la siguiente figura.



En éste escenario el cliente deberá conectarse al servidor (usando `connect()`/`accept()`) para obtener un *socket conectado*. Luego, ambas partes pueden comunicarse usando `send()`/`write()` y `recv()`/`read()`.

Cabe hacer notar que es posible usar las llamadas al sistema `read(s, buf, count)` y `write(s, buf, count)` de entrada/salida con archivos.

Esto permite reusar componentes (funciones/bibliotecas) que leen/escriben desde archivos pasando sockets como descriptores. El SO distingue que al operar sobre sockets se debe redirigir los datos al subsistema (stack) de red.

## Uso de notificaciones de entrada-salida

Las llamadas al sistema `select()` (UNIX) y `poll()` (BSD) permiten al proceso invocante bloquearse hasta que haya algún evento de cambio de estado en descriptores de entrada-salida como archivos y sockets.

La función `select(inputs, outputs, exceptions, timeout)` toma tres conjuntos (bitsets) de descriptores de archivos, pipes o sockets de entrada, salida y de excepciones (descriptores en los que se quieren manejar los errores). Opcionalmente se puede pasar un timeout que la desbloquea.

Cada elemento del bitset se puede manipular mediante el uso de las macros `FD_ZERO(&bitset)`, `FD_SET(fd, &bitset)`, donde `fd` es el valor del descriptor de archivo.

El uso de estas funciones permiten implementar aplicaciones que requieren de múltiples conexiones sin necesidad de implementar un servidor concurrente (usando threads o procesos hijos). Esto permite el desarrollo de una arquitectura típica de un *sistema reactivo*.

La operación `select()` se desbloquea al ocurrir un evento de entrada/salida en alguno de los descriptores.

Al retornar de `select()`, la macro `FD_ISSET(fd, &bitset)` determina si el descriptor `fd` tiene un evento en el set. Si ocurrió un evento en un descriptor en *inputs* significa que hay datos disponibles para leer. Si ocurrió un evento en un descriptor en *outputs* significa que hay espacio en el buffer interno y se puede escribir en él. Si ocurrió un evento en un descriptor de *exceptions* indica un error o excepción como un cierre de conexión.

En lenguajes de alto nivel como Python, los descriptores se pasan en listas y la función `select()` retorna tres listas con los descriptores que han recibido algún evento de entrada-salida.

## Arquitecturas de servidores

En los diagramas de las secciones anteriores las aplicaciones del servidor son *iterativas*, es decir que pueden atender a un requerimiento de un cliente a la vez.

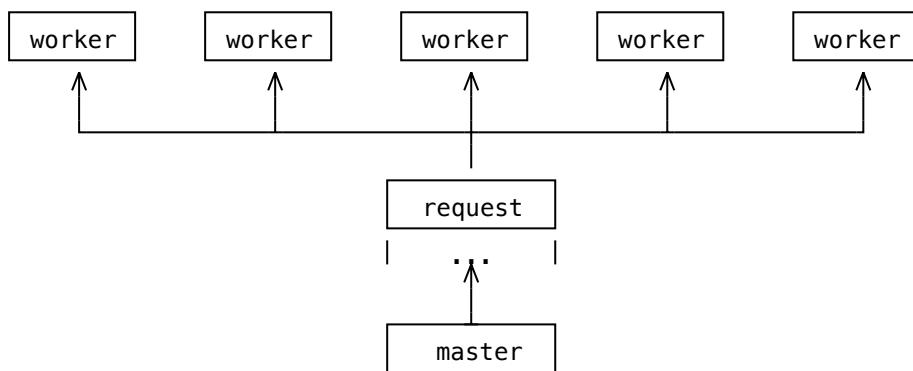
Para permitir interactuar con múltiples clientes en forma simultánea, es común que diseñar e implementar *servidores concurrentes*. Es típico en el caso de protocolos orientados a conexión, que el proceso (o thread) principal, sólo acepte conexiones y luego delegue la atención a cada cliente en procesos o threads *hijos* como se muestra en el siguiente extracto de código.

```
clike
// snip
while (true) {
    conn_socket = accept(s, ...);
    spawn(conn_socket); // create new child process or thread
}
// snip
```

Esta estrategia tiene el problema que un servidor muy sobrecargado de conexiones dispara muchos procesos o threads lo cual genera mucha sobrecarga en el sistema, además de que los tiempos de respuesta pueden ser largos por el tiempo de creación de threads/procesos.

## Threadpool

Una mejora es crear un número predefinido de *workers* (procesos o threads) y el thread/proceso principal (*master*) encola cada requerimiento para que sea procesado por algún *worker* desocupado (*idle*), como se muestra en el siguiente diagrama.



La cola de requerimientos es un recurso compartido por lo cual se debe sincronizar el acceso. Una estrategia de scheduling con balance de carga semi-automática es *job stealing* donde cada *worker idle* intenta *tomar* el primer *job* de la cola de requerimientos.

## Entrada/salida asincrónica

En la arquitectura de software basada en entrada/salida (*E/S*) asincrónica más básica el programa define un patrón de *concurrency* sin usar paralelismo o threads. En ciertos casos en sistemas distribuidos puede lograr un mejor rendimiento sin usar threads.

A bajo nivel se pueden usar las APIs basadas en `select()`, `pool()`, `epool()` típicas de los sistemas UNIX. Otros lenguajes proveen mecanismos similares o abstracciones como *promises* o *futures* basados en el patrón `async/await`.

Este paradigma provee un mecanismo similar a las *corrutinas* o *multitarea cooperativa*, ofreciendo una abstracción de alto nivel.

Intuitivamente, un *future* o *promise* puede verse como un valor que eventualmente estará disponible, es decir puede estar en un estado *ready* o *not ready*. También es posible que tenga un tercer estado de error o *failed*.

Una operación (función o bloque) *async* retorna una *promise* (o *future*). Una operación como `v = await future` bloquea al thread hasta que el valor esté disponible, retornando el control al *loop event*. La primitiva `await` es similar a `yield` usado en *generators* aplicables a *iteradores*.

El mecanismo requiere de un *executor* o *event loop* encargado de invocar el progreso de las tareas asincrónicas.

Es posible encontrar este patrón en los lenguajes de programación modernos como Javascript, Python o Rust.

Los principales conceptos son los siguientes:

- *Event loop*: El ejecutor central de *tasks*.
- *Coroutines*: Función con un *estado*: (*program pointer*, *env*, *state*). El *environment* (o *clausura*) es un registro de activación asociado con los *bindings* a las variables y parámetros a las que tiene acceso. El estado determina si esta completada (*ready*), o *pending* (continuará después de un *await*).
- *Futures*: Objetos que representan valores eventuales.
- *Tasks*: Estado de corrotinas (encapsuladas en *futures*) por el ejecutor para invocar (o continuar con) su progreso.
- *async function or block*: Define una *coroutine*.
- *await keyword*: Retorna el control al *event loop* y actualiza el estado (*program pointer*) de la coroutine. Sólo puede usarse dentro de una función *async*.

Abajo se muestra un ejemplo simple de éste modelo en Python.

python

```
import asyncio

async def task1():
    await asyncio.sleep(1)
    print("Hello, from task 1!")

async def task2():
    print("Starting task 2...")
    await asyncio.sleep(1) # Simulates doing something else f
```

```
print("Finished task 2!")

async def main():
    # Schedule both tasks to run concurrently
    await asyncio.gather(
        task1(),
        task2(),
    )

asyncio.run(main())
```

La función `asyncio.run()` representa el *event loop*. Además provee otras funciones para permitir disparar otras coroutines como `asyncio.gather()` o `asyncio.create_task(fn)`.

---

< Anterior

## Protocolos de transporte

Próximo >

## Congestión

# Control de congestión

Una red es un conjunto de recursos requeridos en forma *competitiva* por las aplicaciones en ejecución. Cada nodo (hosts y routers) debe administrar sus recursos (CPU, memoria, conexiones TCP, etc), para poder brindar un servicio razonable a cada aplicación.

Los routers generalmente deben *encolar* temporalmente los paquetes para poderlos reenviar por algún enlace compartido. Cuando un paquete debe esperar en colas, se incrementa el tiempo de *round trip (RTT)* entre los extremos. Cuando esas colas o *buffers* desbordan se deben descartar paquetes. Cuando las colas alcanzan ciertas longitudes o están desbordados, se dice que existe *congestión*.

La congestión puede aliviarse si se asignan o reservan más recursos como ancho de banda, enlaces, buffers y otros. Otra estrategia es *clasificar* los paquetes en base al tipo de aplicación. Una aplicación de tiempo real como streaming de sonido o video debería tener mayor *prioridad* que los paquetes de una aplicación como por ejemplo, de correo electrónica u otro servicio de mensajería. Asignar políticas de prioridades a diferentes *flujos* se conoce como *Calidad de servicio (QoS)*. En su implementación generalmente se usan buffers con diferentes *disciplinas de colas*, como *weighted fair queuing (WFQ)*, las cuales se asignan a diferentes flujos.

La diferencia entre *control de flujo* y *control de congestión* es que el primero se basa en mecanismos para que el emisor no *desborde* al receptor mientras que el segundo ataca el problema de la red, teniendo en cuenta el estado de los routers y enlaces intermedios.

## Control de congestión en TCP

TCP provee mecanismos para detectar y tratar de aliviar o detectar la congestión de la red, más allá del *control de flujo* entre los extremos visto.

El objetivo es tratar de determinar la *capacidad* de la red entre los extremos y aún es tema de investigación lograr mejores mecanismos.

TCP mantiene una variable de estado, llamada **CongestionWindow** y usa

clike

$$\text{MaxWindow} = \text{MIN}(\text{CongestionWindow}, \text{AdvertisedWindow})$$

como valor máximo de bytes a transferir, donde **AdvertisedWindow** es la recibida en el último ACK. El problema de TCP es inferir el valor de **CongestionWindow**. El hecho que expire el timer de un segmento antes de recibir el ACK correspondiente es una señal de eventual congestionamiento.

La idea de los algoritmos es ir variando la cantidad de bytes a transferir tratando de no generar congestionamiento mientras tanto manteniendo la tasa de transmisión mas alta posible. Estos dos objetivos, obviamente se contraponen.

Un primer enfoque es decrementar su valor ante la ocurrencia de *timeouts*, así dejar de contribuir al congestionamiento. El problema de este enfoque es que al descongestionarse la red no incrementa el valor y se pierde eficiencia en el uso de la red.

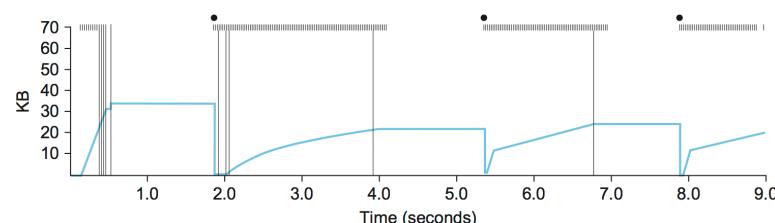
Una mejora a esta estrategia es ir incrementando el valor de **CongestionWindow** mientras reciba ACKs antes de los *timeouts*.

El mecanismo usado inicialmente en TCP se basa en un incremento exponencial, curiosamente conocido como *slow start*. Inicialmente, **CongestionWindow** toma el valor 1. Al recibir el ACK lo incrementa en 1. El emisor transmite el doble de segmentos del valor **CongestionWindow**.

Al ocurrir un *timeout* TCP reduce el valor a la mitad.

A partir de este punto, el emisor usa un crecimiento lineal, aumentando de a un paquete en cada ACK recibido hasta alcanzar el valor de **CongestionWindow** cuando finalizó el ciclo anterior.

La siguiente figura muestra el comportamiento de TCP con este algoritmo.



**Figura 11.1: Control de congestión en TCP.**

Los puntos encima del gráfico muestran los timeouts. Las barras horizontales superiores muestran el tiempo en que se transmite un paquete y las barras verticales muestran el inicio de una transmisión o retransmisión. La línea azul muestra en valor de `CongestionWindow` .

Uno de los problemas con este algoritmo es que la espera por timeouts arroja tiempos de espera con inactividad. La mejora introducida posteriormente se conoce como ***Fast Retransmit***. La idea consiste en que el receptor al recibir un segmento *fuera de orden* reenvía el último ACK. Este ACK duplicado indica al emisor que posiblemente los segmentos anteriores se hayan perdido o estén aún en tránsito. El emisor, luego de recibir tres ACKs duplicados reenvía los segmentos aún no reconocidos, adelantándose a los timeouts.

Una variante de este algoritmo, denominado CUBIC, implementado por primera vez en el kernel GNU-Linux, se enfoca en ajustar periódicamente la ventana de congestión cada vez que se recibe un ACK duplicado. Usa la siguiente función cúbica sobre el tiempo para ajustar la ventana de congestión:

$$CW(t) = C \times (t - K)^3 + W_{max}$$

donde

$$K = \sqrt[3]{W_{max} \times (1 - \beta)/C}$$

$C$  es el *factor de escala* y  $\beta = 0.7$  es el factor de decrecimiento.  $W_{max}$  es el máximo valor alcanzado por la ventana de congestión.

Este último enfoque se comporta mejor en redes con grandes y pequeños RTTs.

La curva generada por los cambios del tamaño de la ventana de congestión es menos variable y se ajusta rápidamente a los cambios.

## Otros algoritmos

El algoritmo descripto para TCP se basa en la *detección* de congestión. Los nuevos enfoques, algunos de ellos implementados en routers e implementaciones de TCP, se basan en la *prevención* de congestión (severa).

El algoritmo conocido como *DECbit* se basa en que cada router setea un bit si su cola de paquetes tiene longitud mayor que su longitud media. Este bit es incluido en el ACK. El emisor cuenta el número de paquetes enviados con el bit seteado. Si ese número es menor que el 50% de los paquetes enviados, la ventana se incrementa linealmente (en 1). En otro caso, la ventana se decrementa por 0.875 (*multiplicative decrease*).

*Random Early Detection (RED)* se basa en la misma idea que DECbit pero con notificaciones implícitas al emisor. Los routers congestionados *descartan (drop)* paquetes aleatoriamente usando alguna estrategia. Luego, el emisor recibirá ACKs duplicados u ocurrirán timeouts y el algoritmo estándar actualizará la ventana de congestión.

Otro enfoque, conocido como *basados en el origen* se basan en que el emisor intenta predecir congestión midiendo los RTTs de paquetes y si éstos se incrementan por sobre el RTT promedio, la ventana de congestión se decrementa (por ejemplo, por 1/8).

Algunos de los algoritmos basados en este último enfoque son *TCP Vegas* y *BBR* de Google. Ambos se basan en medir *delay*, lo cual permite inferir que las colas de los routers intermedios han incrementado su longitud.

---

< Anterior

APIs

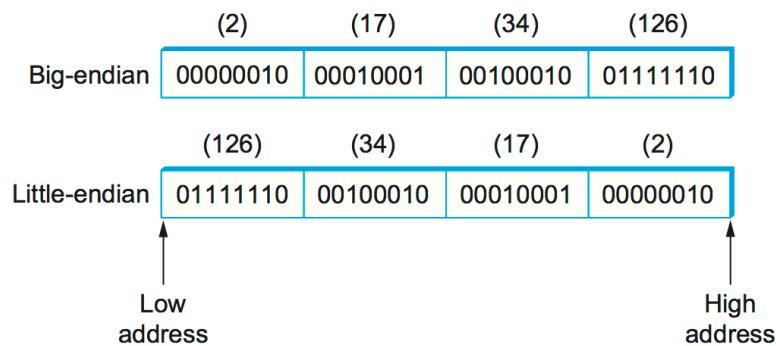
Próximo >

Presentación

# Presentación

Los protocolos analizados hasta ahora se basan en encabezados con un formato predeterminado. A diferencia de los protocolos de transporte, las aplicaciones generalmente requieren comunicarse con diferentes tipos de mensajes.

Para lograr independencia de plataformas un protocolo debe definir la *representación* de los datos. A modo de ejemplo, la API de sockets ya vista provee funciones para convertir valores numéricos como por ejemplo direcciones de red o puertos, desde y hacia el formato esperado por el protocolo, ya que la arquitectura del hardware puede representar en la memoria esos valores con los bytes en diferente orden. Por ejemplo, algunas arquitecturas almacenan los valores numéricos multi-byte en *little-endian* y otras en *big-endian*, como se muestra en la siguiente figura.



**Figura 12.1: Posibles representaciones del valor (en 32 bits)**  
**34677374.**

A diferencia de los protocolos de transporte, las aplicaciones generalmente requieren comunicarse con diferentes tipos de mensajes.

La representación de *strings* generalmente no es inconveniente porque se representan en secuencia.

Los protocolos de red, además deben tener en cuenta otras consideraciones a diferencia de los formatos usados en almacenamiento secundario. En particular, se debe tener en cuenta el orden en que un receptor procesa los datos para evitar excesivo uso de buffers para procesamiento posterior, como se verá más adelante en las aplicaciones multimedia.

# Marshalling

El término *marshalling* se refiere a la codificación y decodificación de las estructuras de datos para su almacenamiento o transmisión. Este proceso también se conoce como *serialización* ya que se debe generar una representación *plana*, es decir en una secuencia en formato binario o textual que la represente.

Para serializar los diferentes tipos de datos generalmente se usa algún esquema de *marcado (tags)* que permita al receptor decodificar y reconstruir la estructura de datos transmitida.

A modo de ejemplo, en algunos formatos cada valor se representa como una tupla de la forma `(type, length, value)`, donde `type` representa el tipo de dato, `length` determina el número de bytes del valor que se encuentra a continuación. De esta forma es simple representar los tipos básicos y tipos estructurados como arreglos y registros.

Uno de los mayores desafíos es representar estructuras enlazadas como listas, árboles o grafos arbitrarios. Una estrategia es *linearizar* con alguna estrategia sus elementos y las referencias remotas pueden representarse como índices en la secuencia. La otra, mas flexible, es agregar identificadores a cada valor y valores que representen *referencias*. Esta última técnica se usa generalmente en las notaciones como XML o JSON para identificar a *objetos remotos*.

## XML

El *Extensible Markup Language (XML)* se basa en que cada dato se *encierra* en *tags* de apertura y cierre, tal como se muestra en el siguiente ejemplo que describe un registro representando los datos de un empleado.

xml

```
<?xml version="1.0"?>
<employee>
    <name>John Doe</name>
    <title>Head Bottle Washer</title>
    <id>123456789</id>
    <hiredate>
        <day>5</day>
        <month>June</month>
        <year>1986</year>
```

```
</hiredate>  
</employee>
```

Cada aplicación deberá definir la sintaxis de tags usados y de los atributos permitidos en cada uno de ellos. Esta definición puede hacerse con *XML Schema*, el cual se describe en un *XML Schema Document (XSD)*, el cual se basa también en XML.

A continuación se muestra un posible esquema que define los tags y atributos permitidos para la definición de un empleado.

xml

```
<?xml version="1.0"?>  
<schema xmlns="http://www.w3.org/2001/XMLSchema">  
  <element name="employee">  
    <complexType>  
      <sequence>  
        <element name="name" type="string"/>  
        <element name="title" type="string"/>  
        <element name="id" type="string"/>  
        <element name="hiredate">  
          <complexType>  
            <sequence>  
              <element name="day" type="integer"/>  
              <element name="month" type="string"/>  
              <element name="year" type="integer"/>  
            </sequence>  
          </complexType>  
        </element>  
      </sequence>  
    </complexType>  
  </element>  
</schema>
```

El esquema define que el elemento (tag) **employee** es un tipo complejo (una estructura) que puede contener una secuencia de otros elementos con tags **name** , **title** , **id** y **hiredate** . Este último también es un tipo estructurado (una fecha). Se debe notar que el documento indica para cada tag el tipo de dato que puede encerrar.

Una aplicación puede usar el XSD para *validar* (typecheck) el documento XML y extraer e interpretar cada valor.

Para evitar conflictos de nombres entre esquemas XML, un XSD incluye una línea definiendo un *espacio de nombres* de la forma:

```
targetNamespace="http://www.example.com/employee"
```

De esta forma en un XSD es posible referenciar tags de otro XSD, como en el siguiente ejemplo:

```
xmlns:emp="http://www.example.com/employee"
```

Un documento XML que use varios espacios de nombres deberá desambiguarlo prefijando el espacio de nombres en el tag. Por ejemplo:

```
<emp:title>Head Bottle Washer</emp:title>
```

## JSON

En aplicaciones web se ha popularizado JSON (siglas derivadas originalmente de *Javascript Object Notation*). Se basa en la notación de valores de datos de Javascript, el cual soporta tipos básicos como números (enteros y reales), lógicos ( `True` , `False` ), caracteres y los agregados como arreglos y objetos (registros). El siguiente ejemplo muestra la notación usada para describir el empleado de la sección anterior.

```
json
{
    "name": 'John Doe',
    "title": 'Head Bottle Washer',
    "id": 123456789,
    "hiredate": {
        "day": 5,
        "month": 'June',
```

```
        "year" : 1986  
    }  
}
```

También existen implementaciones de *BSON*, el cual representa un objeto JSON en formato binario.

La popularidad de JSON en aplicaciones web se debe a que es muy simple serializar (`str=JSON.stringify(value)`) y des-serializar (`value=JSON.parse(txt)`) en Javascript usando la API JSON soportada por los browsers.

## Remote procedural call (RPC)

En los programas distribuidos es deseable ocultar los detalles de la comunicación e implementar la interacción entre los componentes del sistema usando un mecanismo familiar como el de invocación a funciones de la forma `target.f(args)`, donde `target` es el componente (remoto) destino, `f` es el servicio requerido pasándole los argumentos correspondientes.

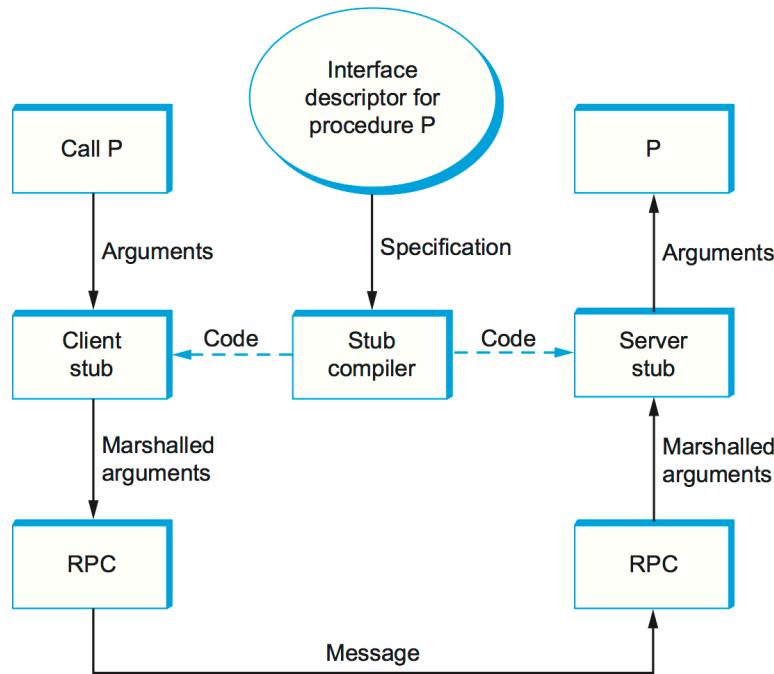
Para lograr esta abstracción en un lenguaje de programación, el sistema de ejecución basado en *RPC* transforma una invocación en un mensaje que será recibido por el destinatario, lo decodificará e invocará a la función de servicio correspondiente. Para eso el sistema debe hacer una codificación del mensaje con la serialización/de-serialización de valores de los argumentos y del resultado en la respuesta.

El pasaje de mensajes puede ser síncrona (el cliente espera por la respuesta) o asíncrona (el cliente continúa con su ejecución).

Un sistema de *RPC* independiente del lenguaje de programación, comúnmente tiene la siguiente arquitectura:

- *Interface Definition Language* o *IDL*.
- Un *IDL-compiler*: Genera los *stubs* del cliente y del servidor.
- *Client stubs*: Funciones que generan los mensajes que representan las invocaciones remotas.
- *Server stubs*: Esqueletos de funciones que el desarrollador debe implementar. Comúnmente también se genera una *dispatching table* que el servidor usará para invocar a las funciones en la recepción de mensajes.

La siguiente figura muestra un escenario de RPC.



**Figura 12.2: RPC con stubs generados a partir del IDL.**

Una de las primeras implementaciones de RPC fue desarrollada por la empresa SUN y se conoce como SUNRp. Usa una técnica de serialización en binario y soporta todos los tipos de datos de C.

Entre las implementaciones modernas de RPC es posible mencionar [SOAP](#) y [gRPC](#) de Google.

SOAP se usa principalmente en la implementación de servicios web. Se basa en XML e incluye un lenguaje de definición de interfaces, el [Web Services Description Language \(WSDL\)](#), XML schema y un mecanismo de publicación y búsqueda de servicios web [UDDI](#).

SOAP es independiente del protocolo de transporte a utilizar pero es muy común que se use sobre HTTP, permitiendo implementar aplicaciones conocidas como *web services*.

gRPC es un framework o *middleware* moderno y libre. La definición de los servicios y la serialización se basa en *Protocol Buffers*, desarrollado por Google, como se muestra en el siguiente ejemplo.

```
service PersonsDB {
    rpc addPerson (Person) returns (bool);
}
```

```
message Person {
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;

    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }

    message PhoneNumber {
        required string number = 1;
        optional PhoneType type = 2 [default = HOME];
    }

    required PhoneNumber phone = 4;
}
```

gRPC compila los *stubs* en el cliente. En el servidor una biblioteca permite decodificar los mensajes y hacer el *dispatching* a las funciones definidas como servicios. Actualmente gRPC está implementado en la mayoría de los lenguajes de programación modernos.

Actualmente, una convención muy usada, principalmente como representación de mensajes RPC en aplicaciones y servicios web es [Representational State Transfer \(Rest\)](#), la cual consisten en los siguientes principios o técnicas:

1. Arquitectura cliente-servidor.
2. Sin estado: El servidor no mantiene estado con los clientes (no hay sesiones).
3. Interfaz (API) uniforme:
  - Un recurso se identifica por medio de [URLs] ([https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Identifier](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier))
  - Uso de comandos (verbos) HTTP: **GET** , **POST** , **PUT** , **DELETE** para la obtención, modificación, creación y eliminación de recursos/objetos.
  - Uso de JSON o XML para la representación de datos. Un recurso debe contener sus datos y los metadatos necesarios para que el cliente pueda manipularlo.

# Datos multimedia y compresión

La transmisión de datos multimedia presentan un desafío adicional ya que estos datos son generalmente muy voluminosos. Las aplicaciones multimedia de tiempo real deben utilizar formatos especializados y generados a partir de técnicas de compresión.

Un cuadro de video de TV de alta definición de 1080x1920 pixels, cada uno representado en 24 bits (RGB) ocupa  $1080 \times 1920 \times 24 = 50MB$ . Para transmitir a una tasa de 24 cuadros por segundo (fps), se requiere una tasa de transmisión de más de 1Gbps.

Sin utilizar técnicas de compresión se requeriría un ancho de banda, generalmente no disponible.

Las técnicas de compresión se pueden categorizar en:

1. Sin pérdida de información: Generalmente se aplica a la compresión de documentos.
2. Con pérdida: Aplicable a datos multimedia (imágenes, sonido y video).

Todos los métodos se basan en explotar la redundancia de datos para luego codificarlos en la menor cantidad de bits posibles.

Los algoritmos de compresión sin pérdida de información se basan en las siguientes estrategias:

- *Run length encoding (RLE)*: La idea es reemplazar las ocurrencias consecutivas de un símbolo por una sola copia del mismo mas un indicador de multiplicidad. Por ejemplo, la cadena **AAABCD<sub>4</sub>DD** se codifica como **3A2B1C4D** .
- *Differential Pulse Code Modulation*: A partir de un símbolo de referencia, el resto se reemplaza por la diferencia o distancia en el código original. Por ejemplo, la cadena **AAABCD<sub>4</sub>DD** se reemplaza por **A0001123333** , dadas las diferencias en el código ASCII desde el carácter **A** .

Las diferencias sólo ocuparían sólo 5 bits (sobradamente para representar diferencias de hasta 25, tomando en cuenta los 26 caracteres alfabéticos).

- Basados en *diccionarios*: La idea es que a partir de un diccionario en forma de tabla que contenga el conjunto de símbolos (por ejemplo palabras, en archivos de texto), se reemplaza cada símbolo por su índice en la tabla.

Estos métodos pueden usar *códigos de longitud variable*: Los símbolos que aparecen mas frecuentemente se codifican en menos bits. Un *código de Huffman* se basa en el diseño de un código de prefijos óptimo.

## Códigos de Huffman

Un código de Huffman se obtiene con el siguiente algoritmo:

Dados un alfabeto  $A = \{a_1, a_2, \dots, a_n\}$  y un mapping  $F : A \rightarrow \mathbb{R}$  que asocia las frecuencias de ocurrencias de cada símbolo, la salida es un código  $C(F) = \{c_1, c_2, \dots, c_n\}$ .

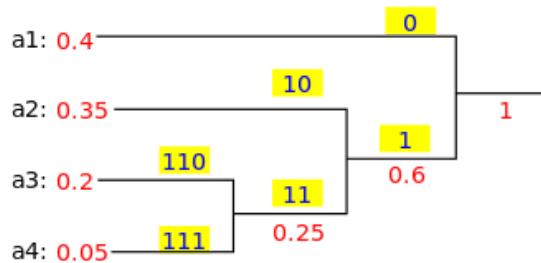
El objetivo es lograr un código óptimo, es decir que  $L(C(F)) < L(T(F))$  para cualquier código  $T(F)$ , donde

$$L(C(F)) = \sum_{i=1}^n F(c_i) \times \text{length}(c_i)$$

La propuesta de Huffman propone la construcción en forma ascendente de un árbol binario a partir de los símbolos y sus frecuencias convertidas en probabilidades como las hojas, inicialmente en una cola ordenados de mayor a menor frecuencia.

1. Mientras haya más de un nodo en la cola
2. Sacar dos nodos del frente de la cola (los de menor probabilidad).
3. Crear un nodo que sea el padre de los dos con probabilidad igual a la suma de sus hijos.
4. Insertar ordenadamente el nuevo nodo en la cola.
5. El nodo restante es la raíz (con probabilidad 1)

Luego se recorre el árbol desde la raíz hasta las hojas asignando a cada rama descendente de cada nodo un 0 a la rama izquierda y un 1 a la derecha. El código de cada símbolo es la concatenación de los rótulos de cada arco desde la raíz hasta el símbolo, como se muestra en la siguiente figura.



**Figura 12.3: Codificación de Huffman.**

Así el diccionario para el ejemplo de la figura queda formado como en la siguiente tabla:

Símbolo	Código
a1	0
a2	10
a3	110
a4	111

Cada código puede ser decodificado fácilmente por un receptor ya que el 0 actúa como marca de fin.

Los algoritmos del tipo Lempev-Ziv (zip, gzip, ...) se basan en estas ideas y construyen el diccionario en base al contenido y luego generan códigos de Huffman. El diccionario se incluye en el archivo.

## Compresión de imágenes

Una imagen *raster* es conceptualmente una matriz de píxeles, donde cada pixel se representa como una tupla de tres colores: rojo, verde y azul (RGB). Generalmente un dispositivo de captura (cámara) representa la intensidad de un color en 8 bits, por lo que cada pixel se representa en 24 bits.

Una técnica de compresión, como la usada en el formato GIF, se basa en representar cada pixel en un byte, el cual constituye un índice de una tabla conocida como la *paleta de colores* con 256 entradas. Cada entrada en la paleta define un color. Esto hace que cada pixel se asocie a un color *aproximado* en la paleta.

GIF además aplica una variante del algoritmo LZ para codificar en menos bits aquellos valores más frecuentes. Así, GIF logra un radio de compresión aproximado de 10:1.

Otra técnica más compleja es la desarrollada por el *Joint Photographic Experts Group (JPEG)*. A diferencia de GIF, JPEG no reduce el número de colores, sino que transforma cada pixel RGB en YUV, un espacio de colores que toma en cuenta la percepción del ojo humano. El espacio YUV se basa en que un color tiene una *luminancia (brillo)* y U y V determinan el *color o crominancia*. Generalmente se usan las siguientes ecuaciones de transformación:

$$Y = 0.299R + 0.587G + 0.114B$$

$$U = (B-Y) \times 0.565$$

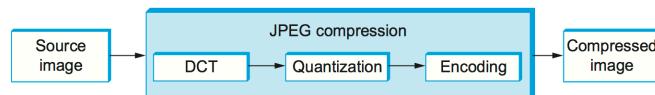
$$V = (R-Y) \times 0.713$$

El valor **U** representa la *proyección desde el azul* y **V** la *proyección desde el rojo*. Esta representación de colores viene desde la transmisión de TV analógica en color.

Existen otros espacios de colores, como el **CMYK** (cyan, magenta, yellow y black), comúnmente usada en impresoras color por inyección de tinta. Otros son HSB, y HSL. Para comparar espacios y sus transformaciones ver [colorizer.org](http://colorizer.org).

El espacio YUV permite comprimir (con pérdida de información) los componentes UV más agresivamente que el componente Y, el cual es el más perceptivo por el ojo humano.

JPEG luego realiza tres pasos para lograr la compresión como se muestra en la figura siguiente:



**Figura 12.4: Compresión JPEG.**

Cada paso opera sobre submatrices de 8x8 pixels. El primero aplica una función llamada *discrete cosine transformation*, similar a la transformada de Fourier, dando por cada pixel una *frecuencia*, la cual representa el *cambio* de valores tanto en el eje **x** como en el **y**. Un cambio más brusco genera una frecuencia más alta y viceversa.

El segundo paso, llamado *cuantización*, es que produce la pérdida de información, ya que *aproxima* valores descartando bits menos significativos.

El paso final es codificar los valores de cada submatriz siguiendo un recorrido en zigzag desde la esquina superior izquierda hasta la inferior derecha usando RLE.

JPEG comúnmente logra radios de compresión de 30:1.

Una alternativa imágenes raster son los gráficos *vectoriales*. Estos contienen una descripción de los elementos gráficos, como líneas, elipses, texto, etc, de la imagen.

Es muy común en la web el uso de imágenes vectoriales como [svg](#), los cuales se basan en un formato en XML y los navegadores web los pueden mostrar sin problemas. Las ventajas de los gráficos vectoriales es que son muy compactos y escalables, es decir que se pueden redimensionar sin perder resolución.

## Compresión de video

Un video básicamente es una secuencia de imágenes (cuadros o frames) junto con alguna información o metadatos que describe su tasa de reproducción y otros.

El estándar [MPEG](#) define el formato de contenedores multimedia (video, audio, subtítulos, ...) y algoritmos de compresión.

La compresión de video en MPEG crea cuadros de tres tipos:

1. *I-frames*: Intrapicture frames, los cuales son imágenes JPEG.
2. *P-Frames*: Predicted frames, contienen diferencias con el I-frame anterior.
3. *B-frames*: Bidirectional frames, contiene diferencias con los I-frames o P-frames anteriores y siguiente.

Los frames de diferencias generalmente tienen una gran cantidad de valores iguales por lo que se pueden comprimir agresivamente.

Los *encoders* deciden cada cuánto incluir I-frames y cuándo generar P-frames y B-frames.

Este esquema hace que este formato sea conveniente para su transmisión, generalmente usando protocolos basados en UDP como RTP. Si se pierde un frame de diferencia, el reproductor simplemente puede mostrar nuevamente el frame anterior, sin cambiar demasiado la ilusión de movimiento del espectador.

MPEG comúnmente alcanza radios de compresión de 90:1, aunque se han alcanzado radios de 150:1.

Existen otros estándares de formatos de video, como los de las series H definidos por el ITU-T, aunque difieren de MPEG sólo en pequeños detalles. Uno de los formatos más actuales es el H.264/MPEG, desarrollado en conjunto por los dos grupos.

## Compresión de audio

El audio digital se genera a partir de tomar muestras (*sampling*) discretas de la señal analógica a una cierta frecuencia.

Por ejemplo, el audio de calidad de CD se obtiene mediante un muestreo a 44.1KHz (aproximadamente cada  $23\mu s$ ) y cada valor discreto es de 16 bits. Transmitir en estéreo (2 canales) una tasa de  $2 \times 44.1 \times 1000 \times 16 = 1.41 \text{ Mbps}$ . En telefonía digital se toman muestras a 8KHz usando valores de 8 bits, lo que requiere una tasa de transmisión de 64kbps.

El estándar MP3 (MPEG level III) utiliza las mismas técnicas que MPEG. Primero divide el audio en subbandas, similar a la transformación YUV de MPEG, luego los divide en bloques, se quantizan y finalmente se codifican de manera similar.

La clave está en la selección de las frecuencias, lo cual se basa en modelos acústicos propuestos por otros investigadores.

## Contenedores multimedia

Un contenedor multimedia se basa en formatos para representar secuencias o flujos de video, audio y otros datos, como por ejemplo subtítulos y otros. Tienen cabeceras con metadatos que describen los flujos que contienen y la información necesaria de sincronismo y tasa de reproducción (fps).

Algunas aplicaciones de *streaming* de video, como YouTube o Netflix, soportan *streaming adaptable*. Un video se almacena en diferentes containers con diferentes calidades y tasas de reproducción. Los clientes solicitan de a grupos de frames de cierta calidad y dinámicamente se determina si el enlace permite o requiere cambiar de calidad, en cuyo caso el cliente requiere un nuevo grupo desde otro contenedor.

En aplicaciones web que usan DASH o HLS (como Youtube o Netflix), cada contenedor generalmente se asocia a un *URL* diferente para que el cliente solicite mediante solicitudes HTTP.

---

< Anterior

## Congestión

Próximo >

## Aplicaciones

# Aplicaciones

Cada aplicación debe definir su propio protocolo que a su vez se basarán en protocolos de transporte y de sesión. También deberá definir el formato o presentación de los mensajes. Por último deberá especificar la *coreografía* de las comunicaciones, que pueden ser simples del tipo *request/reply* o más complejas dependiendo si se basa en sesiones y otros detalles de la aplicación.

La definición completa de un protocolo de aplicación incluye:

1. Entidades participantes
2. Nombrado de recursos
3. Formato de mensajes
4. Interacción. Generalmente dado por un conjunto de sistemas de transición de estados.
5. Protocolos de transporte y otros utilizados

## Aplicaciones de Internet

En Internet las aplicaciones iniciales, descriptas en la RFC 1123 son las siguientes:

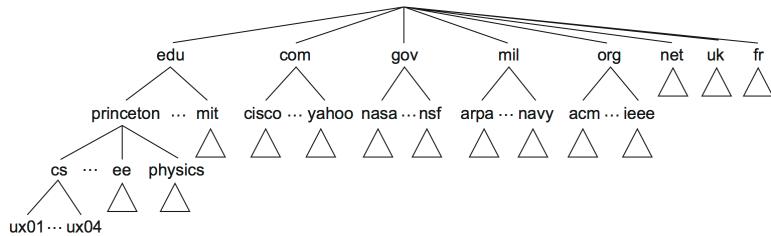
- Login remoto: Telnet y login
- Transferencia de archivos: File Transfer Protocol (FTP y TFTP)
- Correo electrónico: Simple Mail Transfer Protocol (SMTP y POP o IMAP)
- Soporte: Domain Name System (DNS)
- Inicialización de hosts: BOOTP
- Gestión y monitoreo de redes: Simple Network Management Protocol (SNMP)

A continuación se describen las aplicaciones mencionadas y finalmente se analizan las aplicaciones modernas como servicios web, aplicaciones multimedia y otras.

## Dominios y DNS

Identificar hosts y routers por medio de direcciones no es cómodo para los seres humanos, por lo que es necesario nombrar hosts y redes en Internet usando nombres o identificadores más legibles y fáciles de recordar.

La idea es partitionar Internet en *dominios* y usar una estructura de nombres jerárquica. Cada dominio define un espacio nombres lógico en que se incluirán servidores, sitios web, etc. La siguiente figura muestra algunos dominios usados en Internet.



**Figura 13.1: Espacios de nombres o dominios de Internet.**

En el nivel superior encontramos los dominios raíz. La idea es que las redes que pertenezcan a instituciones educativas estén en el dominio **edu**, las organizaciones sin fines de lucro en **org**, etc. También existen dominios raíz para cada país, denotados por una abreviatura de dos letras, para definir dominios geográficos.

La asignación de nombres se realiza por diferentes organizaciones. Los dominios *top-level* son manejados por la *Internet Corporation for Assigned Names and Numbers (ICANN)*. Dentro de cada país, los nombres son gestionados por el *Network Information Center (NIC)* correspondiente.

En la Argentina el **nic.ar**, depende de la Secretaría Legal y Técnica de la Presidencia de la Nación y está integrada por el gobierno nacional, empresas de servicios de Internet y otras organizaciones.

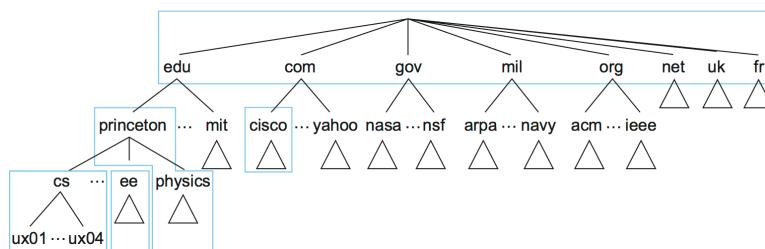
Quien haya obtenido su dominio, tiene la libertad de dividir su propio espacio de nombres en *subdominios*. Por ejemplo, Wikipedia tiene el dominio **wikipedia.org** y a su vez ha definido subdominios para diferentes lenguajes como **en.wikipedia.org** o **es.wikipedia.org**.

Un dominio puede referir a un host o una aplicación o servicio. Por ejemplo, **www.example.com** refiere a la IP de un host que corre un servicio web (website).

El *Domain Name System (DNS)* es un sistema distribuido de soporte formado por una red de servidores que implementan un protocolo del tipo *requerimiento/respuesta* cuya función es dado un nombre o dominio, retornar un valor.

Generalmente el uso común del DNS es hacer consultas para obtener la IP de un nombre asociado a un host o una aplicación.

El espacio de nombres está dividido en *zonas*. Una zona debe desplegar al menos un servidor DNS que pueda responder a los nombres dentro de su zona. Los dominios *top-level* están manejadas por el ICANN. Luego las diferentes instituciones u organizaciones manejan los dominios bajo su control, como se muestra en la siguiente figura.



**Figura 13.2: Zonas en Internet.**

Dentro de cada zona, un servidor DNS define su base de datos local. La base de datos consta de registros que tienen los siguientes campos:

- *Name*: El nombre del recurso
- *Value*: El valor asociado al nombre
- *Type*: Tipo de registro como **A** (host), **NS** (name server), **MX** (mail exchanger) y otros.
- *Class*: En Internet tiene siempre el valor **IN**.
- *Time To Live (TTL)*: Tiempo de validez. Este valor es usado por los servidores que almacenan registros *aprendidos* en su caché.

Por ejemplo, el siguiente registro (obtenido con cliente DNS como **dig**)

```
dc.exa.unrc.edu.ar.
```

```
dc.exa.unrc.edu.ar.      1630      IN      A      200.7.141.44
```

describe un host que es el servidor web del departamento de computación de la Facultad de Ciencias Exactas de la Universidad Nacional de Río Cuarto, que está dentro del dominio `edu.ar`.

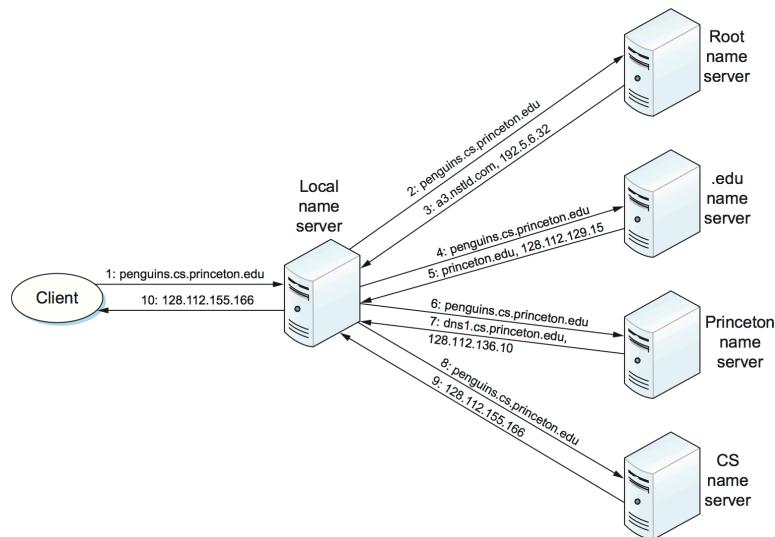
Ante la consulta `dig dc.exa.unrc.edu.ar MX` (registros del tipo MX), se obtiene

```
dc.exa.unrc.edu.ar.    3600    IN    MX    20 ALT1.ASPMX.L.GOOGLE
dc.exa.unrc.edu.ar.    3600    IN    MX    20 ALT2.ASPMX.L.GOOGLE
...
...
```

Una aplicación que deba acceder a un cierto host dado su nombre deberá consultar a su DNS local, el cual resolverá a su IP para que pueda comunicarse.

Recordemos que cada host tiene en su configuración de red la IP de al menos un servidor de nombres primario. En muchos sistemas tipo UNIX, como GNU-Linux, se definen en `/etc/resolv.conf`. Es común que también se defina la IP de un DNS secundario. Este DNS se considera el DNS local para el host, lo cual no significa que resida en su red local.

La siguiente figura muestra la resolución de un nombre desde un host y describe la secuencia de mensajes entre los servidores DNS hasta llegar al DNS de la zona correspondiente, el cual contendrá el valor asociado a la consulta realizada.



*Figura 13.3: Resolución (recursiva) de nombres.*

El DNS local luego de obtener el registro, lo almacena en su caché para futuras consultas.

Esta arquitectura configura un sistema de bases de datos jerárquica distribuida.

En una red local, generalmente el router actúa como un *resolver DNS* que simplemente *reenvía* las consultas desde el interior de la red local hacia un DNS del ISP y usa una caché para futuras consultas. Esto previene mayor tráfico DNS hacia el exterior de la red local.

El protocolo DNS utiliza UDP aunque existen propuestas experimentales basadas en TCP y usando protocolos seguros.

## Acceso (login) remoto

En los primeros años de Internet una de las primeras aplicaciones desarrolladas fue un servicio de acceso remoto (a mainframes, principalmente). Accediendo remotamente a su cuenta de trabajo un usuario podía usar aplicaciones de acceso a su correo electrónico (en esa época el acceso a los buzones de correo era local) y otras aplicaciones (talk) en sistemas multiusuario.

Una de las primeras aplicaciones de acceso remoto fue **telnet** (RFC 15 y 854). Esta aplicación cliente-servidor permite acceder a un servidor remoto, el usuario presenta sus credenciales y luego interactúa con una sesión de shell.

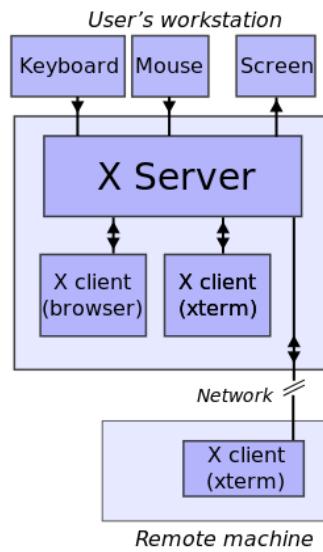
El protocolo de **telnet** usa TCP y tiene una coreografía muy simple: El servidor interactúa con un sistema de *login* y luego con un *shell*. El servidor notifica al cliente de eventos ocurridos enviando *urgent data* en mensajes TCP. El puerto asignado a *telnet* es el 23.

El cliente es un programa genérico que lee comandos ingresados por el usuario, envía al servidor, el cual los procesará y retornará una respuesta. Es posible ver que la sesión es un ciclo de un protocolo *request/reply*. En una terminal o consola de comandos, generalmente se utiliza de la siguiente manera:

```
sh
telnet <host> [port]
```

Actualmente aún es usado principalmente en administración de servidores dentro de una red segura, ya que no ofrece mecanismos de confidencialidad (todo se transmite en texto plano) y para testing de protocolos. Generalmente se ha reemplazado por *secure shell (ssh)* ya que éste brinda confidencialidad por medio de SSL/TLS.

Las aplicaciones de acceso remoto modernas permiten sesiones con *shells* gráficos, tal como en el sistema X usado en interfaces gráficas en los sistemas tipo UNIX. Su arquitectura se muestra en la siguiente figura.



**Figura 13.4: El sistema X-Window.**

En *X* las aplicaciones gráficas toman el rol de clientes, las cuales interactúan con el *X-server* por medio de mensajes (TCP en conexiones remotas y UNIX sockets en conexiones locales). El *X-server* interactúa con el hardware gráfico (por medio de una interface provista por el sistema operativo) y corre como una aplicación de usuario.

Se debe notar que el *X-server* corre en la máquina *cliente*, es decir con la que usuario interactúa.

Los clientes generalmente utilizan una biblioteca gráfica (*Xt*, *GTK*, *KDE*, etc) cuyas funciones abstraen los detalles de la comunicación. Las funciones permiten enviar al server comandos de dibujo como puntos, líneas, imágenes y otras. El *X-server* envía a los clientes (aplicaciones gráficas) eventos de mouse, teclado y otros dispositivos de interacción usados.

Cuando se usa *X* en forma remota, generalmente por un *túnel* en una sesión *ssh*, las aplicaciones gráficas (clientes *X*) corren en el host remoto, el cual no necesariamente debe tener hardware gráfico.

Algunos protocolos usados ampliamente para control remoto son *Remote Desktop Protocol (RDP)* de Microsoft y el *Remote Frame Buffer Protocol (RFB)* usado en VNC y alternativas libres como [KasmVNC](#).

Actualmente es común utilizar servicios de acceso remoto como *TeamViewer* o similares. Algunos permiten control vía Internet utilizando HTTP como protocolo de transporte.

La conexión entre el cliente y el servidor (la máquina bajo control) puede ser directa si la conexión lo permite. Muchas veces no es posible cuando un host está detrás de routers que realizan NAT o firewalls. En este último caso ambos extremos se conectan a un servidor que hace de *relay* en ambas direcciones.

## Correo electrónico

El correo electrónico es un servicio de mensajería aún muy utilizado.

Inicialmente sólo soportaba mensajes de texto. En la actualidad los mensajes pueden ser *multiparte* donde cada parte define su formato. Los formatos pueden ser textos con diferentes codificaciones y anexos con contenidos de archivos binarios, multimedia, etc. Estas extensiones se conocen como

[Multipurpose Internet Mail Extentions \(MIME\)](#).

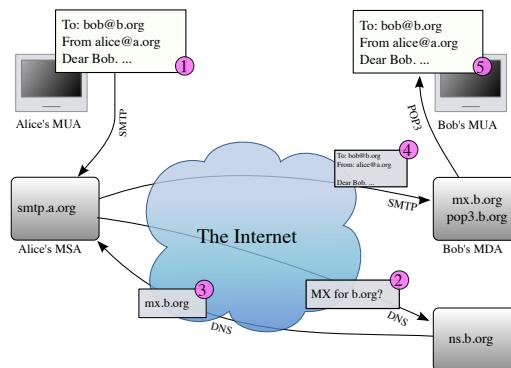
La arquitectura del sistema consiste en una red de servidores (*Mail Transport Agents (MTA)*). Cada dominio generalmente tiene un servidor de correo electrónico en el cual se registran los usuarios. La identidad de un usuario (naming) se denota de la forma `user@domain`. Cualquier usuario puede enviar mensajes a cualquier otro en Internet.

Los usuarios normalmente utilizan una aplicación cliente o *Mail User Agent (MUA)* como *Mozilla Thunderbird*, *MS-Outlook*, clientes web como GMail u otros. El cliente se conecta a un servidor *Simple Mail Transport Protocol (SMTP)* para el envío de nuevos mensajes y a un servicio de acceso al buzón basados comúnmente en *Internet Access Message Protocol (IMAP)* o *Post Office Protocol (POP/POP3)*. Estos últimos dos protocolos permiten al usuario gestionar los mensajes enviados y recibidos como clasificarlos en diferentes *mailboxes* y definir su *estado* como leídos/no leídos y otros.

Los mensajes inicialmente almacenados en el servidor, pueden moverse o copiarse al cliente, según la configuración del usuario.

SMTP (port 25) es orientado a conexión y actualmente es común el uso de su versión extendida *ESMTP* o *SMPTS* (port 587) que permite comunicaciones seguras sobre TLS y alternativas de autenticación.

Para el envío de mensajes desde el cliente al servidor (*mail submission agent o MSA*) también se usa SMTP generalmente en una sesión autenticada. El MSA reenvía el mensaje al MTA local para que lo entregue al servidor destino (en una sesión SMTP no autenticada), el cual a su vez lo entregará al MSA en el servidor del destinatario, como se muestra en la siguiente figura:



**Figura 13.5: Arquitectura de Internet Email.**

Generalmente las instancias del MSA y MTA se implementan en el mismo código del servidor y son simplemente diferentes instancias del servidor SMTP lanzadas con diferentes opciones. La instancia del MSA generalmente requiere una sesión con autenticación y generalmente escucha en el puerto 587, mientras que el servicio SMTP de los MTAs usan el puerto 25.

Como se puede apreciar en la figura, el MTA de origen obtiene la dirección IP del servidor de destino consultando por registros MX al sistema DNS. En el caso que no pueda conectarse en ese momento el servidor realizará varios reintentos.

En caso de falla de un envío ya sea porque no es posible contactar al servidor del dominio de destino o porque éste retorna un error al emisor indicando que el usuario no existe en su dominio, el sistema envía un mensaje al emisor notificando del error.

El formato de un mensaje (ver RFCs 5321 y 5322) consiste en una *cabecera* y su *cuerpo* o contenido. La cabecera mínimamente incluye las direcciones de emisor y receptores, fecha y hora del envío y el título o *subject*. Cada servidor SMTP por los cuales pasó el mensaje, generalmente incluye una línea de cabecera de la forma

```

Received: from us11-010mrr.dh.atmailcloud.com ([10.10.5.20])
by us11-011ms.dh.atmailcloud.com with esmtp (Exim 4.90_1)
(envelope-from <someuser@example.atmailcloud.com>)
  
```

```
id 1gYlz6-0005Df-Hx
for someuser@example.atmailcloud.com; Mon, 17 Dec 2018 16:02
```

Un mensaje puede pasar por varios servidores porque es común que en un dominio los usuarios se partitionen en grupos o subdominios manejados por varios servidores y un mensaje deba seguir una *ruta* hasta su destino.

La cabecera se separa del cuerpo por una línea en blanco.

A continuación se muestra un ejemplo de una sesión SMTP entre el cliente y un servidor.

```
C: telnet smtp.example.com 587
S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.org
S: 250 Hello relay.example.org, I am glad to meet you
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.org>
C: To: "Alice Example" <alice@example.com>
C: Cc: theboss@example.com
C: Date: Tue, 15 Jan 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields and 4 lines in th
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345
C: QUIT
S: 221 Bye
{The server closes the connection}
```

El servidor inicia la sesión con un mensaje de bienvenida al cual el cliente debe responder con el comando `HELO` (`EHLO` para indicar que desea usar ESMTP). Luego El cliente envía mensajes `MAIL FROM` para establecer el usuario emisor y `RCPT TO` para cada destinatario. Ante cada comando, el servidor responde con un código de estado o resultado.

El comando `DATA` indica el comienzo del mensaje compuesto de la cabecera y del cuerpo. Una línea conteniendo un punto (caracter `.`) indica el final de la entrada del mensaje ante lo cual el servidor responde con el identificador asignado. En la sesión es posible continuar interactuando con el servidor SMTP hasta que el cliente envíe el comando `QUIT`, momento en que el servidor cierra la conexión TCP.

Algunos de los servidores SMTP con licencias de software libre mas utilizados son [sendmail](#) y [postfix](#).

## Acceso a *mailboxes*

Los protocolos de acceso y gestión de *mailboxes* como IMAP o POP3 también tienen una arquitectura cliente-servidor. Se aconseja asociar diferentes nombres a cada servicio como `smtp.example.com` e `imap.example.com` aunque ambos servicios estén ejecutándose en el mismo servidor.

Actualmente el protocolo IMAP se ha constituido como el protocolo de acceso preferido por sobre POP3 ya que es más flexible y permite, por ejemplo transferir sólo las cabeceras de mensajes para que el usuario decida si recuperar el cuerpo, reduciendo así el volumen de transferencia.

Tanto IMAP como POP utilizan TCP. Los puertos asignados para POP3 son el 110 y el 995 en forma segura y 143 o 993, respectivamente para IMAP. Ambos se basan en comandos provistos por el cliente. Los comandos más relevantes de IMAP se listan en la siguiente tabla:

Comando	Descripción
<code>SELECT &lt;mb&gt;</code>	Selecciona el <i>mailbox</i> <code>mb</code>
<code>CREATE &lt;mb&gt;</code>	Crea un nuevo <i>mailbox</i>
<code>DELETE &lt;mb&gt;</code>	Borra un <i>mailbox</i>
<code>LIST &lt;mb&gt; &lt;f&gt;</code>	Lista los mensajes usando el filtro <code>f</code>
<code>STATUS</code>	Muestra el estado de los mensajes

Comando	Descripción
<code>FETCH</code>	Lista los mensajes y su estado
<code>SET &lt;flags&gt;</code>	Setea flags (no/leído, ...)
otros	Mover, borrar mensajes, ...

Uno de los servidores IMAP (software libre) ampliamente utilizados es [dovecot](#).

## Transferencia de archivos

Unas de las primeras aplicaciones para transferencias de archivos fue el *File Transfer Protocol (FTP)* y tiene una arquitectura cliente-servidor sobre TCP. Una vez lograda la conexión (por ejemplo, mediante el comando `ftp ftp.example.com` o un cliente gráfico) generalmente el servidor requiere la autenticación por medio de usuario y contraseña. Muchos servidores proveen *ftp anónimo* mediante el usuario *anonymous*.

Luego el cliente puede realizar comandos como listar ( `list` ) el directorio actual (initialmente, por lo general la carpeta *home*), cambiar de directorio ( `cd` ), borrar, renombrar archivos o descargar ( `get` ) y subir ( `put` ) archivos.

En un comando `get` o `put` FTP abre una conexión de datos desde el servidor al cliente para realizar la transferencia. Como eso puede ser inconveniente para clientes detrás de servicios de NAT o firewalls, se puede hacer conexiones *pasivas* mediante el comando `PASV`, es decir desde el cliente al servidor. Para esto, previamente el cliente envía un mensaje solicitando al servidor la IP y puerto para establecer la conexión desde el cliente.

Existen otros protocolos para la transferencia de archivos. Actualmente se utiliza ampliamente HTTP, el protocolo de transporte de la web, el cual es más simple y usa una única conexión TCP. A diferencia de FTP, HTTP es sin estado, es decir no existe el concepto de directorio corriente y otros detalles. Los archivos se acceden mediante URLs que el usuario debe conocer.

Recientemente los navegadores web han dejado de soportar el protocolo `ftp`.

*Secure shell (ssh)* también incluye un protocolo de transferencia segura. Ver el comando `scp` (secure copy).

# WWW

La web se ha convertido en una de las aplicaciones de mayor uso en Internet. Los sitios web han pasado de ser conjuntos de documentos estáticos a aplicaciones que permiten generar contenido dinámico y dan posibilidades de interacción a los usuarios, mediante la creación o actualización de contenidos o mediante comentarios integrados con otras aplicaciones de mensajería o redes sociales.

La web fue desarrollada por [Tim Berners-Lee](#) con el objetivo de formar una red de documentos de interés científico inter-relacionados. Un documento puede contener enlaces a otros documentos en otro sitio.

La idea es simple: Los usuarios utilizan una aplicación cliente, como un *navegador (browser) web* u otra aplicación que envía requerimientos a servidores web los cuales responden con contenidos como documentos HTML, objetos multimedia u otros datos, los cuales son procesados por el cliente.

Como un documento puede contener referencias a otros objetos, posiblemente en otros sitios, el navegador realiza esos requerimientos adicionales automáticamente hasta obtener todos los objetos necesarios.

Inicialmente los navegadores usaban una interfaz de texto. El primer navegador gráfico fue [Mosaic](#), desarrollado por NCSA. Luego este proyecto derivó en la creación de *Netscape Communications*, que lanzó el *Netscape Navigator*, que a su vez es el predecesor del *Mozilla Firefox*. El código de *Internet Explorer* (*Microsoft*) también fue un derivado inicial de Mosaic. Actualmente existen varios navegadores y los más utilizados son Firefox (software libre) y Google Chrome.

El primer servidor web fue el *httpd* desarrollado en el [CERN](#) en 1991. Actualmente existen varias alternativas. Los más usados son productos con licencias libres como [Apache](#) y [NGINX](#).

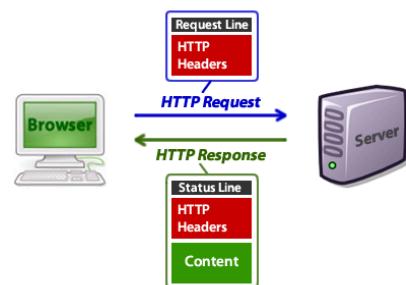
El formato de los documentos, el [HyperText Markup Language \(HTML\)](#), se basa en XML y ha evolucionado (actualmente, HTML5) para contener objetos multimedia como audio y video permitiendo construir documentos complejos e interactivos.

La interactividad y contenido dinámico se logra gracias a que un documento HTML permite incluir programas (scripts) en lenguajes de programación como Javascript que permiten manipular el documento, realizar comunicaciones, almacenar información localmente (en almacenamiento gestionado por el navegador web), validar datos de formularios y otras funciones.

La apariencia de los elementos en un documento se describe por medio del lenguaje [\*\*Cascade Style Sheets \(CSS\)\*\*](#) que contienen reglas de formato aplicables a las diferentes partes de un documento.

El protocolo de transporte para el acceso a recursos, como documentos o páginas web, imágenes u otros datos es el [\*\*HyperText Transport Protocol \(HTTP\)\*\*](#).

El protocolo es del tipo cliente/servidor y aunque generalmente opera sobre TCP es sin estado.



**Figura 13.6: Requerimiento/respuesta HTTP.**

Este protocolo evolucionó desde HTTP/1.0 descripto en la [\*\*RFC 1945\*\*](#) en 1996, pasando por HTTP 1.1 ([\*\*RFC 2616\*\*](#)) en 1999 hasta [\*\*HTTP/2\*\*](#) propuesto por Mozilla en 2015 que permite conexiones sobre TCP seguras usando TLS. HTTP/3, propuesto en 2020, aún se encuentra en estado *borrador (draft)*. Se basa en el uso de [\*\*QUIC\*\*](#), el cual se analizó en el capítulo de protocolos de transporte y actualmente ya está soportado por algunos navegadores web y usado opcionalmente en algunas aplicaciones web como los servicios de Google.

Los mensajes se representan en texto y como se puede apreciar en la figura de arriba y tienen las siguientes partes:

1. El comando ( [\*\*GET\*\*](#) , [\*\*POST\*\*](#) , [\*\*PUT\*\*](#) , [\*\*DELETE\*\*](#) , ...) en un requerimiento o la línea de estado en una respuesta (por ejemplo: [\*\*200 OK\*\*](#) , ...)

2. Una secuencia de líneas de encabezados o *headers* de la forma `header : value`
3. Una línea en blanco
4. El cuerpo o contenido opcional. En el caso de las respuestas puede ser el texto HTML o datos multimedia representado en alguna codificación como base64 o directamente en binario. En un requerimiento `POST` generalmente van los datos ingresados por el usuario en un formulario web.

Cada recurso se identifica por su Uniform Resource Identifier que tiene el siguiente formato:

`URI = scheme:[//authority]path[?query][#fragment]`

Un *Uniform Resource Locator (URL)* es un URI en el cual se indica su método de acceso, como por ejemplo su localización en la red, como por ejemplo:

`https://mysite.com/mypicture.png`. Un *Uniform Resource Name (URN)* identifica un recurso por su nombre o identificador. Por ejemplo, el URN `urn:isbn:0-486-27557-4` identifica una edición específica del libro *Romeo y Julieta*.

## Web APIs

Para el desarrollo de aplicaciones web se desarrollan APIs básicamente sobre el lenguaje de programación Javascript, ampliamente aceptado como lenguaje de scripting en navegadores web y también para la implementación de servicios generalmente sobre plataformas del tipo nodejs.

Algunas APIs soportadas por los navegadores modernos son las siguientes:

- *Document Object Model (DOM)*: Manipulación de la representación del documento HTML
- *Fetch*: Requerimientos a servidores y recepción de respuestas
- *Canvas*: Generación de gráficos
- *Geolocalización*: Permite descubrir la ubicación del dispositivo
- 

## Monitoreo y control

La administración de redes de tamaño considerable requieren herramientas de control y monitoreo.

Uno de los primeros protocolos desarrollados para este fin fue el *Simple Network Management Protocol (SNMP)*. Se basa en que cada nodo de importancia como un servidor o un router ejecuta el servicio el cual responde a consultas sobre su estado por parte de un cliente usado por el administrador de la red.

SNMP está implementado sobre UDP y permite que el administrador defina *variables de estado* o *Management Information Bases* en la cual se pueden asignar y consultar sus valores por medio de comandos **GET** y **PUT**.

Estas variables se actualizan mediante un *SNMP agent* que monitorea el dispositivo y actualiza su estado en el *SNMP server*, el cual puede ser consultado por sus clientes.

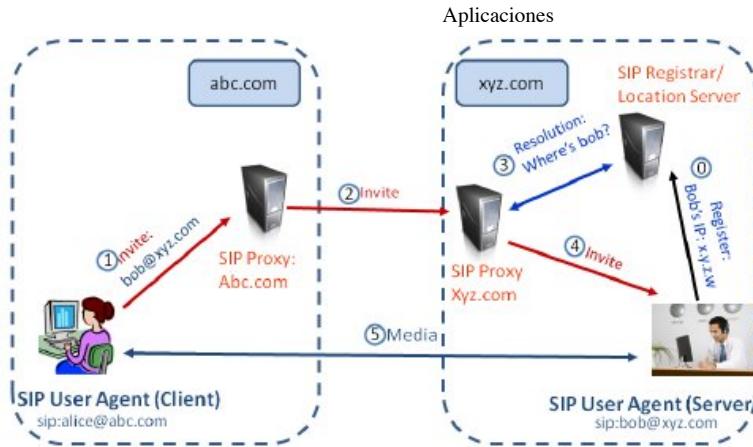
Otro sistema de control y monitoreo muy usado actualmente es [OpenConfig](#). Se basa en que los fabricantes de dispositivos especifican los *modelos* de datos, como un conjunto de variables y sus significados. Estos modelos se especifican en el lenguaje [YANG](#). Los dispositivos transmiten su estado (telemetría) por medio de *Network Configuration Protocol (NETCONF)* o *gNMI (gRPC Network Management Interface)*, lo cual permite implementar RPC sobre HTTP.

Lo atractivo de OpenConfig es que es posible el monitoreo y control de la red con prácticamente *cero configuración*.

## Aplicaciones multimedia

Las aplicaciones multimedia como telefonía IP (VoIP), chats y video-conferencias requieren de otros servicios como localización y estado de los usuarios, establecimientos y cierres de sesiones, como por ejemplo en una video-llamada y negociación de formatos multimedia y otros parámetros entre los extremos.

El [Session Initiation Protocol \(SIP\)](#) es un protocolo de *señalización (signaling)* usado para inicio de sesiones generalmente multicast. Usa mensajes en formato de texto y se usa principalmente para implementar las operaciones de inicio y finalización de llamadas de audio y/o video como se muestra en la siguiente figura.



**Figura 13.7: Llamada con SIP.**

El usuario utiliza un *SIP phone*, una aplicación o dispositivo de comunicación de voz/video. Cada dispositivo se conecta a un *proxy* y se *registra* en un servicio que provee localización de los dispositivos, dando soporte a dispositivos móviles. Generalmente cada dominio establece uno o más *proxies* de registro de los usuarios.

En una llamada, SIP realiza los siguientes pasos:

1. Envía un mensaje **INVITE** al proxy con el dominio de destino.
2. El proxy de destino solicita la *localización* del dispositivo al servidor de localización correspondiente al dominio.
3. Reenvía el mensaje **INVITE** al destino. Esto hace que el dispositivo *suene*.
4. Si el receptor acepta la llamada, comienza la comunicación multimedia.
5. Cualquier extremo puede finalizar la llamada (mensaje **BYE**).

La comunicación multimedia generalmente incluye la transmisión de un descriptor del medio, generalmente usando el **Session Description Protocol (SDP)** el cual, entre otras cosas, indica el protocolo de transporte de tiempo real a utilizar, como por ejemplo RTP.

```
v=0
o=jdoe 2890844526 2890842807 IN IP4 10.47.16.5
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.example.com/seminars/sdp.pdf
e=j.doe@example.com (Jane Doe)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
```

```
m=video 51372 RTP/AVP 99  
a=rtpmap:99 h263-1998/90000
```

En [multimedia Applications](#) del libro en línea *Computer Networks: A System Approach* se describe en mayor detalles los protocolos SIP y estándares recomendados por la ITU como H.323 y H.225.

## WebRTC

[Web Real Time Communication](#) permite realizar comunicaciones en tiempo real peer-to-peer, es decir, browser a browser. Estas APIs son comúnmente usadas en aplicaciones web de video-conferencia o video/voice/chat.

WebRTC define varias APIs Web:

- Captura de *streams* desde dispositivos multimedia (cámara, micrófono, captura de pantalla).
- Permite intercambiar datos como texto y streams multimedia.
- Los navegadores web modernos no requieren *plugins* adicionales.

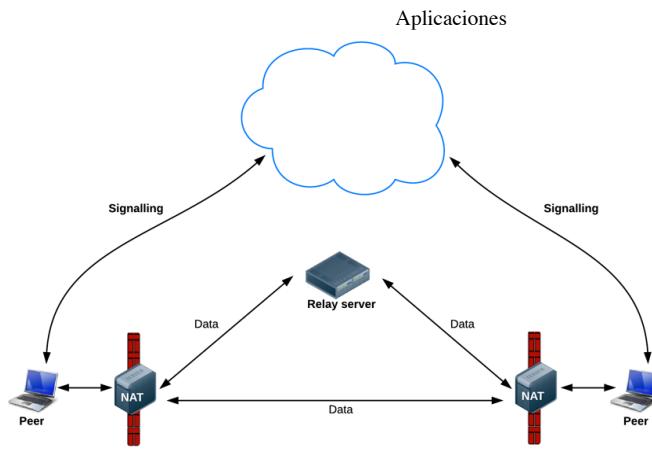
Uno de los principales problemas a resolver es la conectividad entre browsers en máquinas con IP privadas que están detrás de routers o firewalls que hacen NAT (en particular masquerading).

Para lograr de encontrar una forma de interconectar los peers se usa [Interactive Connectivity Establishment \(ICE\)](#), el cual usa generalmente *STUN servers*.

[STUN](#) es un protocolo que permite descubrir el par *IP:PORT* de un peer detrás de un *NAT router*.

ICE recolecta esa información conocida como *ICE candidates*.

En el caso que no se pueda lograr conectividad directa entre los peers se puede usar un [TURN relay server](#), el cual *retransmite* los datos entre los peers.



**Figura 13.8: Comunicaciones peer-to-peer con WebRTC.**

Cada peer deberá comunicar a los demás la forma de alcanzarlo (*ICE candidates*) y además intercambiar los parámetros de la sesión usando mensajes de Session Description Protocol (SDP). Estos mensajes incluyen información sobre los medios (flujos) a transmitir, como video, audio o datos, su codificación, formatos, compresión, etc.

Un peer generalmente iniciará la sesión (o llamada), mediante el envío de un mensaje SDP *offer*, que incluye los parámetros de la comunicación, a los demás peers, los cuales responderán con un mensaje SDP de tipo *answer* (incluyendo también sus parámetros).

Además cada peer deberá enviar los *ICE candidates* a los demás, para que puedan conectarse entre sí.

Este intercambio de mensajes se debe hacer bajo un *protocolo de inicio de sesión (SIP)*, conocido también como un *signal channel* o *signaling service*.

Este servicio no está especificado en WebRTC y generalmente se realiza mediante un servidor simple de intercambio de mensajes usando websockets o cualquier otro protocolo de comunicación.

Este servidor simplemente tiene como objetivo *reenviar* un mensaje de un peer a los demás. No necesita realizar ningún análisis de los mensajes.

Finalmente, los peers pueden intercambiar los distintos flujos de datos usando RTP u otros protocolos de transporte de tiempo real, según lo descripto en los mensajes *SDP*.

En WebRTC samples se pueden encontrar ejemplos para experimentar y analizar su código.

# Servicios distribuidos

Actualmente muchas empresas basan su modelo de negocios ofreciendo parte de sus recursos de computación, almacenamiento e infraestructuras de software a clientes.

Esto tiene la ventaja que muchas empresas pequeñas o medianas no necesitan desplegar instalaciones importantes de hardware para desplegar sus aplicaciones.

Además es posible implementar servicios complejos como geolocalización (mapas), seguimiento de mercadería (tracking) y otros utilizando servicios web de terceros.

Un *web service* es una aplicación que usa HTTP y eventualmente tecnologías de RPC como SOAP o REST con una API definida. A su vez un servicio web puede utilizar otros. Muchas aplicaciones web modernas ofrecen una API para acceder a sus servicios como [Google Cloud](#) y [AWS](#). Los servicios pueden ser variados, desde información de clima hasta servicios de computación o despliegue de software en la nube.

Con la aparición de la web, surgió el negocio de *hosting* de sitios y aplicaciones web. Estas aplicaciones corren en servidores (generalmente virtuales) ofrecidas por el proveedor de hosting.

Los avances en las tecnologías de virtualización y los *containers* permitieron que se ofrezcan plataformas de hardware y software de base y frameworks para el desarrollo de diferentes tipos de aplicaciones, incluyendo aquellas que requieren alto poder de cómputo. Esto derivó en lo que se conoce como *computación en la nube*.

El término [\*cloud computing\*](#) se refiere a servicios de computación y almacenamiento ejecutando en *data centers* distribuidos mundialmente. Los usuarios *alquilan* poder de cómputo en máquinas virtuales y/o contenedores y junto con software de base y frameworks necesario para el desarrollo de sus aplicaciones permitiendo su despliegue en una plataforma totalmente transparente en cuanto a su ubicación geográfica.

Los usuarios pueden contratar mas recursos a medida que los necesiten.

## Overlay networks

Estos servicios y otras aplicaciones distribuidas, como las redes sociales, generalmente desarrollan *overlay networks*, las cuales forman una red de *data centers* interconectados sobre Internet, generalmente formando sus propios sistemas autónomos.

Un ejemplo de esto son las *redes de distribución de contenidos (CDNs)* que ofrecen servicios de almacenamiento de recursos generalmente ampliamente usados por otros servicios, como aplicaciones web basadas en frameworks de Javascript usadas en aplicaciones web, imágenes y videos. Cada *data center* contiene los recursos *replicados* y actualizados con alguna política de *caché*.

Una aplicación cliente (como un web browser) al acceder a un recurso mediante su URL a una CDN, en realidad se conecta generalmente con un *redirector*, el cual le dará información de conexión al servidor mas conveniente, ya sea por cercanía o disponibilidad. Estos *redireccionadores* generalmente se basan en servidores DNS que realizan *balance de carga*.

Generalmente utilizan un conjunto de ISPs a nivel mundial que les permiten interconectar los diferentes data centers de manera efectiva, formando una red de un servicio especializado *sobre* Internet.

Servicios como los de Google, Youtube y Netflix operan sobre este tipo de redes.

---

< Anterior

Presentación

Próximo >

Seguridad

# Seguridad

Los protocolos analizados hasta ahora no contemplan mecanismos de seguridad. Internet es una red pública y no se debería confiar en ninguno de los routers o extremos en una comunicación crítica.

Actualmente es muy común las transacciones bancarias u operaciones de compras y ventas en Internet, lo que requiere que se garanticen la *autenticación* de las partes y la *confidencialidad* de los datos como por ejemplo los de una tarjeta de crédito. Además hay que tener en cuenta ciertos tipos de ataques como el *man in the middle* en el cual el atacante puede interceptar y modificar o reemplazar los mensajes entre los extremos, por lo cual es necesario proveer *integridad* de los mensajes: Un mecanismo que permite determinar que un mensaje no ha sido alterado en el camino.

También existen otros ataques conocidos como *denial of service (DoS)* que consisten en *bombardear* a un servidor con mensajes que lo sobrecargan, impidiéndole procesar y responder a los mensajes lícitos de los clientes. Esta característica que se debe preservar se conoce como *disponibilidad*.

En este capítulo se describen los problemas de seguridad y las técnicas, mecanismos y protocolos para sus soluciones.

Si bien la seguridad en redes normalmente se basa en el uso de protocolos seguros y mecanismos auxiliares también hay que tener en cuenta otros aspectos, principalmente en el desarrollo de aplicaciones, como las políticas y mecanismos de *control de acceso*, para evitar acceso a recursos por usuarios no autorizados.

Hay varios modelos de control de acceso:

- *Discretionales*: Los usuarios determinan quién y cómo pueden acceder a sus recursos. Un ejemplo de este modelo son las *listas de control de acceso* usados en los sistemas de archivos de los sistemas operativos.
- *Obligatorios*: El acceso a los recursos es definido por el *administrador de seguridad* y es el modelo comúnmente usado en bases de datos.

# Criptografía

La criptografía es uno de los mecanismos principales para asegurar propiedades como autenticación, confidencialidad e integridad.

Asumiendo que las comunicaciones se realizan sobre una red pública y por lo tanto un atacante puede *observar secretamente (eavesdropping)* los mensajes, es necesario *cifrar* los mensajes para garantizar la confidencialidad.

Un sistema de cifrado es una tupla  $(P, C, E, D, K)$  donde

- $P$  es el conjunto de los *textos planos*
- $C$  es el conjunto de los *textos cifrados*
- $E : P \rightarrow C$  es la *función de cifrado*
- $D : C \rightarrow P$  es la *función de descifrado*
- $K$  es el conjunto de *claves*

Un sistema *simétrico* usa la misma clave (*secreta*) para el cifrado y descifrado.

Algunos de los algoritmos más usados son [3-DES](#) y [American Encryption Standard \(AES\)](#).

AES soporta claves de 128, 192 y 256 bits y opera sobre bloques de 128 bits.

Estos algoritmos aplican varias etapas o *rondas* de operaciones de *sustitución* (reemplazo de bits por otros) y *transposición* (cambios de posición y desplazamientos) sobre los bits de datos y la clave (o valores derivados). Una función de *cifrado* de  $n$  rondas es de la forma:

$$E(P, K) = f_n(\dots f_2(f_1(P, K), K_2), K_n)$$

donde  $P$  es el dato de entrada o *texto plano* y  $C$  es el *texto cifrado* y cada  $f_i(p_i, k_i) \rightarrow c_i$ ,  $1 \leq i \leq n$ , es decir, genera un texto cifrado y toma una *subclave* ( $k_i$ ) generada desde el par  $(c_{i-1}, K)$  o  $(c_{i-1}, k_{i-1})$ .

Generalmente los algoritmos están diseñados para que al aplicar las rondas en orden inverso se obtiene la función inversa, es decir la de *descifrado*.

$$D(C, K) = f_1(\dots f_{n-1}(f_n(C, K), K_{n-1}), K_1))$$

Los algoritmos que operan sobre bloques tienen el problema que bloques idénticos generan la misma salida, habilitando técnicas de análisis diferencial y otros que permitirían inferir las claves. Por este motivo, generalmente se extienden para operar en forma dependiente del contexto, llamados *modos de operación*. Un modo comúnmente usado es *cipher block chain (CBC)* en el que a cada bloque de texto plano se aplica la operación *XOR* con el bloque cifrado previo. Al primer bloque se le aplica el *XOR* con un valor *aleatorio*, conocido como el *vector de inicialización*, el cual se transmite con el texto cifrado para que el receptor pueda aplicar el algoritmo de descifrado. Otro modo de operación usado, el *counter mode* se basa en que se agrega como entrada a la función de cifrado un *contador o nounce* (un valor usado por única vez).

Uno de los principales problemas con los sistemas de criptografía simétrica es que ambos participantes tienen que compartir la clave secreta, lo que genera el problema de distribución de claves. Una solución propuesta por W. Diffie y M. Hellman, conocido como el protocolo de *intercambio de claves Diffie-Hellman*, el cual permite calcular el mismo secreto a partir de ciertos datos públicos.

El protocolo *Diffie-Hellman* toma dos parámetros, un número primo  $p$  y un generador  $g$ , el cual debe ser una *raíz primitiva de  $p$* , es decir que para todo  $n$  entre 1 y  $p-1$  debe existir  $k$  tal que  $n = g^k \text{ mod } p$ .

El protocolo se consta de los siguientes pasos:

1. *Alice* y *Bob* acuerdan usar  $p = 5$  y  $g = 2$ .
2. *Alice* envía a *Bob*  $A = g^a \text{ mod } p$  donde  $a \in [1..p - 1]$  es un número aleatorio generado por *Alice*. Por ejemplo, con  $a = 3$ ,  $A = 2^3 \text{ mod } 5 = 3$ .
3. *Bob* hace lo propio, por ejemplo, generando  $b = 4$ , envía a *Alice*  $B = 2^4 \text{ mod } 5 = 1$ .
4. *Alice* computa  $s = B^a \text{ mod } 5 = 1$ .
5. *Bob* computa  $s = A^b \text{ mod } 5 = 1$ .

Así *Alice* y *Bob* han computado  $s=1$  como su *clave secreta*. La fortaleza del algoritmo se basa en que un atacante, quien conoce  $p$ ,  $g$ ,  $A$  y  $B$ , para calcular  $s$  debe conocer  $a$  o  $b$  para lo cual deberá calcular un **logaritmo discreto** para el cual no se conoce un algoritmo eficiente (problema *NP*).

Esto sentó las bases de la *criptografía asimétrica*. En *Diffie-Hellman* es posible ver el par  $(p,g)$  como la *clave pública* y  $a$  y  $b$  como las *claves privadas* (secretas) de *Alice* y *Bob*, respectivamente. Lo interesante de este esquema es que las claves secretas o privadas no necesitan distribuirse.

Se debe notar que este protocolo no autentica a las partes por lo que es vulnerable a un ataque *man-in-the-middle* (el atacante captura y reenvía los mensajes, haciéndose pasar por *Alice* y *Bob*).

Algunos de los algoritmos de cifrado/descifrado de criptografía asimétrica ampliamente usados son [RSA](#) y [El-Gamal](#).

#### RSA:

1. Seleccionar dos números primos. Sean  $p$  y  $q$ . Sea  $n = p \times q$
2. Obtener el *exponente*  $1 < e < \Theta(n)$ , *coprimo* a  $\Theta(n)$ , donde  $\Theta(n) = (p - 1) \times (q - 1)$
3. La clave pública es el par  $(n, e)$
4. Obtener  $d = (k \times \Theta(n) + 1)/e$
5. Función de cifrado con clave pública:  $E(d, (e, n)) = d^e \bmod n = c$
6. Función de descifrado (con la clave privada):  $D(c, (d, n)) = c^d \bmod n$

La seguridad de RSA se basa en que para obtener el secreto  $d$  es necesario conocer  $p$  y  $q$ , para lo cual hay que *factorizar*  $n$ , lo cual se considera un problema computacionalmente muy complejo para valores grandes.

Actualmente se requiere claves de al menos 2048 bits para ser lo suficiente robusta a ataques con el poder de cómputo actual, mientras que El-Gamal se basa al igual que Diffie-Hellman en el problema de los logaritmos discretos.

La criptografía asimétrica tiene las siguientes propiedades:

Si  $E(P, S_{sk}) = C$  entonces  $D(C, S_{pk}) = P$  y si  $E(P, S_{pk}) = C'$  entonces  $D(C', S_{sk}) = P$ , donde  $S_{pk}$  y  $S_{sk}$  son las claves públicas y privadas del sujeto  $S$ , respectivamente.

Estos sistemas criptográficos incluyen algoritmos para la generación de las claves públicas y privadas que aseguran sus relaciones matemáticas requeridas. La computación de cifrado y descifrado es mucho más costosa que los usados en criptografía simétrica por lo que generalmente son usados para autenticación y generación e intercambio de *claves de sesión* temporarias para luego usar algún algoritmo simétrico, los cuales son muy eficientes, para cifrar y descifrar mensajes.

## Hashes criptográficos

Un *hash criptográfico*, como **MD5** o **SHA**, es una función que oman una entrada de longitud arbitraria y genera una salida de longitud fija con las siguientes propiedades:

1. La función distribuye uniformemente sobre su imagen.
2. El algoritmo  $h(x)$  es eficiente (generalmente lineal).
3. Dado  $d$ , es computacionalmente impráctico encontrar  $x$  tal que  $h(x) = d$ .
4. Dados  $x$  y  $d$  es muy difícil encontrar  $y \neq x$  tal que  $h(y) = d$ .

Su uso fundamentalmente provee *integridad* ya que permite verificar que un mensaje u archivo no ha sido modificado. En comunicaciones, el *digest* se anexa al mensaje. El receptor puede verificar que el mensaje no contiene errores o ha sido modificado en su camino.

Otro uso común es su uso como *autenticador*. El *digest* se anexa *cifrado* por el emisor y el receptor puede verificar que ha sido enviado por el emisor desifrando el hash con la clave correspondiente y luego verificar la integridad del mensaje.

En algunos sistemas que permiten autenticación por *usuario/contraseña*, como por ejemplo el programa *login* de los sistemas tipo UNIX, se compara el hash de la contraseña ingresada por el usuario contra el hash almacenado en el archivo o base de datos de usuarios del sistema.

El *digest* de un mensaje, cifrado con la clave privada de un sujeto  $s$  constituye una *firma digital*.

**Verificación de la firma:**  $\text{hash}(\text{message}) = D(\text{Sig}_s, s_{pk})$ , donde  $\text{Sig}_s$  es la **firma** del sujeto  $s$  y  $s_{pk}$  es la clave pública de  $s$ . La firma digital garantiza:

1. *No repudio (de origen)*: El firmante no puede desconocer su firma
2. *Integridad* del mensaje: El mensaje no ha sido modificado luego de ser firmado.

Un *Message Authenticator Code (MAC)* usa una función de hash tradicional (no criptográfico) con una clave secreta conocida por las partes. El valor hash obtenido se anexa al mensaje. Una variante, conocida como *HMAC* aplica un hash criptográfico a la concatenación del mensaje y la clave secreta tal como se muestra en la siguiente figura.

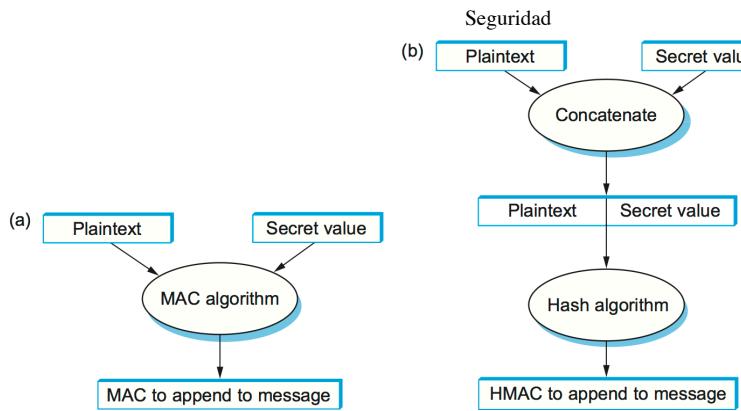


Figura 14.1: a) MAC b) HMAC.

## Distribución de claves

La distribución de claves secretas puede hacerse usando *Diffie-Hellman* o usando un *centro de distribución de claves*, una entidad de confianza como en *Kerberos*, descripto en la sección de autenticación.

Las claves públicas en principio podrían publicarse en el sitio web de cada persona u organización o enviarse por e-mail u otro servicio de mensajería. Un problema a resolver es que el sitio web podría haber sido vulnerado y el atacante la podría haber cambiado por su propia clave pública. Así podrá descifrar los mensajes con su clave privada. Lo mismo podría ocurrir si el atacante logra enviar su clave pública por email con el remitente falso.

El problema es que es necesario un mecanismo para asociar la verdadera *identidad* de un sujeto con su clave pública. Generalmente se utilizan dos enfoques: Confiar en terceros usando una *infraestructura de claves públicas (PKI)* o en base a otro concepto como la *web de confianza (web of trust)*.

Una PKI se basa en una jerarquía de *autoridades de certificación (CA)*. Una CA es una organización que se encarga de brindar un servicio de asociar la *identidad de un sujeto* con su *clave pública*. Esta asociación se plasma en un *certificado*, firmado por la CA.

La clave pública de una CA está en un certificado de una CA de mayor nivel. Una CA *raíz* su certificado conteniendo su identidad y su clave pública está *auto firmado*.

Un usuario puede generar sus pares de claves y genera un *certificate signing request (CSR)*, el cual se envía a una CA para que lo firme. La CA deberá verificar la identidad del sujeto, generalmente solicitando información por otro medio y luego le devolverá el certificado firmado. Uno de los formatos usados para los certificados es el estándar X.509.

En una comunicación segura, el emisor usa el certificado del receptor, lo valida (verificando la autenticidad de la firma y su integridad) usando el certificado (el cual contiene la clave pública) de la CA firmante y transmite cifrando con la clave pública del receptor. El receptor puede descifrar el mensaje usando su clave privada.

El paquete de software [openssl](#) incluye una gran variedad de algoritmos criptográficos y las utilidades necesarias para la gestión de certificados para desarrollar una CA.

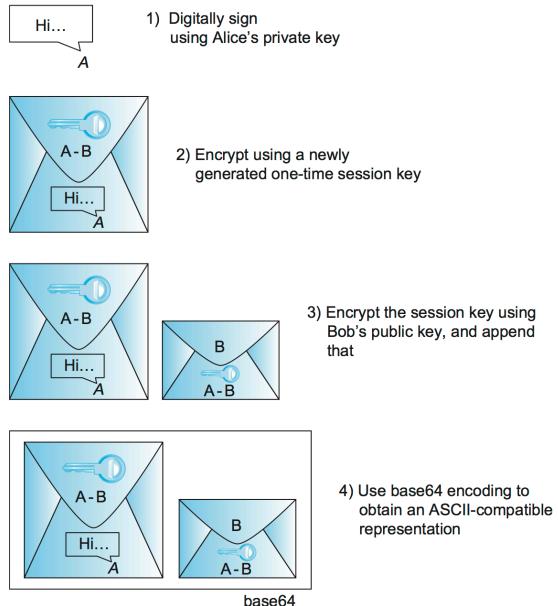
En [Pretty Good Privacy\(PGP\)](#), usado principalmente para cifrado de mensajes por email se basa en que los usuarios deberían confiar en las claves públicas de otros en principio por relaciones directas. Esto se conoce como la *web of trust*.

Un certificado PGP es auto-firmado y puede contener firmas de otros usuarios que pueden acreditar su autenticidad, aumentando su confiabilidad. Esto permite un sistema descentralizado de confianza.

Generalmente cada usuario distribuye su clave pública con la forma de una *huella o fingerprint* (hash) del certificado. El software PGP (ver [OpenPGP](#)) permite generar claves, firmar certificados y almacenar y buscar claves públicas de usuarios en servidores públicos.

La siguiente figura muestra los pasos en el cifrado de un mensaje usando PGP.

Hi...=The plaintext message

**Figura 14.2: Preparación de un mensaje usando PGP.**

## Autenticación

En una comunicación se debe asegurar que cada mensaje es auténtico, es decir que realmente fue enviado por alguna de las partes en esa sesión. La adición de un autenticador en cada mensaje y cifrarlo no siempre alcanza.

Un *reply attack* consiste en que el adversario transmite una copia de un mensaje anterior, el cual para el receptor podría ser un mensaje auténtico. Para prevenir este ataque se necesita un mecanismo que garantice la *originalidad* de mensajes. Si el adversario retiene un mensaje y lo reenvía más tarde (*suppress-reply attack*) genera un mensaje fuera de orden (*no timeliness*).

Estas dos propiedades caracterizan *integridad*. Además es necesario establecer una *clave de sesión* para poder usar criptografía simétrica.

Una técnica para resolver los problemas de originalidad y secuencialidad es incluir en los mensajes un *timestamp*, lo que requiere que las partes tengan sus relojes sincronizados.

Esos *timestamps* actúan como *nounces*: valores aleatorios usados una única vez, lo que requiere que el receptor almacene los nounces recibidos para verificar que el recibido recientemente no ha sido usado previamente.

Es común usar una combinación de ambas técnicas: pseudo-timestamps y nounces para la sesión, únicamente.

Un protocolo simple para lograr *timeliness* y autenticación es un protocolo del tipo *challenge-response* como se muestra en la siguiente figura.

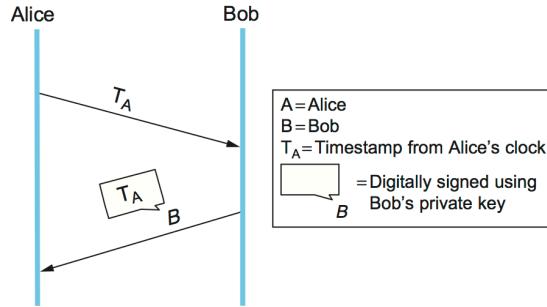


Figura 14.3: Protocolo challenge-response.

Alice envía su timestamp y Bob responde con el timestamp recibido con su firma anexada. Así Alice puede determinar que Bob es realmente quien responde al mensaje original.

Una alternativa es que Bob retorne una respuesta incluyendo el timestamp de Alice cifrada con su clave privada en lugar de usar una firma.

## Autenticación con clave pública/privada

La siguiente figura muestra un protocolo de autenticación usando criptografía asimétrica.

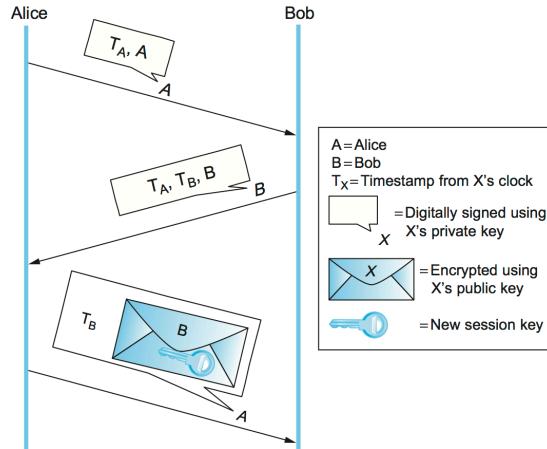


Figura 14.4: Autenticación con clave pública/privada.

Alice envía su identidad y su *timestamp* firmado. Bob responde con el timestamp de Alice, su propio timestamp y su identidad, con su firma anexada. Finalmente, Alice responde con el timestamp de Bob (en forma plana) y la clave secreta de sesión generada cifrada con la clave pública de Bob, así éste la podrá descifrar y ambos la pueden usar para cifrar los siguientes mensajes.

Se debe notar que los timestamps son como nounces y los relojes no necesitan estar sincronizados.

## Autenticación con cifrado simétrico

En la siguiente figura se muestra el protocolo de autenticación propuesto por Needham y Schroeder.

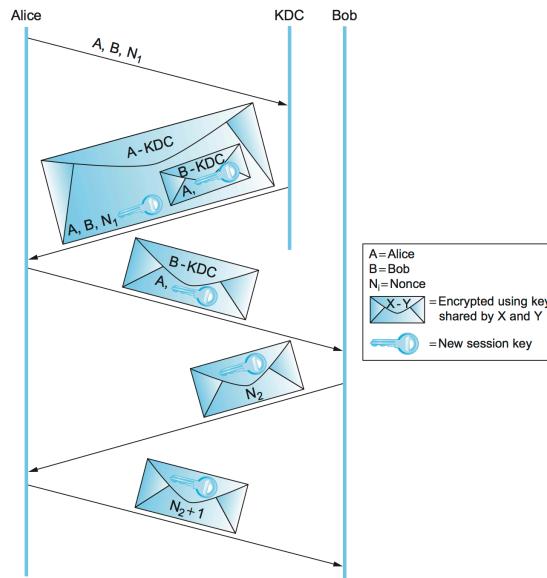


Figura 14.5: Protocolo de Needham-Schroeder.

Se basa en el uso de una *key distribution center* (*KDC*) el cual contiene las claves secretas compatidas por él y cada participante. A continuación se describen los pasos del protocolo.

1.  $A \rightarrow KDC : A, B, N_1$

A envía al KDC un mensaje con las identidades A y B junto con un *nounce*.

2.  $KDC \rightarrow A : \{N_1, A, K_{AB}, \{K_{AB}, A\}K_{B-KDC}\}K_{A-KDC}$

El KDC responde con el nonce, una nueva clave de sesión entre A y B ( $K_{AB}$ ) y la clave de sesión con la identidad de A cifrada con la clave secreta de Bob. Todo el mensaje se cifra con la clave secreta compartida entre A y el KDC.

3.  $A \rightarrow B : \{K_{AB}, A\}K_{B-KDC}$

Alice envía el par recibido (cifrado con la clave secreta de Bob) a Bob. Notar que Alice no puede descifrar este mensaje.

4.  $B \rightarrow A : \{N_2\}K_{AB}$

Bob envía a Alice un nuevo nounce (challenge) a Alice.

$$5. A \rightarrow B : \{N_2 + 1\}K_{AB}$$

Alice responde al reto aplicando una función (+1) al nounce recibido.

Cabe analizar que el protocolo es vulnerable a un *reply attack*. Si el atacante usa un viejo mensaje comprometido del paso 2, podría enviarle  $\{K_{AB}, A\}K_{B-KDC}$  a Bob y éste lo reconocerá como un mensaje válido proveniente de Alice.

## Kerberos

El protocolo Kerberos se basa en la idea del protocolo anterior. En el paso 2 el KDC envía a Alice una especie de certificado o *token*, el cual consiste en una clave de sesión y la identidad de Alice. Kerberos separa esta funcionalidad en un *authentication server (AS)* que sólo da acceso a un *ticket granting server (TGS)* el cual otorga un *token* a Alice para presentar a Bob, como se muestra en la siguiente figura.

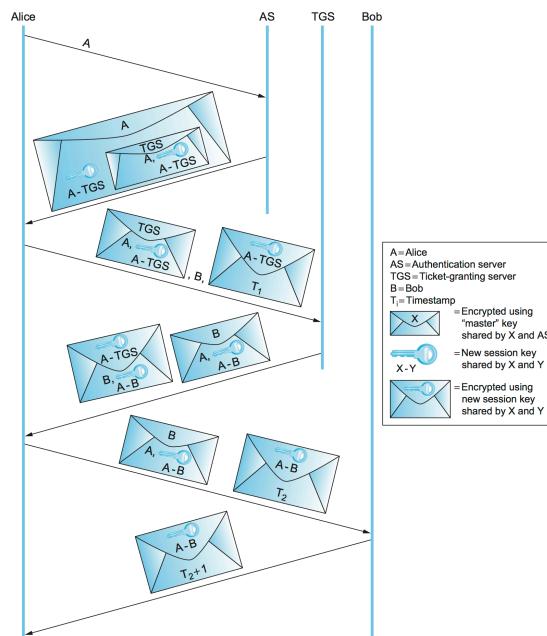


Figura 14.6: Kerberos.

La ventaja de este esquema es que si Alice debe contacta a otro host, sólo debe obtener un nuevo token del TGS, sin necesidad de re-autenticarse con el AS.

## OAuth y OpenId

Actualmente muchas aplicaciones (generalmente web) requieren acceder a recursos del usuario, por ejemplo, su foto y datos de su perfil de su cuenta de Google, Facebook u otra aplicación. En estas aplicaciones se presenta al usuario una ventana donde tiene la posibilidad de seleccionar la cuenta (Google, Twitter, etc) a la que quiere permitir el acceso y debe *consentir* que dará acceso a los recursos que se le solicitan.

Actualmente muchas aplicaciones usan este escenario simplemente para el *login* del usuario usando algunas de sus cuentas existentes en aplicaciones de uso común como las mencionadas para no tener que crear nuevas cuentas y recordar su usuario y contraseña.

El protocolo **OAuth** fue desarrollado para *autorizar* a una aplicación cliente el acceso a recursos adueñados por un usuario residiendo en otra aplicación. Los recursos son generalmente APIs que permiten acceder a recursos como datos del perfil del usuario o para la obtención y envío de mensajes en una red social u otras aplicaciones.

OAuth se basa en HTTP y se usa ampliamente en aplicaciones Web, móviles y nativas. En la siguiente figura se muestra el escenario correspondiente.

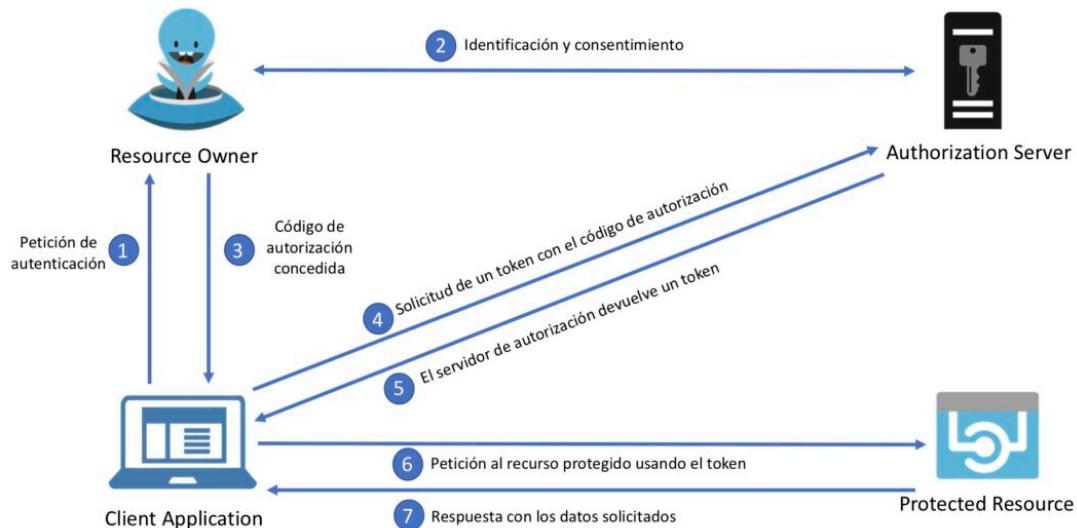


Figura 14.6: Autorización usando OAuth.

1. La aplicación solicita al usuario que se autentique. En ese punto se le presentará un diálogo para ingresar su usuario y contraseña o usando alguna de sus cuentas de Google, GitHub, Facebook, Twitter, etc.
2. Si elige una de las cuentas (por ejemplo Google), la aplicación envía un requerimiento con los recursos requeridos y una URL de redirección, al *servidor de autorización* OAuth de Google el cual pide el consentimiento al usuario sobre qué recursos la aplicación original está solicitando.

3. Luego que el usuario dió su consentimiento, se le redirige a la página dada por la aplicación pasándole un *código*.
  4. La aplicación solicita un *token de acceso* al *servidor de autorización* pasándole el código recibido en el paso anterior.
  5. El servidor responde con el *token* luego de verificar el código recibido.
  6. La aplicación puede acceder a la API del servicio usando el token recibido.
- Los tokens se representan como [JSON Web Tokens \(JWT\)](#).

Para poder usar OAuth el desarrollador de una aplicación debe registrarla en aquellos servidores OAuth de los servicios a los que quiere acceder. Por ejemplo, en la página [Google OAuth 2.0 sign-in](#) se describen los pasos y las APIs provistas por Google para desarrolladores que quieran integrar en su aplicación este servicio.

Una vez registrada la aplicación se obtiene un *client ID* y un *client secret*. Estos valores deben pasarse al servidor de autorización para obtener el token de acceso.

Como se puede apreciar, OAuth es un protocolo de *autorización*, aunque muchas aplicaciones lo usan para *autenticación*.

[OpenID Connect](#) es un protocolo que opera sobre OAuth específicamente para autenticación, es decir sólo con el objetivo de permitir acceso a usuarios con cuentas registradas en otras aplicaciones.

## Protocolos seguros

En esta sección se describen algunos protocolos y aplicaciones seguras.

### SSH

Secure shell (ssh) provee un servicio de acceso (login) remoto y transferencia de archivos con servicios de autenticación y canales seguros.

La autenticación entre hosts se hace usando criptografía de clave pública/privada. Cada host tiene su par de claves que se generan en el momento de instalación del software. Una vez autenticado los hosts, se autentica el usuario (por password o su propio par de claves pública/privada). Finalmente se acuerda una clave de sesión, comúnmente usando Diffie-Hellman, para cifrar el tráfico usando cifrado simétrico (usualmente AES).

Consiste de tres protocolos:

1. **SSH-TRANS (RFC 4344)**: Provee un canal seguro (cifrado) sobre TCP usando un cifrado simétrico (generalmente AES) usando una clave de sesión obtenida luego que el cliente autentica al servidor usando comúnmente RSA. Puede haber una doble autenticación, donde el server autentica al cliente. Cada host SSH contiene su par de claves pública/privada.
2. **SSH-AUTH (RFC 4252)**: Un protocolo de autenticación de usuarios. Puede ser basado en usuario/contraseña, usando claves pública/privada o *host based*.
3. **SSH-CONN (RFC 4254)**: Permite correr otras aplicaciones no seguras como SMTP o X11 sobre un canal (túnel) seguro. Es común su uso para realizar una sesión gráfica remota (ver [X app over ssh](#)).

Una implementación como [openSSH](#) ofrece comandos para gestionar claves (`ssh_add`, `ssh-keygen`, ...), realizar operaciones remotas como login remoto y ejecución remota de comandos (`ssh user@server [cmd]`) y transferencia archivos (`scp file user@server:path`).

En el servidor se ejecutan los servicios (daemons) `sshd`, `sftp-server` y `ssh-agent` (autenticación).

Para usar autenticación de usuario usando criptografía de claves pública/privada se debe agregar (*append*) la clave pública del usuario en el host cliente en el archivo `$HOME/.ssh/authorized_keys` de la cuenta de usuario correspondiente en el servidor. Se puede hacer con el script `ssh_copy_id`. Luego el usuario será autenticado sin solicitar ningún password.

## TLS/SSL y HTTPS

*Transport Layer for Security (TLS)* es la evolución de *Secure Sockets Layer (SSL)*, este último desarrollado por Netscape communications. Es un protocolo que opera sobre TCP y provee servicios de transporte seguro. El protocolo HTTP usando SSL/TLS se conoce como HTTPS (puerto 443).

El protocolo se basa en autenticación basada en certificados y usa hashes y HMACs para integridad de datos, establecimiento de claves de sesión usando Diffie-Hellman o RSA para luego usar 3-DES o AES para el cifrado de mensajes.

La siguiente figura muestra el *handshake* de alto nivel entre el cliente y servidor.

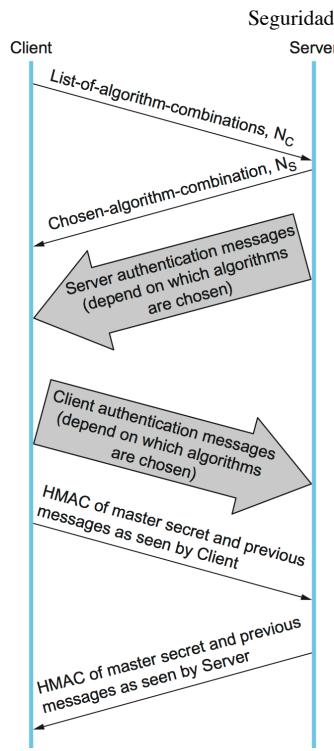


Figura 14.4: Handshake TLS.

El cliente y servidor *acuerdan* los algoritmos de hash, HMAC y cifrado a usar. Cada mensaje inicial incluye el *nounce* (valor aleatorio usado una única vez).

Posteriormente, el server envía al cliente su certificado para que el cliente valide. El server a su vez puede indicarle al cliente que requiere autenticarlo. En este caso el cliente envía su certificado.

A partir de la *PMS*, y los *nounces* se genera la *master secret* y de esta las claves de sesión. Finalmente, el cliente envía un mensaje que incluye un hash de todos los mensajes anteriores. El server responde similarmente.

Cuando el handshake finaliza, la comunicación se basa en el *record protocol*, el cual provee confidencialidad e integridad al protocolo de transporte subyacente.

Cada mensaje entregado por la aplicación se fragmenta en bloques, los cuales se comprimen, se les agrega un HMAC como un autenticador y se cifran usando un cifrado simétrico como AES.

Las nuevas versiones de TLS soportan *re-establecimiento de sesión*, donde el cliente envía en su mensaje inicial un *identificador de sesión* previamente establecido. Si el servidor lo reconoce, utilizan los parámetros negociados previamente.

En HTTPS los navegadores web incluyen una base de datos de los certificados de las CAs más populares o reconocidas. Así la validación de un certificado recibido de un servidor no requiere que las CAs estén en línea.

## IPSec

Este protocolo ofrece seguridad a nivel de capa de red. Es opcional en IPv4 pero de uso obligatorio en IPv6. Es modular y muy flexible.

Es posible analizar IPSec en sus dos grupos de protocolos. En el primer grupo hay dos protocolos para brindar servicios de control de acceso, integridad de mensajes, autenticación y *forward secrecy* (resistencia a descifrar mensajes anteriores al descubrir una clave) y se conocen como *Authenticacion Header (AH)* y *Encapsulating Security Payload (ESP)*. El segundo grupo permite la gestión e intercambio de claves conocido como *Internet Security Association and Key Management Protocol (ISAKMP)*.

Si bien IP es un protocolo sin conexión, IPSec crea una *asociación de seguridad (SA)* entre las partes usando ISAKMP. Una SA es unidireccional. Así, por ejemplo, para una comunicación TCP se requieren dos SAs. A cada SA recibida se asigna un *índice de parámetros de seguridad (SPI)* (índice en una tabla que describe los algoritmos a usar). Una SA se identifica por el par *(SPI, destination-address)*. El ESP header incluye la SPI así el receptor puede determinar la SA a usar para descifrar el paquete, como se muestra en la siguiente figura.

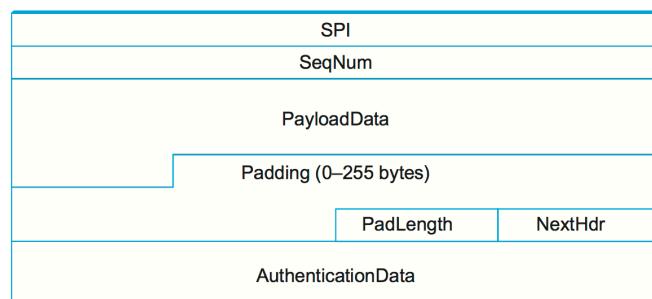


Figura 14.4: Formato de la cabecera ESP.

La cabecera incluye un número de secuencia de los paquetes para prevenir *ataques de réplicas (reply attacks)*. En IPv4 el ESP header sigue a la cabecera IP. En IPv6 es un extension header.

IPSec puede usarse en *transport mode* el ESP payload contiene los datos de la capa superior (UDP, TCP o aplicación). En modo *tunnel* el ESP payload es directamente otro paquete IP permitiendo implementar una VPN.

## Seguridad en Wifi

Actualmente es común el uso de *access points (APs)* que utilizan *Wifi Protected Access (WPA)* (802.11i). Hay dos modos de autenticación. El primero se basa en que todos los clientes comparten una *pre shared key (PSK)* generada a partir de una *contraseña* definida en el AP. Cada cliente deriva su clave de sesión a partir de la PSK y otros parámetros como su *MAC address*.

El otro modo mas fuerte se basa en el uso de un *authentication server*, conectado en forma segura al (o corriendo en el mismo) AP. Esto permite definir una clave por cada cliente.

En WPA2 la clave de sesión generada se usa como clave de AES en modo *counter* para cifrar los paquetes y una MAC como autenticador.

## Firewalls, IDSs y VPNs

Un *cortafuegos* usa una política basada en entrada-salida, es decir actúa como un *filtro de paquetes* para prevenir tráfico no deseado.

Generalmente se basan en *reglas* las cuales tiene dos partes: EL *matching pattern* y la *acción* a ejecutar en cada paquete coincidente.

Las acciones pueden ser *Drop*, *Accept*, *Reject* u operaciones de *NAT* o operaciones de *marcado de paquetes (mark)*. Las operaciones de *NAT* pueden usarse para hacer *maskerading*, *port forwarding*, *balance de carga* u otras operaciones como el encolado en *buckets* de diferentes prioridades para implementar políticas de *calidad de servicio (QoS)*.

Los patrones permiten reconocer paquetes por medio de las *cabeceras* de los mensajes ya que generalmente se aplican internamente en el procesamiento de los paquetes en su arribo interiormente en el kernel del sistema operativo.

Un ejemplo de un firewall es el módulo [iptables](#) de GNU-Linux. El siguiente listado muestra un conjunto de reglas para *filtrar* paquetes en un router GNU-Linux hogareño.

```
#!/bin/sh
# reject all initially
iptables -A INPUT DROP

# accept all on internal interface
iptables -A INPUT -i eth1 -j ACCEPT
```

```

# accept tcp connections to port 80 on external interface
iptables -A INPUT -i eth0 -p tcp -m tcp --dport 80 -m state --st

# accept packets on established TCP connections
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

# forward packets comming from external interface
iptables -A FORWARD -i eth0 -j ACCEPT

# accept output traffic
iptables -A OUTPUT -j ACCEPT

# NAT: all incoming tcp to port 80 redirect to internal web serv
iptables -A PREROUTING -i eth0 -p tcp --dport 80 -j DNAT --to-de

# masquerade all outgoing traffic
iptables -A POSTROUTING -o eth0 -j MASQUERADE

```

#### **Listado 14.1: Ejemplo de reglas de *iptables*.**

Un firewall o *filtro de aplicación* contiene reglas de filtrado de mensajes para aplicaciones o protocolos específicos. Es común usar filtros de aplicaciones web de un servidor web o de aplicaciones que impida el paso de mensajes que no cumplan con una API HTTP determinada. También es posible filtrar mensajes de salida HTTP para prevenir que los usuarios de una organización no accedan a sitios específicos. Comúnmente se basan en el uso de expresiones regulares para hacer matching con un conjunto de URLs. Otro ejemplo son las aplicaciones de control parental a menores de edad en una red hogareña.

Un *Intrusion Detection System (IDS)* es un servicio generalmente corriendo en modo usuario que permite identificar tráfico no deseado principalmente analizando el *payload* de los paquetes. Son útiles para analizar ataques conocidos identificando sus patrones o firmas, de manera similar a los antivirus o antispam de mail.

Otros mecanismos usados en IDS es la detección de tráfico anómalo o no habitual, generalmente usando técnicas de *machine learning*.

Un IDS libre ampliamente usado es **SNORT**.

Una *Virtual Private Network (VPN)* permite conectar hosts de diferentes subredes físicas interconectadas por una red pública como Internet con el fin de simular que están en la misma red lógica.

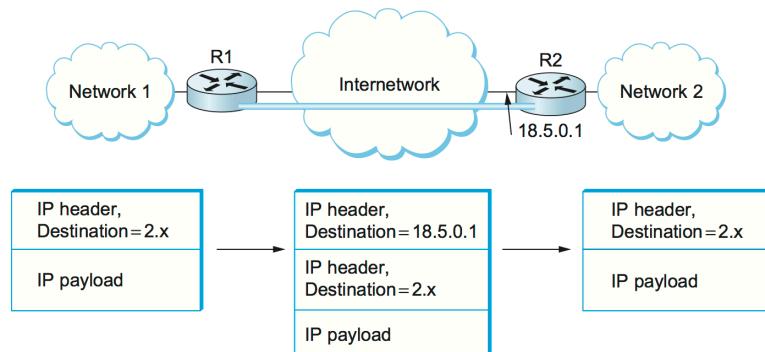
Un software de *vpn* consiste de al menos un programa cliente instalado en cada estación de trabajo y un servidor, instalado comúnmente detrás de un router en la red remota donde están los servicios o recursos a alcanzar.

La siguiente figura muestra un esquema de una *vpn*. Un host en *network 1* desea enviar un mensaje a un host de *network 2*. Ambas redes comúnmente usan direcciones *privadas*.

El programa cliente de la *vpn* intercepta el paquete (en el host o en R1) y lo *encapsula* (forma un *túnel*) en un paquete con dirección de destino la IP pública del router R2. El router R2 ejecuta (o reenvía el paquete recibido a) el servidor *vpn* el cual *extrae* el paquete encapsulado y lo redirige al host de destino.

Para realizar este proceso comúnmente se requieren configurar reglas de firewall o NAT en los routers (R1 y R2).

Para garantizar autenticación y confidencialidad comúnmente se usa IPSec o un protocolo de transporte como UDP o TCP sobre TLS, como en [OpenVpn](#).



**Figura 14.5: Esquema de un túnel VPN.**

Un servicio de *vpn proxy* ofrece el servicio de *relay* de paquetes entre los extremos de la comunicación (*proxy*). Esto provee *anonimato* del cliente ya que el servidor *vpn* es el que realmente interactúa con el servidor, reenviando los mensajes entre ambos extremos como se muestra en la siguiente figura.

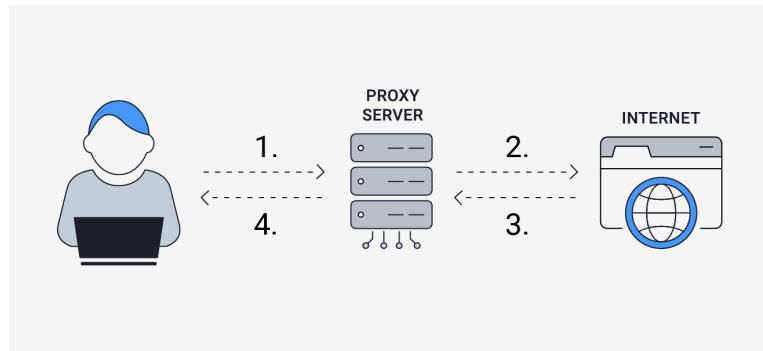


Figura 14.6: VPN Proxy service.

## Servicios de Proxy (o relay)

El uso de proxies permite interceptar mensajes de clientes y realizar redirecciones o filtrado y pueden usarse con diferentes fines como puentes (conversores de protocolos), seguridad o balance de carga.

Es posible usar dos configuraciones (ver las figuras de abajo):

- *Forward proxy*: Se sitúa en frente de un grupo de hosts o clientes.
- *Reverse proxy*: Se sitúa en frente de servidores.

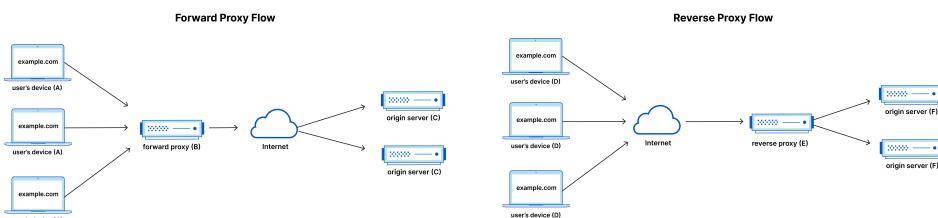


Figura 14.7: Forward y reverse proxies.

< Anterior

## Aplicaciones

# Telecomunicaciones y Sistemas Distribuidos

Estas son las notas de la segunda parte del curso "*Telecomunicaciones y Sistemas Distribuidos*" de la Licenciatura, dictado por el Departamento de Computación - FCEFQyN, de la Universidad Nacional de Río Cuarto.

En estas notas se abordan temas de *sistemas distribuidos*, es decir, algoritmos y protocolos para implementar sistemas sin una coordinación central, permitiendo lograr sistemas escalables y tolerantes a fallas.

Las notas de la primera parte del curso, incluyen temas de redes de computadoras.

## Equipo docente

- Marcelo Arroyo
- Gastón Scilingo

## Bibliografía recomendada

1. *Distributed Systems*. 4th edition. Maarten Van Steen and Andrew S. Tanenbaum. 2013. URL: <https://www.distributed-systems.net/>. Puede descargar su copia libre personalizada.
2. *Distributed Systems*. Course notes of University of Cambridge 2021/22. URL: <https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/dist-sys-notes.pdf>

# Sistemas distribuidos

Actualmente existe la necesidad de desarrollar sistemas que brinden servicios a escala global. Algunos ejemplos de este tipo de aplicaciones son Google Docs, redes sociales (Facebook, Instagram, ...). Otras organizaciones proveen acceso a servicios de infraestructura de computación y almacenamiento de datos, como Amazon Web Services (AWS), Google cloud y otros.

Estas aplicaciones requieren de alta *disponibilidad*, *escalabilidad* y de fácil *mantenibilidad* o evolución.

El requerimiento de *disponibilidad* implica que no es posible que esos servicios o aplicaciones tengan un *único punto de acceso* como en el caso de una aplicación web tradicional ya que ante una caída de una parte de la red podría dejar a millones de usuarios sin acceso. Además es punto de acceso podría requerir un ancho de banda prácticamente imposible de lograr.

Los requerimientos de *escalabilidad* para satisfacer a millones de clientes interactuando concurrentemente con el sistema presenta no sólo desde el punto de vista del poder de cómputo necesario, sino también de las capacidades de la red interna para soportar una gigantesca cantidad de tráfico y con una bajísima latencia para alcanzar tiempos de respuestas razonables.

Un sistema de estas características debe ser fácilmente *mantenible* y configurable ya que no puede soportar interrupciones por tareas de mantenimiento y/o reconfiguración. En lo posible estos sistemas deberían ser auto-configurables.

El diseño de estos sistemas prácticamente obliga a abandonar la idea de tener control, estado y procesamiento centralizado como en una arquitectura básica cliente-servidor y se debe ir hacia un diseño de un *sistema distribuido*.

Un *sistema distribuido* es una colección de elementos de computación (nodos) interconectados que cooperan para brindar un conjunto de servicios formando un sistema que se observa como único y coherente.

Es importante para que un sistema califique como distribuido que tenga la *transparencia* necesaria para que para los usuarios aparezca como un sistema ejecutándose en una única computadora o como si fuera en apariencia un sistema cliente-servidor tradicional.

Esto significa que los recursos que gestiona deben ser accesibles por medio de un sistema de identificación (*naming*) que sea independiente de su ubicación física.

En las notas del curso de redes y telecomunicaciones se analizan algunos ejemplos de sistemas distribuidos. Uno de ellos es el sistema DNS, el cual brinda el servicio de resolución de nombres a sus respectivas direcciones IP, formando un conjunto de nodos que *particionan* la base de datos de dominios de manera jerárquica. Otro ejemplo que puede mencionarse dentro de los protocolos de infraestructura de Internet son el sistema de ruteo que puede verse como un sistema que contiene las rutas (utilizando algoritmos de descubrimiento de nuevas rutas) lo que constituye una gran base de datos distribuida.

Un sistema distribuido generalmente organiza como una *overlay network*, es decir, una red *lógica* o *virtual* sobre una red actual, típicamente Internet. Algunas veces el sistema define una *topología virtual* de esta red en base a los algoritmos utilizados. Mas adelante se verá que algunas aplicaciones se basan en topologías virtuales como *anillos*, *árboles* o *grillas*.

Para lograr alta *disponibilidad* generalmente los recursos se deben *replicar* para que en el caso de fallas de algunos nodos, el resto del sistema pueda seguir funcionando.

Para lograr *escalar*, es decir que el sistema soporte inmensos volúmenes de datos o pueda ejecutar cientos de miles de operaciones por segundo es necesario *particionar* los datos y/o computaciones. Esto se conoce como *escalado horizontal*, es decir, agregar elementos (computadoras) con aplicaciones que operen en forma cooperativa usando un modelo de *pasaje de mensajes*.

Agregar capacidad de cómputo (memoria, CPUs, interfaces de red, etc) a una computadora existente se denomina *escalabilidad vertical* lográndose una escalabilidad limitada.

El particionado genera problemas de *disponibilidad* ya que si un nodo que contenía ciertos valores se cae, éstos datos se tornan inaccesibles. Una solución al problema de disponibilidad y tolerancia a fallas es la *replicación* de los recursos del sistema.

Si los recursos replicados pueden evolucionar, es decir son objetos mutables, surge el problema de mantener su *coherencia* en las réplicas.

En estas notas se analizan los principales problemas a resolver en los sistemas distribuidos y sus alternativas de solución.

## Tipos de sistemas distribuidos

Hay diferentes tipos de sistemas distribuidos para diferentes propósitos. Cuando el objetivo es lograr *computación de alto desempeño* (*high performance computing - HPC*) es posible clasificar en dos tipos de sistemas:

1. **Clusters:** Red de computadoras interconectadas por enlaces de alta velocidad como por ejemplo Gigabyte Ethernet. Generalmente cada nodo corre el mismo sistema operativo como GNU-Linux. El uso de clusters se conoce como *computación paralela*.

Un ejemplo de software utilizado en clusters es [Beowulf](#).

Se utiliza un *planificador de tareas (jobs)* en base a sus requerimientos de recursos, tiempo estimado de computación y otros parámetros. Los usuarios *encolan* sus *jobs* y luego el sistema los notifica con los resultados obtenidos generalmente por correo electrónico.

2. **Grid computing:** Sistemas que comprenden un conjunto de computadoras generalmente de alto rendimiento interconectados que se ofrecen al usuario como un gran supercomputador. La principal diferencia con la computación paralela en clusters es que los nodos pueden ser heterogéneos tanto en hardware como en software y generalmente están dispersos geográficamente e inclusive pertenecer a diferentes propietarios y/o administrados por diferentes dominios.

El término *computación en la nube* o *cloud computing* se aplica a ofrecer los recursos de hardware (generalmente virtualizados o contenerizados) y servicios de computación (software) en Internet. Ejemplos de estos servicios son los provistos por [Amazon We Services \(AWS\)](#) y [Google cloud](#).

Generalmente una *nube* se diseña en capas, así una de las capas mas bajas ofrece los servicios de recursos de computación (computadoras reales o máquinas virtuales) y almacenamiento. Ejemplos de servicios en este nivel son como [Amazon EC2](#) y [Amazon S3](#). El siguiente nivel ofrece *frameworks de software* y APIs para el desarrollo de aplicaciones distribuidas, como [MS Azure](#) y [Google Apps](#).

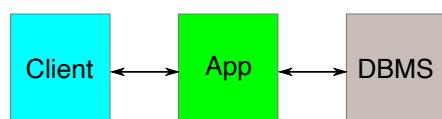
Otro tipo de sistemas distribuidos son los *sistemas de información* que generalmente tienen los datos distribuidos y posiblemente replicados. Estos sistemas generalmente ejecutan *transacciones sobre bases de datos distribuidas*.

Finalmente cabe nombrar los sistemas conocidos como *pervasive (penetrantes) systems* en los cuales intervienen dispositivos móviles formando sistemas con poca *estabilidad* en cuanto a que sus nodos aparecen (se unen), desaparecen y cambian su ubicación geográfica. Algunos ejemplos de estos tipos de sistemas son las redes de telefonía celular, redes de sensores y lo que se conoce como *Internet de las cosas*.

## Arquitecturas

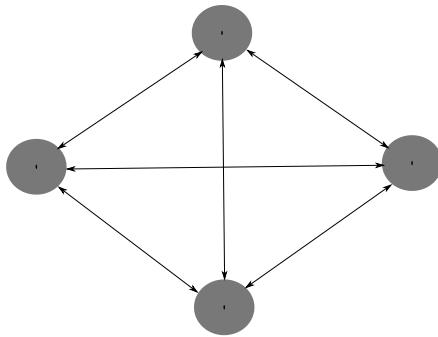
Para que un sistema califique como un sistema distribuido, tolerante a fallas y con un desempeño aceptable, generalmente no es posible un diseño basado en una arquitectura tradicional del tipo cliente-servidor ya que tiene un único punto de falla y congestionamiento de computación y/o comunicaciones.

Una extensión inmediata es la arquitectura *multitier* o *multicapa*.



**Figura 1.1: Arquitectura multitier.**

La arquitectura que permite la mejor escalabilidad y tolerancia a fallas son aquellas basadas en una estructura descentralizada. En una red *peer to peer (P2P)*, cada nodo ejecuta algoritmos distribuidos generalmente sin un coordinador central fijo.

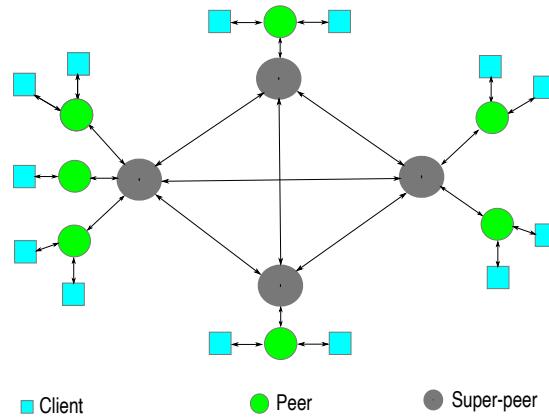


**Figura 1.2: Arquitectura peer-to-peer.**

Algunos problemas pueden solucionarse de manera más eficiente o simple utilizando un coordinador central, pero éste constituye un único punto de falla por lo que es necesario seleccionarlo de manera colaborativa entre los peers. Por eso analizaremos los algoritmos de *elección* generalmente usados para seleccionar un coordinador o líder temporal.

Un problema a resolver en una red peer-to-peer es cómo cada nodo conoce a los demás. Generalmente esto se resuelve iniciando el sistema con un conjunto base de nodos los cuales conocen de su existencia o usando un conjunto de servidores *conocidos* al que pueden consultar por la existencia de los demás. En este último caso cada peer tiene que *suscribirse* en uno de estos servidores.

Una arquitectura muy usada es de varios niveles (o capas).



**Figura 1.3: Arquitectura peer-to-peer de dos niveles.**

En esta arquitectura, un peer se debe conectar a un *super peer*, generalmente seleccionado por proximidad o de menor costo (utilización, ancho de banda, etc). Los super-peers forman la *overlay network* conformando el sistema distribuido.

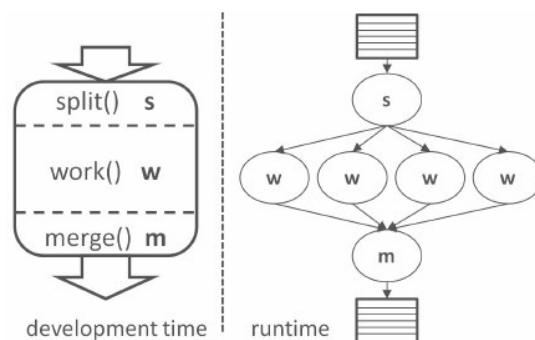
Varias aplicaciones distribuidas, como por ejemplo Skype, se basan en esta arquitectura. Un cliente toma la forma de una aplicación de usuario o una aplicación web conectada a un *peer*. Un super-peer puede ser un simple nodo o un cluster de computadoras (*data center*) operando como una gran computadora paralela.

Un *peer* (también conocido como *edge point*) puede ofrecer servicios de *caching* de contenidos formando parte de la red de entrega de contenidos o *Content Delivery Contents (CDN)*, permitiendo entregar contenidos en base a la proximidad del usuario, como por ejemplo videos de YouTube (Google Cloud) o Netflix. En el caso de Google Cloud es posible ver a los *puntos de presencia* como los *peers* y los *data centers* como los *super peers*.

Otras arquitecturas comúnmente usadas en aplicaciones distribuidos, principalmente en nodos con grandes requerimientos de procesamiento son:

## Workers Farm

Esta arquitectura permite distribuir los requerimientos entre un conjunto de *workers*. El distribuidor o *splitter* (*s*) generalmente actúa como *balanceador de carga* y determina la mejor estrategia de *splitting*, generalmente dividiendo la tarea en sub-tareas independientes. Esto generalmente requiere algún análisis de *dependencia de datos*. El nodo *merge* recolecta las soluciones parciales y las *combina* para entregar la solución final.



**Figura 1.4: Arquitectura Workers-Farm.**

## Map-Reduce

El patrón map-reduce se popularizó con su aplicación por Google en su motor de búsqueda y otras aplicaciones.

Se basa en las siguientes funciones:

- $\text{map}(k_1, v_1) \rightarrow [(k_2, v_2)]$ : Toma un par  $(key, value)$  de tipo  $(k : k_1, v : k_2)$  y genera una lista de pares de tipo  $(k_2, v_2)$
- $\text{reduce}(k_2, [v_2]) \rightarrow [(k_3, v_3)]$ : Toma una clave de tipo  $k_2$  y una lista de valores de tipo  $v_2$  y genera una lista de pares de tipo  $(k_3, v_3)$ .

La salida de cada *mapper* se *combina* y *particiona* (*shuffled*) formando listas de valores asociadas a cada *key* de tipo  $k_2$  para asociar a cada *reducer*, como se muestra en la siguiente figura.

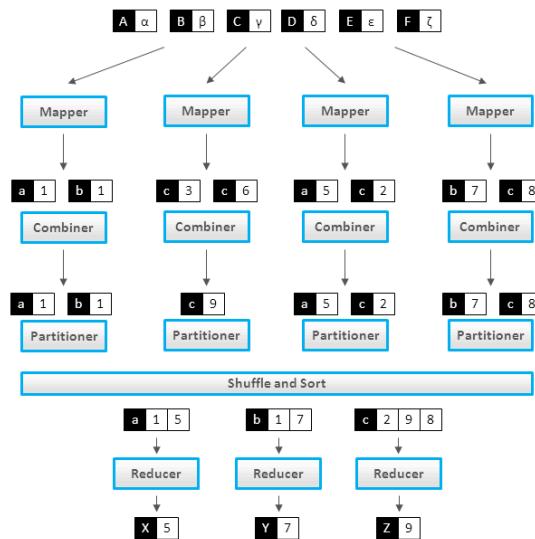


Figura 1.5: Arquitectura Map-Reduce.

Frameworks como el de Google Map-Reduce (descripto en [GoogleMapReduce](#)) o [Apache Hadoop](#) ofrecen una API para el desarrollo de aplicaciones distribuidas basadas en este patrón.

Estas operaciones generalmente se implementan sobre una arquitectura del tipo *farm workers*.

El siguiente pseudo-código muestra una aplicación map-reduce de recuento de palabras en un conjunto de documentos.

```
function map(String name, String document):  
    // name: document name  
    // document: document contents  
    for each word w in document:  
        emit (w, 1)  
  
function reduce(String word, Iterator partialCounts):  
    // word: a word  
    // partialCounts: a list of aggregated partial counts  
    sum = 0  
    for each pc in partialCounts:  
        sum += pc  
    emit (word, sum)
```

Los frameworks ofrecen las operaciones de *splitting* de los items de entrada a los *mappers* y las operaciones *combine*, *sort* y *partition (shuffling)* distribuyendo sus salidas entre los *reducers*. Estas operaciones pueden ser redefinidas por el usuario.

## Topologías virtuales

Independientemente de la arquitectura física, es común que los nodos se organicen lógicamente formando determinadas formas o *topologías* de comunicación. En lugar de favorecer una comunicación todos con todos, se sigue un patrón de comunicaciones siguiendo alguna estrategia sugerida naturalmente por la topología lógica seleccionada.

Las topologías comúnmente usados son *pipelines*, *árboles*, *anillos* y *grillas*, como se muestra en la siguiente figura. Una topología menos estructurada puede tener formas de grafos arbitrarios.

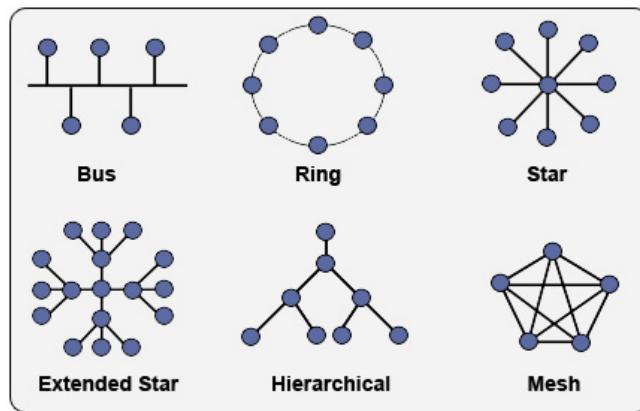


Figura 1.3: Algunas topologías.

## Comunicaciones

Los sistemas distribuidos se basan en la interacción entre los nodos participantes mediante la comunicación de mensajes. Si bien es posible implementar sistemas distribuidos usando las APIs provistas por una capa de transporte de algún protocolo de comunicaciones generalmente se utilizan bibliotecas o frameworks de comunicación, conocidos como *middleware* que brindan un mayor nivel de abstracción.

Uno de los frameworks muy utilizados son los basados en *Remote Procedure Call (RPC)*, los cuales proveen una abstracción basada en interfaces de módulos u objetos donde el programador sigue la técnica familiar de invocación a funciones o métodos. Cada objeto remoto tiene un *proxy* local el cual convierte cada invocación a alguna de sus funciones en un mensaje (*serialización*), lo envía al host remoto y éste convierte el mensaje en una invocación local (*dispatching*). Luego de la ejecución de la función, el resultado se retorna al host invocante. RPC generalmente usa comunicación sincrónica, aunque varios frameworks y bibliotecas modernas usan comunicación asincrónica retornando *futuros*, es decir, objetos que quedarán a la espera del resultado y que podrá obtenerse posteriormente por la aplicación. Esto permite que la aplicación solape comunicaciones con computaciones concurrentemente. Para más detalles sobre RPC ver [RPC](#) en las notas del curso de Redes y Telecomunicaciones.

## Frameworks de pasaje de mensajes

Otras abstracciones posibles son aquellas basadas en *patrones de comunicación* convenientes entre *grupos de nodos*.

ZeroMQ es una biblioteca de abstracciones de patrones de comunicación basado en mensajes sobre sockets. Se basa en comunicaciones asincrónicas, permitiendo así explotar al máximo el paralelismo entre comunicación y uso de CPU. Soporta varios patrones útiles de comunicación como

- *Request/Reply*: Provee mecanismos similares a RPC.
- *Publish/Subscribe*: Conecta un conjunto de *publishers* con un conjunto de *subscribers*. En este patrón cada mensaje enviado por un *publisher* es enviado (multicast) a sus *subscribers*.
- *Pipeline*: Permite configurar un conjunto de nodos en una topología virtual de *etapas*, donde cada etapa procesa datos de la anterior y los pasa a la siguiente. Este patrón es común para la división de tareas paralelas y su sincronización.

Otros frameworks y bibliotecas permiten otras abstracciones y patrones de comunicación de uso común como por ejemplo, MPI.

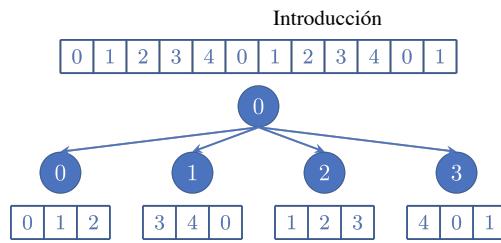
El estándar Message Passing Interface (MPI) provee la descripción de una API de funciones que ejecutan patrones de comunicación comúnmente usados para la implementación de aplicaciones paralelas de alto rendimiento (HPC). MPI se utiliza generalmente en *clusters* de computadoras y brinda las abstracciones para ver un cluster como un sistema multi-procesador basado en el modelo de programación *Single Program Multiple Data*. En este modelo existe un único programa que se ejecuta en cada nodo participante. La API permite que cada nodo tome diferentes roles en cada comunicación grupal.

## 1. Comunicaciones punto a punto

- `send(data, dst)` : El nodo invocante envía `data` al nodo de destino dado.
- `recv(data, src)` : El nodo invocante recibe datos del origen dado. Si la comunicación es sincrónica, el invocante se bloquea hasta la recepción del mensaje.

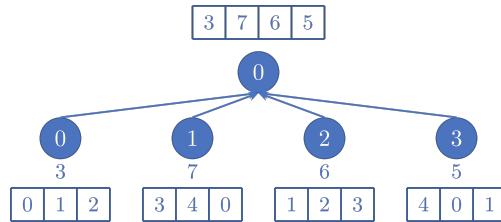
## 2. Comunicaciones grupales

- `broadcast(data, src, grp)` : Envía copias de `data` desde `src` a todos los demás nodos del grupo. Esta operación se conoce también como `map` .
- `scatter(data, src, grp)` : El nodo origen envía partes disjuntas de `data` a los demás nodos del grupo.



**Figura 1.4: Scatter con src=0.**

- `gather(data, dst, grp)` : El nodo `dst` recibe partes disjuntas de `data` desde el resto del grupo.



**Figura 1.5: Gather con dst=0 (de la suma de los valores de cada nodo).**

- `reduce(data, op, dst, grp)` : El nodo `dst` actúa como el receptor de datos enviados por los demás y aplica la operación `op` para retornar el valor final. El parámetro `data` es el valor transmitido por cada nodo diferente del destino. La operación `op` (+, -, \*, min, max, avg, ...) generalmente debe ser asociativa y conmutativa para permitir su aplicación en cualquier orden (a medida que los datos arriban). Es posible implementar `reduce` usando `scatter` y `gather`.

En MPI, cada nodo se identifica por su *rank* (identificador de proceso independiente del sistema operativo). La función `rank()` ejecutada por un nodo retorna su identificador. También MPI permite formar topologías lógicas en forma de grillas o grafos, realizando el *mapping* de posiciones lógicas a identificadores de procesos, facilitando al programador coordinar las comunicaciones.

Un *cluster* es un conjunto de computadoras personales inter-conectadas por interfaces de red de alta velocidad usado como un sistema multi-procesador (*multi-computadoras*)

Comúnmente los sistemas basados en mensajes soportan comunicaciones asíncronas por lo que se requieren que los mensajes sean (temporalmente) encolados para su posterior procesamiento.

Los frameworks mencionados se implementan en forma de biblioteca, comúnmente en C/C++ con *bindings* a un gran número de lenguajes de programación.

---

Próximo >

## Coodinación

# Coordinación

En aplicaciones distribuidas es común que un proceso deba esperar a que otro alcance cierto estado antes de poder continuar. En otros casos los procesos deben acceder a recursos compartidos y para garantizar consistencia deben acceder de manera exclusiva. Este último caso se conoce como *sincronización en el acceso a datos*. *Coordinación* es un término más general para lograr una interacción controlada entre los procesos.

Como cada proceso en un sistema distribuido tiene su propio reloj físico, lograr una sincronización de estos tipos de relojes con un alto nivel de precisión es difícil de lograr. Algunos protocolos como [Network Time Protocol \(NTP\)](#) permiten la sincronización de los relojes físicos de las computadoras conectadas en red y ampliamente usado en Internet. El [NIST](#) soporta un conjunto de servidores con relojes atómicos de máxima precisión y ofrece el servicio de hora y fecha basado en NTP y otros. Nuestras computadoras personales generalmente ejecutan un cliente NTP (utilidad del sistema) que mantiene el reloj de hardware actualizado.

Hay que considerar que la sincronización no es perfecta, ya que se deben tomar en cuenta las demoras en las transmisiones de los mensajes, por lo que los protocolos utilizan algoritmos de ajustes aproximados. Para más información, ver la sección 5.1 de [DS-4ed] y [NTP](#).

## Reloj lógico

Para conseguir operar en forma coordinada, alcanza con el uso de un modelo simplificado de tiempo. El objetivo generalmente es lograr capturar la relación de *causalidad en la ocurrencia de los eventos* en el sistema.

Denotaremos  $a \rightarrow b$  si  $a$  ocurre antes que  $b$ . Esta relación implica:

1. Si  $a$  y  $b$  ocurren en el mismo proceso y  $a$  ocurre antes que  $b$ , entonces  $a \rightarrow b$ .
2. Si  $a$  es el evento de enviar un mensaje desde un proceso y  $b$  es el evento de recibir ese mensaje en otro proceso, entonces  $a \rightarrow b$ .

Esta relación (*sucede antes que*) es transitiva.

Leslie Lamport, en 1978, en [Time, Clocks and the Ordering of Events in a Distributed System](#) propuso el siguiente algoritmo conocido como *relojes escalares*.

1. Cada proceso  $p_i$  tiene su *reloj local*  $c_i$ .
2. En cada cambio de estado interno de  $p_i$ :  $c_i \leftarrow c_i + 1$ .
3. En  $p_i$ : Antes de enviar un mensaje,  $c_i \leftarrow c_i + 1$ . El mensaje toma la forma  $(msg, c_i)$ , donde  $c_i$  se conoce como el *timestamp*.
4. En cada mensaje recibido de  $p_j$ :  $(msg, c_j)$ ,  $p_i$  ajusta su reloj:  $c_i = \max(c_i + 1, c_j)$

Es posible definir una función que captura la noción de un reloj global del sistema:  $C(e) = c_i$  si el evento  $e$  ocurrió en el proceso  $p_i$ .

Es fácil demostrar que  $C()$  captura la noción de paso del tiempo discreto interna de cada proceso y de los eventos de emisión y recepción de mensajes:

1. Si  $a$  y  $b$  son eventos que ocurren en el mismo proceso y  $a \rightarrow b$ , entonces,  $C(a) < C(b)$ .
2. Si  $a$  es el evento de *enviar un mensaje* a otro proceso y  $b$  es el evento de recepción de ese mismo mensaje,  $C(a) < C(b)$ .

La implementación de incluir los *timestamps* en los mensajes generalmente se implementa en el *middleware* utilizado para la comunicación entre procesos.

Cabe notar que  $C(a) < C(b)$  no necesariamente implica que  $a \rightarrow b$  como en el caso que  $a$  y  $b$  sean eventos que ocurren en dos procesos independientes (no se comunican). En este caso esos eventos se consideran *concurrentes*.

**Los relojes escalares no capturan la relación de *causalidad*:  $C()$  es un orden parcial.**

Es posible capturar *causalidad* si cada proceso tiene en cuenta el *estado interno* de cada otro proceso en el sistema. Ya que un reloj escalar abstrae los cambios de estado en un proceso, es posible capturar esta información si cada proceso incluye su último valor conocido del estado de los demás.

Los *relojes vectoriales* son una implementación de esta idea.

1. Cada proceso (sea  $p_i$ ) mantiene un *vector*  $vc_i[N]$ , con  $vc_i[i]$  como su reloj lógico local.  $N$  es el número de procesos en el sistema.
2. En cada evento interno (o cambio de estado),  $p_i$  hace  $vc_i[i] = vc_i[i] + 1$ .
3. En cada mensaje enviado, se incluye el *timestamp*:  $send(p_j, msg + vc_i)$ .
4. En cada mensaje recibido desde  $p_j$ ,  $p_i$  hace  $vc_i[k] = \max(vc_i[k], vc_j[k])$  para todo  $k$  con  $1 \leq k \leq N$ .

La principal desventaja de los *relojes vectoriales* es que el *timestamp* de cada mensaje puede tener un tamaño considerable para un  $N$  grande.

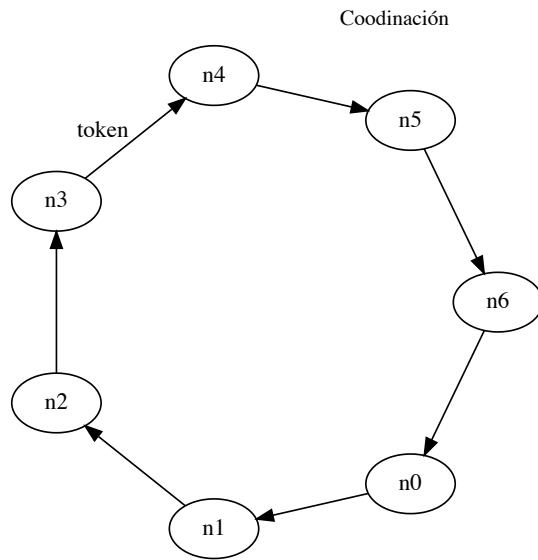
## Exclusión mutua

El acceso exclusivo en el uso de recursos compartidos es común en sistemas concurrentes y distribuidos. En sistemas basados en memoria compartida es común el uso de primitivas de sincronización como los *spinlocks*, *variables de condición*, *semáforos* o *monitores*. En un sistema distribuido se requiere lograr un protocolo que garantice que los procesos accederán al recurso sólo cuando los demás no lo estén usando y además se debería garantizar *progreso*, es decir, que un proceso que requiera el recurso, eventualmente lo logrará.

Una solución inmediata es designar un proceso (nodo) del sistema como coordinador de los recursos, centralizando el estado del sistema. Cada nodo que requiere un recurso le envía al coordinador un mensaje **REQUEST** al coordinador y queda a la espera por la respuesta. El coordinador responde cuando el recurso esté libre. Luego el proceso podrá utilizar el recurso y notificar al coordinador cuando lo libere.

La elección de un coordinador puede ser definida estática o dinámicamente mediante un algoritmo de *elección de líder* que se analiza más adelante.

Obviamente el coordinador es un punto de falla crítico por lo que si se quiere lograr un sistema tolerante a fallas el coordinador debería ser *elegible* ante la detección de su caída.



**Figura 2.1: Topología token-ring.**

En esta topología es fácil lograr la exclusión mutua en el acceso a un recurso. Simplemente el protocolo define un *token* (o *capacidad*) por recurso. El mecanismo de circulación del token en forma secuencial garantiza el acceso exclusivo al recurso. Cada proceso que desea acceder a un recurso, debe esperar por el token, acceder al recurso y luego reenviar el token al próximo proceso en el anillo.

La configuración del anillo puede realizarse a-priori (estáticamente) o dinámicamente, asumiendo que cada nodo puede determinar su sucesor, por ejemplo mediante un esquema de identificación numerable de los nodos.

Las desventajas de tal enfoque es que es necesario construir a priori la topología virtual y si el número de procesos es grande, pueden ocurrir demoras considerables. Otra desventaja es que para que sea tolerante a fallas hay que asumir que el anillo es dinámico y que cada nodo es capaz de detectar una falla e iniciar una *reconfiguración* del anillo en forma automática.

En 1981, Ricart y Agrawala propusieron un algoritmo distribuido y requiere el uso de mensajes con *timestamps* (*reloj escalar*) y cada proceso tiene una cola de mensajes ordenados por *timesteps* por cada recurso. Sean  $n$  procesos:

- El proceso  $p_i$  desea utilizar el recurso  $r$ :
  1.  $\text{broadcast}(\text{enter}\{p_i, r, ts_i\})$
  2.  $p_i.\text{state} \leftarrow \text{waiting}(r, ts_i)$
  3. Espera por  $n$  mensajes  $(p_j, OK_r)$ , luego
    1.  $p_i.\text{state} \leftarrow \text{using}(r)$  y usa  $r$
    2. Al salir de la región crítica hace  $\text{send}(k, OK_r), \forall(k | \{k, ts_k\} \leftarrow r\_dequeue())$

- Un proceso  $p_j$  recibe un mensaje  $enter(p_i, r, ts_i)$ :
  - Si  $p_j.state \neq using(r)$ :  $send(p_i, OK_r)$
  - Si  $p_j.state = waiting(r, ts_j)$  y  $ts_i < ts_j$ :  $send(p_i, OK_r)$
  - Si  $p_j.state = using(r)$ :  $r\_queue.insert(\{p_i, ts_i\})$

## Broadcasting/multicasting de mensajes

Muchas aplicaciones y algoritmos distribuidos requieren que un nodo envíe mensajes a un grupo de nodos en la red. Este tipo de comunicación se conoce como *multicast*. Se denomina *broadcast* cuando el grupo contiene la totalidad de los nodos.

Es común que en redes locales un protocolo de enlace, como por ejemplo los basados en Ethernet, provean un mecanismo eficaz. En general en redes como Internet se use IP como protocolo punto a punto.

Se requiere que esta operación sea *confiable*, es decir que debe cumplir con algunas de las siguientes propiedades.

- ***FIFO broadcast***: Si  $m_1$  y  $m_2$  son enviados por el mismo nodo y  $broadcast(m1) \rightarrow broadcast(m2)$ , entonces en cada receptor  $m_1$  debe entregarse (*deliver*) antes que  $m_2$ .
- ***Broadcast causal***: Si  $broadcast(m1) \rightarrow broadcast(m2)$ , entonces en cada receptor  $m_1$  debe entregarse (*deliver*) antes que  $m_2$ .
- ***Broadcast con orden total***: Si  $m_1$  se entrega antes que  $m_2$  en un nodo, entonces también se deberán entregar en ese orden en todos los nodos.
- ***Broadcast FIFO-orden total***: Combinación de *FIFO* y *orden total*.

El último tipo de *broadcast* es la más fuerte y se conoce también como *broadcast atómico* mientras que *FIFO broadcast* es el más débil ya que permite que los mensajes sean entregados *fuerza de orden*.

## Algoritmos

Realizar broadcasting en un protocolo del tipo *token ring* sobre una topología de anillo es natural, ya que el mensaje de broadcast puede implementarse como el envío a sí mismo. El mensaje pasará por cada nodo hasta retornar al emisor, el cual al recibirla lo retira de la red.

Este algoritmo no es muy usado en sistemas grandes, escalables y tolerantes a fallas ya que cada enlace y cada nodo representa un punto de falla crítico.

La mayoría de los algoritmos se basan en la siguiente idea:

1. El nodo emisor envía a todos sus *vecinos* por sus *enlaces directos*.
2. Cada nodo, cuando recibe un mensaje lo reenvía por sus enlaces, excepto por el que lo recibió.

Es fácil ver que el mensaje se diseminará por toda la red.

Este algoritmo es muy mejorable ya que un nodo podría recibir varias copias del mensaje (reenvíos de otros vecinos). Esto se puede solucionar con un *timestamp* en el mensaje que permita detectar y descartar mensajes duplicados. Este algoritmo se conoce comúnmente como *inundación (flooding)*.

Otra modificación es que en lugar de enviar el mensaje por los enlaces directos, cada nodo reenvíe el mensaje a  $n$  nodos elegidos aleatoriamente. Esto cambia la noción de *vecino*. Estos protocolos reciben el nombre de *gossip* (*pandemia* o *rumor*) ya que se basan en modelos de propagación de enfermedades o *chismes*.

## Elección

Cualquier algoritmo o protocolo que requiera un coordinador central, deberá realizar una selección del mismo. En un sistema tolerante a fallas, si falla un coordinador, algún otro proceso deberá detectar esta anomalía y disparar una elección de otro líder o coordinador.

En este caso un modelo centralizado no corresponde ya que justamente este algoritmo se debe disparar ante la falta de tal coordinador.

En una topología virtual de anillo nuevamente el algoritmo es simple, ya que el proceso  $p_i$  que detectó la falla en el coordinador, luego de reconfigurar el anillo, genera un token ( $ELECTION, p_i$ ) y lo envía a su sucesor. Cada proceso  $p_j$  que reciba un token ( $ELECTION, p_k$ ), reenvía ( $ELECTION, max(p_j, p_k)$ ). El proceso iniciador ( $p_i$ ) cuando reciba el token ( $ELECTION, p_w$ ), lo consume y envía ( $ELECTED, p_w$ ), el cual será reenviado por cada proceso intermedio definiendo en su estado *coordinator* =  $p_w$ . Finalmente  $p_i$ , al recibir ( $ELECTED, p_w$ ) lo consume. El proceso elegido es  $p_w$ .

García y Molina en 1982 propusieron un algoritmo distribuido que se conoce como *bully* (matón) que selecciona el proceso con mayor identificador. El algoritmo se describe en los siguientes pasos:

- Sea  $p_i$  el proceso *iniciador*. Envía  $send(p_k, ELECTION)$  a todos los procesos con  $k > i$ 
  - Si no obtiene respuestas (o no conoce  $p_k$  con  $k > i$ ), cambia su estado a  $coordinator = p_i$  y  $broadcast((ELECTED, p_i))$ .
  - Si recibe una respuesta de  $p_j$ ,  $p_j$  es el nuevo *iniciador*.
- Un proceso  $p_j$  que recibe  $ELECTION$  desde  $p_i$  responde sólo si  $j > i$ .
- Cada proceso que recibe  $(ELECTED, p_i)$ , actualiza su estado a  $coordinator = p_i$ .

Es un algoritmo básicamente de *flooding (inundación)* en un *grafo acíclico dirigido (DAG)*. Se debe notar que en un momento dado puede haber varios *iniciadores* simultáneos. Ejercicio: Mostrar que aún con múltiples iniciadores, se logrará una elección.

Uno de los problemas de este algoritmo es que es costoso ya que requiere la transmisión de  $O(n^2)$  mensajes en un grupo de  $n$  nodos.

No es tolerante a fallas, ya que queda esperando por todas las respuestas. En la sección [tolerancia a fallas](#) se analizan algoritmos de *consenso* que pueden usarse para elegir un coordinador en presencia de fallas.

## Protocolos Publish/Subscribe

Estos protocolos permiten implementar el conocido patrón arquitectural *publish/subscribe*. En este patrón un *publisher* envía *notificaciones* por un canal al que están conectados los *subscribers*.

Estos protocolos son aplicables en una gran variedad de escenarios como la adquisición del estado de contactos en aplicaciones de chat y bases de datos distribuidas, donde los subscriptores se corresponden a las réplicas de un conjunto de datos que reciben (y posiblemente envíen) notificaciones de *actualizaciones de datos*.

Una topología descentralizada requiere la formación de una *red overlay* de peers interconectados. El principal problema es la comunicación de las notificaciones a los nodos *subscriptores*, lo cual puede hacerse mediante algunos protocolos de diseminación como los ya mencionados.

Una técnica comúnmente usada es el *ruteo selectivo*, lo cual es similar a los protocolos de *ruteo multicast* vistos previamente.

En este modelo, cada router mantiene una tabla (generalmente parcial) de los nodos (vecinos) que se han suscripto a cada evento. En la recepción de un mensaje de *suscripción* se lo incluye en la tabla.

Al ocurrir un evento (ejemplo: un usuario se conecta a ese nodo) o recibir una *notificación* de un evento, se reenvía el mensaje con la notificación a los nodos de la tabla cuya entrada *coincide* (*match*) con el evento registrado.

**Ejercicio:** Comparara esta estrategia con el algoritmo de *ruteo multicast* visto en el curso de redes.

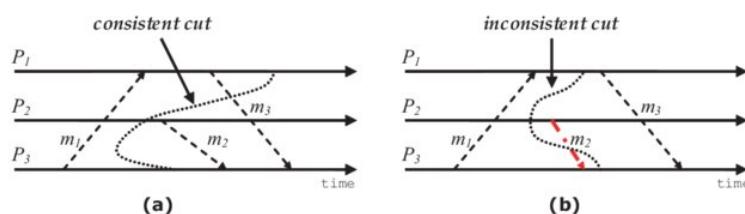
## Estado global y toma de instantáneas

Una *instantánea* (*snapshot*) es un *estado consistente* del sistema que puede ser usado en una recuperación. Otras aplicaciones incluyen recolección de basura distribuida, detección de deadlock y terminación y monitoreo o depuración.

Sean  $N$  procesos, la *historia* de un proceso  $h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$  es la secuencia de eventos ocurridos en  $p_i$ . Cada evento es un cambio de estado interno o el envío y recepción de mensajes. La *historia global del sistema* se define como  $H = \bigcup_{i=0}^{n-1} p_i$ .

**Definición:** Un *corte* (*cut*)  $C$  es un subconjunto de  $H$ , el cual está formado por la unión de *prefijos* de  $h_i$ ,  $0 \leq i < N$ .

Cualquier subconjunto podría ser *inconsistente* ya que podría incluir el evento de recepción de un  $m$  por  $p_j$  (desde  $p_i$ ) pero no el evento de envío de  $m$  por  $p_i$ , como en la figura (b).



**Definición:** Un *corte*  $C$  es *consistente* si  $\forall e \in C, f \rightarrow e \implies f \in C$ .

Un corte puede incluir mensajes enviados aún no recibidos.

**Una *linearización* es un orden de eventos de  $H$  que respeta la relación  $\rightarrow$ .**

En una linearización es una secuencia de eventos que *recorre cortes consistentes*.

## Toma de instantáneas (snapshot)

A continuación se describe el algoritmo propuesto por Chandy y Lamport en 1985 útil para determinar el estado global de un sistema distribuido.

El algoritmo se basa en registrar los estados locales en cada proceso. Cada proceso podría enviar su estado a un monitor de estados central el cual podría verificar propiedades globales.

El algoritmo se basa en el envío de un mensaje especial (la *marca  $M$* ) que actúa como un *iniciador* del proceso de registrar el estado en cada proceso.

Cada proceso  $p$  tiene un conjunto de variables *vars<sub>p</sub>*, *canales de entrada (ic<sub>p</sub>)* y *salida (oc<sub>p</sub>)*.

El estado de un proceso es el conjunto de valores de sus variables y de los mensajes recibidos por cada canal de entrada.

- Un proceso *iniciador p* realiza los siguientes pasos:
  1.  $state_p = (vars_p, messages(ic_p))$ .
  2.  $send(M, oc_p)$  (envía  $M$  por los canales de salida)
  3. Comienza a registrar los mensajes de sus canales de entrada.
- Cuando un proceso  $p$  recibe la marca sobre un canal de entrada  $c \in ic_p$ :
  - Es la primera vez que ve (conoce) la marca:
    1.  $state_p = (vars_p, messages(ic_p))$
    2.  $c = \emptyset$
    3.  $send(M, oc_p)$  (colabora en la redistribución de la marca)
    4. comienza a registrar mensajes de los demás canales de entrada.
  - En otro caso:
    1.  $messages(c) = rcvd\_msgs(c)$  (mensajes recibidos por  $c$  desde el último estado registrado)
    2. Deja de registrar mensajes nuevos recibidos por  $c$

El algoritmo descripto permite registrar estados que permiten construir *cortes consistentes*.

Ejemplo: <> Sean dos procesos  $p_1$  y  $p_2$ , conectados por dos canales unidireccionales  $c_{12}$  (desde  $p_1$  a  $p_2$ ) y  $c_{21}$  (desde  $p_2$  a  $p_1$ ) donde  $p_2$  es un vendedor de items.

1. Estado  $S_0$ :  $p_1 = (\text{account} = \$1000, \text{items} = 0)$ ,  $p_2 = (\text{account} = \$50, \text{items} = 50)$ ,  $c_{12} = c_{21} = \emptyset$

2.  $p_1$  registra su estado y envía un requerimiento de compra

Estado  $S_1$ :  $p_1 = (\text{account} = \$900, \text{items} = 0)$ ,  $p_2 = (\text{account} = \$50, \text{items} = 50)$ ,  $c_{12} = \{\text{order}(10, \$100), M\}$ ,  $c_{21} = \emptyset$

3.  $p_2$  recibe  $\text{order}(10, \$100)$  antes que la marca, así responde con el mensaje  $\text{items}(5)$ .

Estado  $S_2$ :  $p_1 = (\text{account} = \$900, \text{items} = 0)$ ,  $p_2 = (\text{account} = \$150, \text{items} = 45)$ ,  $c_{12} = \emptyset$ ,  $c_{21} = \text{items}(5)$

4.  $p_1$  recibe el mensaje  $\text{items}(5)$  por  $c_{21}$

Estado:  $S_3$ :  $p_1 = (\text{account} = \$900, \text{items} = 5)$ ,  $p_2 = (\text{account} = \$150, \text{items} = 45)$ ,  $c_{12} = c_{21} = \emptyset$

Cada estado es un *corte consistente*.

## Referencias

1. [Distributed Systems. 4th edition. Maarten Van Steen and Andrew S. Tanenbaum](#)
2. *Distributed Systems: Concepts and Design.* G. Coulouris et al. Fifth Edition. 2012.
3. [Distributed Systems: Course notes at University of Cambridge 2021/22](#)

&lt; Anterior

## Introducción

Próximo &gt;

## Replicación y Distribución

# Particionado y Replicación

La *replicación* de datos y computadores permite que un sistema escale permitiendo distribuir los requerimientos en diferentes nodos de la red.

Por otra parte, cuando el sistema debe manejar grandes volúmenes de datos, a veces es imposible almacenar la totalidad de esos datos en un nodo, por lo que los datos deberán *dividirse* en partes y ser almacenados en un conjunto de nodos determinados.

Ambas técnicas permiten la *escalabilidad* de los sistemas y también la tolerancia a fallas. La replicación permite que si un nodo de la red se torna inaccesible, otros puedan seguir brindando el servicio. El particionado de los datos permite la distribución de las computaciones sobre ellos y asegura que al menos algunos datos serán accesibles, aún en presencia de fallas. Por supuesto, ambos enfoques generan problemas adicionales.

La *replicación* genera el problema de mantener las réplicas *consistentes* (actualización de las copias).

El *particionado* de datos genera el problema que las operaciones deben aplicarse a un conjunto de nodos (los que contienen los datos involucrados)- Otro problema es lograr un *particionado uniforme* (balance de carga) entre los nodos.

## Distribución de datos

Uno de los principales problemas es decidir cómo dividir los datos para lograr una fragmentación uniforme. Analizaremos tres estrategias comúnmente usadas, asumiendo que los items de datos se identifican únicamente por alguna clave.

1. **Particionado de claves:** El conjunto de claves se partitiona en grupos entre el conjunto de nodos. El particionado puede ser basado en conjuntos o rangos de claves o usar un esquema jerárquico como por ejemplo, en el sistema DNS.

La ventaja de este enfoque es que la búsqueda de un ítem puede dirigirse al nodo en el cual está almacenado, por ejemplo, mediante la consulta de un índice de particionado.

La principal desventaja es que algunos nodos pueden quedar sobrecargados de ítems o que algunos ítems son más frecuentemente accedidos o actualizados que otros, en cuyo caso no se logrará un buen balance de carga, creando *hotspots* (*puntos calientes*) en el sistema. La detección de *hotspots* permite una re-distribución de los datos logrando un mejor balance de carga.

2. **Hashing de claves:** El utilizar una función de la forma  $\text{hash}(\text{key}) \bmod N$  para mapear el nodo que contiene el ítem provee una mayor uniformidad en la distribución de los ítems.

Desventajas: Ítems lógicamente contiguos podrán estar en diferentes nodos. Además si se cambia la cantidad de nodos hay que redistribuir completamente los datos.

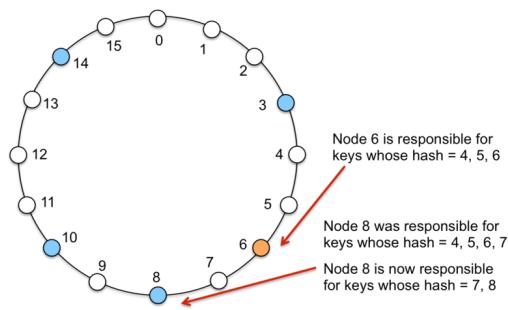
3. **Hashing consistente:** En este esquema, a cada nodo se le asocia un identificador aleatorio de  $m$  bits que determina su posición en una topología virtual de anillo. Cada ítem de datos tiene una clave correspondiente a un  $k = \text{hash}(\text{value})$  de  $m$  bits y se almacena en el nodo  $n = \text{successor}(k)$ , es decir, en el nodo identificado como  $k$  o (si no existe) en el nodo  $k'$  próximo en el anillo tal que  $\text{hash}(k) < k'$ . De esa forma, cada nodo es responsable de los ítems con claves mayores (módulo  $m$ ) que su predecesor.

El uso de funciones *hash* provee una dispersión uniforme. La inserción (o remoción) de un nuevo nodo en una posición determinada del anillo requiere que se redistribuyan las claves de su sucesor y que se actualicen las referencias de *sucesor* y *predecesor* en los nodos involucrados.

Cuando un nodo recibe desde un cliente una operación `read(k)` o `write(k, v)` la debe reenviar al nodo  $\text{successor}(k)$ .

Esta estrategia también se conoce como *distributes hash tables (DHT)* y hay varios protocolos basados en esta idea como por ejemplo [Chord](#). Este protocolo incluye detalles de implementación para encontrar  $\text{successor/predecessor}(k)$  en  $O(\log(N))$  en un anillo de  $N$  nodos. Usa como función hash SHA-1 que genera claves de 160 bits.

También este esquema sugiere una estrategia simple de replicación. Por ejemplo, un nodo podría replicar los datos de los  $k$  predecesores o sucesores.



**Figura 3.1:** Inserción de un nodo en el anillo.

Un ejemplo de uso de esta técnica de distribución de datos es Amazon Dynamo el cual da soporte a los servicios como AWS S3.

## Replicación

La replicación de datos y servicios (computaciones) permite la escalabilidad y hace a los sistemas más tolerantes a fallas. Hay varios problemas a resolver con la replicación. Uno de ellos es el número de réplicas y su ubicación en la red. También se debe determinar cómo replicar contenidos. Esto tiene como consecuencia el problema de *consistencia* de las réplicas, esto es, cómo hacer que el sistema como un todo funcione transparentemente como un almacén único o centralizado ante los clientes.

La ubicación de las réplicas puede hacerse en base a la geografía, muy común en aplicaciones en Internet, muchas veces luego de un estudio de las demandas. Esto habilita a que los clientes se conectan con un server o réplica más próxima, teniendo en cuenta la topología de interconexión. Esta técnica se aplica en *redes overlay* usadas por los gigantes de servicios en Internet como Google, YouTube, AWS, Netflix, y otros.

En cuanto a la replicación de contenidos pueden aplicarse varias estrategias:

1. *Notificación de una actualización*: También se conoce como *protocolos de invalidación*, donde un nodo notifica a los demás que cierto contenido ha cambiado, por lo que las réplicas se tornan inválidas. Esta técnica se usa comúnmente en sistemas multiprocesadores para *invalidar* entradas en las cachés locales.

2. *Propagación del dato o de la operación del update:* En este caso el nodo que procesó un *update (write)* sobre un ítem, propaga el nuevo valor o la operación para que sea ejecutada en las demás réplicas. Cuando se propagan *operaciones* (en vez de los datos), se dice que se usa *replicación activa* ya que cada nodo es un proceso capaz de ejecutar las operaciones. Este mecanismo es común en los sistemas de bases de datos distribuidos.

El manejo de las réplicas puede hacerse de diferentes formas:

1. *Leader-replica:* En este caso existe un nodo *leader* o coordinador que recibe los *writes* desde los clientes y luego propagará la actualización a las réplicas. Las lecturas pueden hacerse sobre cualquier réplica. Un dispositivo de almacenamiento **RAID** tiene una arquitectura de este tipo.
2. *Multi-master:* Los clientes pueden realizar *writes* en cualquier réplica y ésta propagará los cambios a las demás réplicas. Esta arquitectura genera el problema de *consistencia* ya que pueden realizarse escrituras sobre el mismo dato en forma concurrente en diferentes réplicas.
3. *Basadas en caché:* En este caso, los clientes o servidores que han resuelto una consulta previa de un ítem, lo almacenan temporalmente en su memoria local conformando una caché de datos. Los ítems en la caché pueden invalidarse por su *tiempo de vida* o por notificaciones.

## Consistencia

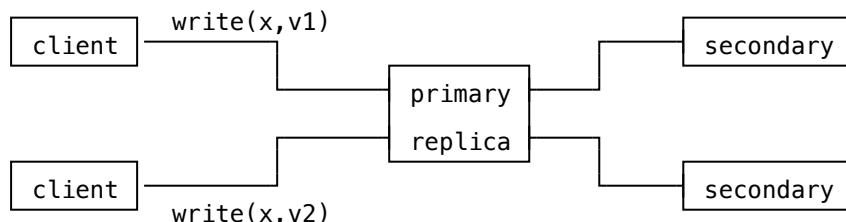
En sistemas distribuidos la arquitectura *leader-replica* conduce a un sistema centralizado (al menos para procesar los *writes*), siendo aplicable a sistemas con pocas actualizaciones. Además tienen un *único punto de falla*.

Los sistemas *multi-master* (o *peer-to-peer*) deben resolver el problema de consistencia generado por las posibles actualizaciones concurrentes. Hay varios tipos o variantes de la noción de consistencia, dependiendo del tipo de servicio requerido por la aplicación.

- **Secuencial:** Una secuencia de operaciones (*read/write*) es válida si se corresponde a una traza posible de cada cliente. Esta noción es fuerte ya que requiere que todos los procesos deberían seleccionar esa traza, requiriendo un sistema determinístico. Requiere que el sistema preserve el orden de operaciones de cada cliente. Puede lograrse con un protocolo de *replicación activa* basado en comunicación *multicast confiable con orden total*.
- **Linearización:** Similar a la consistencia secuencial pero centrada en el tiempo en que ocurren las comunicaciones. Puede lograrse con una arquitectura de *replicación pasiva* (*backup primario* o *replicación basa en líder*).
- **Causal:** Garantiza que se fuerza la secuencialidad sólo sobre las operaciones sobre el mismo dato. Es decir, si un proceso  $p_i$  ejecuta  $\text{write}(x, v)$  y luego  $p_j$  ejecuta  $x = \text{read}(x)$  (antes de un nuevo  $\text{write}(x, ?)$  en cualquier otro proceso), entonces  $x = v$ . Dos operaciones  $\text{write}(x, v_1)$  y  $\text{write}(y, v_2)$  son *concurrentes* y pueden hacerse en cualquier orden.
- **Eventual:** Tolera inconsistencias temporales, pero que luego de un tiempo sin *updates* todas las réplicas se tornarán consistentes. Ejemplos de aplicaciones que usan este modelo de consistencia son los *content delivery networks (CDN)* que contienen archivos de sólo lectura y el sistema DNS, los cuales usan replicación basada en *caché (pull based)*.

## Replicación pasiva (leader-replica)

Un sistema de replicación pasiva o *master-replicas* se basa en la arquitectura de la siguiente figura.



Los clientes realizan las operaciones `write` a la réplica primaria, éste ejecuta la operación y en caso de éxito reenvía (broadcast) la operación a las réplicas secundarias (que deberían tener éxito ya que ejecutan el mismo programa). Esto es equivalente a *1 phase commit*.

La réplica primaria o *master* aplica las operaciones en el orden en que los recibe y se los reenvía a las réplicas en el mismo orden. Esto garantiza *linearización*.

## Arquitectura multi-master

En una arquitectura multi-master, los clientes envían las operaciones a todas las réplicas.

En un sistema *multi-master* pueden ocurrir *conflictos*. Un conflicto *write/write* ocurre cuando se realizan concurrentemente escrituras del mismo dato en diferentes nodos. Un conflicto *read/write* es cuando ocurre una lectura en un nodo y concurrentemente una escritura del mismo ítem en otro, antes que se inicie su propagación.

Una solución puede basarse mediante comunicaciones *multicast confiable totalmente ordenada* que garantiza que los mensajes serán entregados en cada réplica en el mismo orden. Este tipo de comunicación garantiza *consistencia secuencial* ya que preserva el orden de las operaciones en cada programa cliente.

Otras posibles soluciones incluyen *regiones críticas*, *transacciones distribuidas* o usando protocolos basados en *quorums* que se analizan en el próximo capítulo.

En un sistema basado en *quorum*, cada ítem de datos tiene asociado su *versión* o *timestamp*, un valor numérico creciente en cada *write*. El *quorum N* es el conjunto mínimo de nodos que deben responder a un requerimiento de un cliente. Un cliente debe contactar a un grupo de servidores ( $N_R$  para leer y  $N_W$  para escribir) y debe cumplir con las siguientes condiciones (con quorum  $N$ ):

1.  $N_R + N_W > N$
2.  $N_W > N/2$

La primera condición soluciona conflictos *read/write*. La segunda soluciona conflictos *write/write*.

# Registro compartido

Un dato replicado en un sistema distribuido puede verse como un registro compartido en un sistema de memoria compartido.

Las operaciones de lectura y escritura deben ser consistentes (consistencia secuencial). Attiya, Bar-Noy y Dolev en 1990 propusieron el siguiente algoritmo tolerante hasta  $f$  fallas.

```

write(v):
    t++
    broadcast(v, t)
    wait for n-f acks
    register = v

read():
    broadcast(read_msg)
    wait for n-f responses (v, t)
    return v with max t
  
```

Este algoritmo requiere al menos  $n > 2f$  procesos.

Una implementación es *linearizable* si:

1. Si las operaciones se ejecutan secuencialmente en el orden de la serialización, deberían arrojar el mismo resultado que en la ejecución concurrente.
2. Si  $op_1$  finaliza antes que  $op_2$ , entonces  $op_1$  precede a  $op_2$  en la linearización.

Se puede demostrar que esta implementación es *linearizable*.

El orden de operaciones es: ordenar los `writes` en el orden que ocurren e insertar los `reads` luego de los `writes` del valor que cada `read` retorna.

En el caso que  $op_1 = write$  y  $op_2 = read$  y  $op_2$  sigue inmediatamente a  $op_1$  puede demostrarse por contradicción. Supongamos que el `read` no ve al `write` previo. Entonces, por la implementación, deben haber dos conjuntos disjuntos de procesos de tamaño  $n - f$ . Así,  $2 * (n - f) \leq n$ , entonces  $n \leq 2f$  lo que contradice la hipótesis que  $n > 2f$ .

## Coordinación

## Tolerancia a fallas

# Tolerancia a fallas

La tolerancia a fallas es una propiedad indispensable para lograr *sistemas confiables (dependable systems)*, los cuales deben garantizar:

1. *Disponibilidad*
2. *Confiabilidad*
3. *Seguridad*
4. *Mantenibilidad*

Una falla en el sistema se debe por el malfuncionamiento tanto en el software como en el hardware de algún nodo o enlaces de la red. Hay diferentes tipos de fallas como se muestran en la siguiente tabla.

Tipo de falla	Descripción
Caída (crash)	Caída del sistema con funcionamiento previo normal
Omisión	Fallas en recepciones/envíos de mensajes
Timeouts	No responde dentro de los límites establecidos
Respuesta	Responde incorrectamente
Arbitraria	El nodo falla en intermitentemente

**Tabla 4.1: Tipos de fallas.**

Las fallas de tipo *respuesta* se conocen como *fallas bizantinas*. El término se debe al paper de Leslie Lamport quien fue un pionero en el estudio de este tema. Un nodo que tiene *fallas de respuesta* puede representar un *atacante* en la red ya que no sigue las reglas del protocolo o algoritmo y se dice que el nodo es *malicioso* o *traidor*. Este modelo de fallas se usa en análisis de seguridad.

## Enmascarado de fallas

Una forma de ocultar (*enmascarar*) fallas es por medio de redundancia. Es muy usado en hardware y común en sistemas distribuidos para soportar fallas en nodos o enlaces.

Una forma de replicación de software común en sistemas distribuidos se conoce como *process resilience (resistencia)* y se basa en la replicación de procesos formando grupos. Cada grupo puede organizarse según alguna de las topologías virtuales ya descriptas.

## Consenso

En un grupo de procesos replicados, si existen nodos con fallas la respuesta del sistema debería lograr un *consenso* sobre la operación ejecutada concurrentemente en las réplicas. Este consenso puede lograrse mediante un algoritmo de *inundación (flooding)* o mediante mensajes desde un *coordinador*, aunque es difícil de lograr en presencia de fallas arbitrarias.

En presencia de *fallas bizantinas* comúnmente se deberá lograr consistencia en coincidir en algún valor replicado o en una decisión en un sistema sin coordinación central mediante algún algoritmo distribuido de consenso.

Para ilustrar el problema Lamport planteó el *problema de los generales bizantinos*. En este problema hay al menos tres generales que deben sitiuar una ciudad protegida. Sólo puede garantizarse la victoria si al menos una mayoría de generales coincide en atacar en forma simultánea. El problema es que algunos generales pueden ser *traidores* o estar bajo el control del enemigo.

El objetivo es que los generales *honestos* coincidan en la decisión de atacar o no.

En 1988, Dwork y oros demostraron que para lograr *consenso por mayoría* en presencia de hasta  $k$  fallas arbitrarias (*Bizantinas*) se requieren al menos  $3k + 1$  nodos.

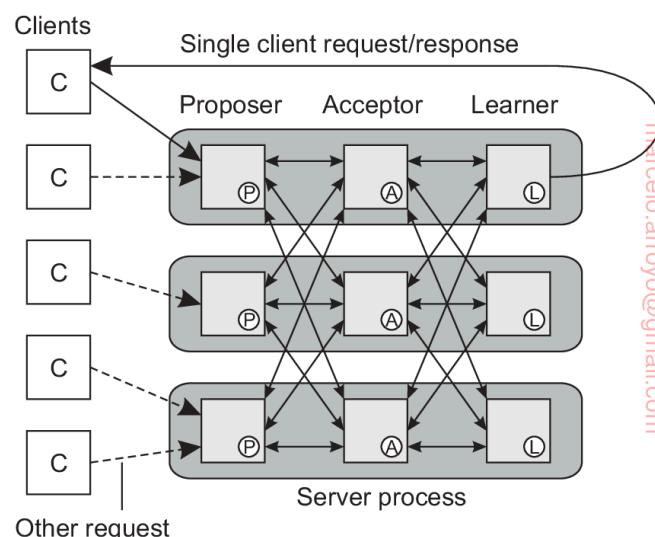
## Paxos

Leslie Lamport en 1989, publicó el artículo [The Part-Time Parliament](#), donde describe el algoritmo conocido como *Paxos* (por la isla Griega).

Los nodos en el sistema pueden tener los siguientes roles:

- *Client*: Realiza requerimientos de operaciones a un server que se convertirá en el *proposer*.
- *Proposer*: Servidor que acepta el requerimiento del cliente y será el *Líder* de inicio del consenso.
- *Acceptor*: Servidores que aceptan (o *votan* por) el requerimiento recibido.
- *Learner*: Proceso que ejecutará el requerimiento al recibir mensajes desde una mayoría de *acceptors*.

Es común que los roles *proposer*, *acceptor* y *learner* se implementen en el mismo proceso.



**Figura 4.1: Arquitectura de Paxos.**

**Definición:** Un *quorum* es un conjunto de una *mayoría* de nodos del sistema.

Paxos asegura las propiedades de *conditional liveness* si existen suficientes procesos sin fallas. También garantiza *safety* en el sentido que la misma operación será ejecutada (*learned*) por los procesos que no fallan.

El algoritmo se puede describir en los siguientes pasos.

#### Fase 1:

1. Un cliente envía a  $P$  una propuesta (u operación)  $p$ .

2. *Preparación*:  $P$  asume el rol de *proposer* y envía a un *quorum* de *acceptors*  $\text{PROPOSAL}(p, n)$  donde  $n$  es el *reloj lógico (escalar)* de  $P$ .

3. *Promesa*: Un *acceptor* puede recibir múltiples propuestas (de diferentes *proposers*). Sea  $\text{PROPOSAL}(p, n)$  recibida de  $P$ :

1.  $n$  es el mayor de todos los recibidos previamente: Envía  $\text{PROMISE}(p, n)$  a  $P$ , indicando que ignorará  $\text{PROPOSAL}(p', n')$  con  $n' < n$ .
2. Si ya había aceptado una  $\text{PROMISE}(p', n')$  con  $n' < n$  responde a  $P$   $\text{PROMISE}(p', n')$ .
3. Aún no aceptó  $(p, n)$  pero recibió previamente  $\text{PROMISE}(p', n')$  con  $n < n'$ , responde con  $\text{NACK}$  (negative ack).

### Fase 2:

1. *Aceptar*: Si  $P$  recibe  $\text{PROMISE}(q, m)$  de un *quorum* de *acceptors*, envía  $\text{ACCEPT}(q, m)$  a los *acceptors*. En otro caso, no hay concenso y se aborta el intento.
2. *Aceptada*: Cuando un *acceptor* recibe  $\text{ACCEPT}(q, m)$  de  $P$  y no había enviado  $\text{PROMISE}(q', m')$  con  $m' < m$ , envía  $\text{LEARN}(q)$  a los *learners* y  $\text{ACCEPTED}(q, m)$  a  $P$ . Sino, no hace nada, no participando del quorum de *acceptors*.
3. Un *learner* al recibir  $\text{LEARN}(q)$  desde los *acceptors*, procesa (o acepta)  $q$  y notifica al cliente.

## Análisis de Paxos

El protocolo dado asume que hay un único *leader*. En el caso que existan más de un *leader* puede suceder que *compitan* por un mismo  $\text{PROPOSAL}(p, n)$ , en cuyo caso podrán fallar los intentos de lograr concenso no garantizando *progreso* pero aún garantizando *safety*.

De hecho, es posible usar *Paxos* para *elección de líder* (donde  $p$  es *elect*), así permitiendo elegir un único *proposer*.

Es *tolerante a fallas* tanto en el *proposer*, *acceptors* y *learners*. Se deja como ejercicio, analizar diferentes escenarios de fallas.

## Raft

Una alternativa a Paxos es el algoritmo propuesto en 2014 por Diego Ongaro y Ousterhout conocido como *Raft* (por *Reliable, Replicated, Redundant and Fault-tolerant*).

Se basa (como Paxos) en lograr concenso basado en máquina de estado replicadas. Cada proceso contiene una máquina de transición de estados y un *log*. La máquina de estados representa el componente del sistema que deseamos que sea tolerante a fallas y aparece ante los clientes como una única máquina de estados aún en presencia de fallas minoritarias de nodos. El log actúa como una cola de comandos a ejecutar por la máquina de transición de estados.

El algoritmo se asegura que si una máquina ejecutó la operación *n*-ésima del log ninguna otra aplicará otro *n*-ésimo comando.

Un cluster Raft contiene un conjunto de servers. Cada server contiene su copia del log, que es una secuencia de pares `(cmd, term)`. El tiempo del sistema avanza por intervalos denominados `terms` (una especie de reloj lógico escalar).

Cada *server* en un momento dado puede estar en uno de los siguientes estados:

- *leader*: Maneja los requerimientos de clientes. Cualquier *server* que recibe un mensaje de un cliente, lo redirige al *leader*. El líder es quien inicia una transacción distribuida (*commit*) a los *followers*.
- *follower*: Responden a requerimientos del *líder* y de los *candidates*.
- *candidate*: Un server que detectó la falla del líder e inició una elección de líder.

Raft divide el tiempo en *terms*, numerados con enteros consecutivos. Cada *term* inicia una *elección de líder* y también actúa como un reloj lógico.

1. **Inicialización:** Los servers comienzan en el estado *follower*.
2. **Elección:** El líder envía periódicamente un mensaje `heartbeat(term)` a los *followers*. Cada *follower* luego de recibir un *heartbeat* reinicia su *timeout* con un valor aleatorio. Cuando un *follower*  $p_i$  no recibe un *heartbeat* antes de su *heartbeat timeout*, inicia una *elección*, cambiando su estado a *candidate*, incrementa su *term* y envía un mensaje `leader(term)` a los demás junto con su log para que algún follower no sincronizado actualice su copia.

Si recibe una mayoría de votos positivos se convierte en el nuevo *líder*, enviando periódicamente el *heartbeat*. Un *follower* vota positivamente si el *term* recibido del *candidate* es mayor al suyo y no ha respondido a otro *candidate* con ese *term*. En el caso que no reciba una mayoría de votos positivos, incrementa el *term*, espera un tiempo aleatorio y luego inicia una nueva elección. Si recibe un mensaje `append(entries)` con *term* mayor que el de él, pasa a estado *follower*, asumiendo que otro líder ganó una elección iniciada concurrentemente.

3. **Replicación del log:** El líder acepta un comando *cmd* de un cliente y envía un mensaje `append(entries)` a cada *follower*. El líder mantiene una variable *lastIndex* aceptado (*committed*) por cada follower. *entries* contiene la secuencia de entradas del log desde el *lastIndex*. Si el líder recibe una mayoría de mensajes `ok`, les envía `commit cmd` y actualiza el *lastIndex* de los *followers* que respondieron. Luego todos hacen persistente las entradas en el log.

La aceptación de un *commit* garantiza la siguiente propiedad global:

**Si dos entradas en la posición *i* en diferentes logs tienen el mismo *term*, entonces almacenan el mismo *cmd* y todas las entradas precedentes coinciden.**

En Raft, cada *follower* actualiza su log para satisfacer la propiedad anterior.

Si un *follower* falla, cuando se vuelva operativo, recibirá del líder las entradas desde su último *commit* aceptado (desde el punto de vista de este líder) y actualiza su log quedando en un estado consistente con el del líder.

Ante una falla del líder, la próxima elección seleccionará a un *candidate* con el máximo *term* y éste impondrá a los demás servers a sincronizarse con su log en el próximo mensaje `append(entries)`.

Para más detalles del algoritmo Raft, ver el artículo original disponible en <https://raft.github.io/raft.pdf>.

Es posible ver una explicación animada en <http://thesecretlivesofdata.com/raft/>.

## Otros algoritmos de consenso

Los algoritmos de consenso anteriores son adecuados para redes en donde se conoce el número de participantes.

En redes abiertas, donde pueden unirse nuevos participantes o algunos dejar de participar de manera imprevista, se requieren revisar la idea de cómo lograr consenso. Estos sistemas se conocen como *permissionless systems*.

En estos sistemas, lograr consenso es más difícil porque determinar una mayoría es comúnmente más difícil por el hecho que una máquina puede aparecer como muchas (usando diferentes IPs). Esto habilita un **\*Sybil attack** donde un nodo enmascara muchas identidades para ganar influencia y poder engañar a un algoritmo clásico de consenso o voto por mayoría.

La aplicaciones de *criptomonedas* como **Bitcoin** y **Ethereum**. Estos sistemas mantienen réplicas de un log de operaciones en un ambiente *abierto*, es decir que los nodos no son necesariamente *autenticados*. El objetivo es mantener un log (*machine*) replicado consistentemente en todos los nodos participantes de la red.

La consistencia se basa en un algoritmo de consenso que trata de impedir transacciones (*smart contracts*) inválidas como el pago doble.

La estrategia dominante en la actualidad es el uso de algunas técnicas criptográficas para sustituir mayorías expresándolas en forma de alguna capacidad indiscutible como poder de cómputo (*proof of work*) u otra.

El log es una secuencia de bloques  $b_0, b_1, \dots, b_{n-1}$  donde cada  $b_i$ , con  $0 > i < n$ , tiene la forma  $\langle header, transactions \rangle$ .

El header comúnmente contiene los siguientes campos:

- **Versión:** Número de versión del protocolo
- **Previous:** Identificador (hash) del bloque previo
- **Time:** Fecha y hora de creación (timestamp)
- **Nounce:** Valor usado por única vez en el log.
- **Hash:** Hash del bloque

El *hash* del bloque comúnmente se calcula desde los restantes campos del header y de las transacciones. El nombre *cadena de bloques* proviene de que el hash de cada bloque depende del anterior y provee la *intangibilidad* del log. Si un intruso modifica un bloque, deberá modificar todos los siguientes para que sea validado por otros participantes.

Un cliente (billetera o *wallet* en cripto-monedas) genera transacciones y se las envía a un grupo de nodos de la red (*mineros* o *validadores*). Un nodo recolecta un grupo de transacciones y propone un nuevo *bloque* a los demás para ser incorporado al log.

Cada réplica que recibe un bloque, si éste es válido, lo acepta agregándolo al log y cancela la recolección de transacciones y la construcción del bloque a proponer. Es común que se incluya una transacción de *recompensa*, el cual puede ser una *nueva criptomonedas* o *cargos* aplicados en las transacciones recolectadas.

Ante la recepción de bloques válidos en forma concurrente desde dos o más *mineros* (*proposers*) se crea una rama por cada bloque recibido. Luego, al recibir un siguiente bloque, éste pertenecerá a una una de las ramas. La rama que eventualmente se convierta en la mas larga y se convertirá en el log *definitivo* (*commit*). En la práctica, en un corto tiempo (dos o tres rondas de minado) se logra el consenso sobre la cadena mas larga.

La diferencia entre las cadenas de bloques con los algoritmos clásicos es que el líder (*minero* o *validador*) debe *demonstrar alguna capacidad* para proponer o validar un bloque ya que éstos no son *autenticados* (es una red *abierta*).

La *capacidad* de un minero/validador puede basarse en diferentes ideas como:

- *Proof of work*: Debe construirse un bloque de transacciones cuyo *hash* debe ser un valor menor que un valor de *dificultad*. Esto requiere encontrar un valor en el campo *nounce* de la cabecera del bloque para que el hash cumpla con lo requerido. Encontrar el *nounce* adecuado es un problema NP-hard, básicamente prueba y error (fuerza bruta). Este método requiere de un gran consumo de energía y se usa en [Bitcoin](#).
- *Proof of stake*: Los validadores deben *invertir* una cierta cantidad para calificar como tal. Un validador se selecciona aleatoriamente como el *block proposer* en cada período o *término* (*time slot*). En el caso que un validador opere incorrectamente (ataque o falla) se penaliza quitándole monedas desde su *stake*. Con cada validación obtiene una pequeña recompensa. Los validadores *votan* agregando el nuevo bloque al log. Comúnmente se requiere lograr más de 2/3 de los votos para aceptar un nuevo bloque. Este algoritmo se usa en [Ethereum](#).

El paper [Bitcoin: A Peer-to-peer Electronic Cash System](#) de un autor anónimo bajo el pseudónimo de Satoshi Nakamoto, describe el protocolo propuesto y usado en *bitcoin*.

En el próximo capítulo se describe en mayor detalle las técnicas conocidas como *blockchain* o *cadenas de bloques*.

## Consistencia, disponibilidad y particionado

En un sistema confiable sería deseable lograr consistencia, una propiedad de *safety*, es decir que nada malo sucederá y disponibilidad, una propiedad de *liveness*, es decir que eventualmente el sistema responderá.

Cuando un grupo de procesos no puede comunicarse entre sí, se dice que la red está *partida*. Algunos clientes pueden seguir operando sobre una partición y otros sobre las demás.

**Teorema CAP:** Un sistema distribuido con posibles fallas sólo puede garantizar dos de las tres siguientes propiedades:

1. *Consistencia*
2. *Disponibilidad (Availability)*
3. *Tolerante al Particionado (Partitioning tolerance)*

Esto significa que en una red sujeta a fallas de comunicación arbitrarias deberá resignar una de las tres propiedades.

Los algoritmos como Paxos o Raft garantizan consistencia y disponibilidad pero no toleran particionado.

El modelo *BASE* relaja los requisitos *CAP* y es un acrónimo de:

- *Basically-Available*: El sistema debería siempre responder con algún reconocimiento.
- *Soft-state*: Puede cambiar de estados al recibir nueva información.
- *Eventually-consistent*: Se toleran inconsistencias temporales pero garantiza consistencia eventual.

## Transacciones distribuidas

Un conjunto de operaciones que deben realizarse de manera atómica se denomina una *transacción* y tiene las siguientes propiedades (*ACID*):

1. *Atomicidad*: La secuencia de transacciones se consideran una operación indivisible, es decir se ejecutan todas o ninguna.
2. *Consistencia*: El sistema transiciona de un estado válido a otro.
3. *Aislamiento (Isolation)*: Los efectos de una transacción no deben afectar a otra. Esto requiere que si se realizan en un ambiente concurrente, debe haber control de la concurrencia.
4. *Durabilidad*: Cuando una transacción se completa, los cambios producidos se vuelcan en el próximo estado del sistema. A modo de ejemplo, en un sistema de bases de datos, en un *commit* los cambios pasan de la memoria temporal a la persistente.

Es común el uso de transacciones en operaciones de bases de datos para la gestión de operaciones concurrentes en un ambiente cliente-servidor.

## Two phase commit

Este algoritmo fue propuesto por Gray en 1978 y asume que el sistema es libre de fallas. Se basa en un nodo con el rol de *coordinador* y el resto son los *participantes*.

Suponiendo que algún cliente envió una secuencia de operaciones (como por ejemplo delimitadas por `begin transaction` y `end transaction` en SQL) al *coordinador* y éste replica las operaciones en cada *participante*. Posiblemente cada participante reciba parte de la transacción.

Cada participante almacena los resultados de las operaciones en una memoria temporal, conocido como el *log de transacciones*. El paso final es el *commit*, el cual dispara el siguiente algoritmo:

1. El *coordinador* envía a los *participantes* el mensaje `VOTE-REQUEST`.
2. Cada *participante* que recibe `VOTE-REQUEST`, responde con `VOTE-COMMIT` en caso que esté listo para realizar su *commit* local o `VOTE-ABORT` en otro caso.
3. El *coordinador* si recibe `VOTE-COMMIT` de todos los participantes les envía `COMMIT`, sino `ABORT`.
4. Cada participante que recibe `COMMIT` realiza su *commit* local, sino se eliminan las operaciones en la memoria transaccional (*log*).

# Recuperación

Cuando un sistema ha sufrido una falla es deseable que se pueda *recuperar*, es decir continuar funcionando desde un *estado anterior* consistente o tratar de realizar alguna transición a un *nuevo estado consistente*.

Es común usar un *log* de las operaciones (y datos) de la transacción. Al final se escribe un registro especial de *commit*.

Luego se aplica cada operación del log en el almacenamiento permanente finalmente el log se elimina.

El mecanismo de transacciones se utiliza comúnmente para implementar el primer enfoque tanto en bases de datos como en sistemas de archivos.

Puede ocurrir una falla en el sistema mientras se están aplicando las operaciones o antes de eliminar el log.

En el reinicio del sistema, si el log contiene un registro de *commit* al final, se aplica la transacción nuevamente. Si el log está incompleto, se ignora y se elimina, dejando al sistema en un estado consistente. Esto se corresponde a una operación *rollback*, ya que queda en un estado anterior consistente.

A modo de ejemplo, en un sistema de archivos, el borrar un archivo requiere liberar los bloques de datos (apagar los bits correspondientes en el *bitmap*), luego eliminar los metadatos del archivos (ej: los inodes) y finalmente eliminar la entrada en el directorio correspondiente. Si ocurre una falla entre esas operaciones el sistema, en el reinicio, el sistema de archivos quedaría en un estado inconsistente.

En sistemas distribuidos, el *roolback* puede requerir que cada *log* local contribuya a la consistencia global. Una técnica es utilizar *checkpoints consistentes* basado en *tomas de instantáneas* como el descripto.

# Detección de fallas

Un nodo puede detectar que otro falla ya sea porque no recibe respuestas o reconocimientos (*acks*) en un cierto intervalo de tiempo (*timeout*) o porque un nodo responde con valores incorrectos.

Comúnmente las aplicaciones se implementan haciendo que los nodos pueden periódicamente enviar *pruebas de vida* a otros nodos, conocidas como *heartbeat*, enviando una *prueba de vida*. La periodicidad de estos mensajes es importante para no sobrecargar la red con estos mensajes de control o que la detección no sea tardía.

La notificación al resto de los nodos de una detección de una falla puede hacerse utilizando *broadcasting*, o protocolos de *flooding* o *gossip*.

---

< Anterior

## Replicación y Distribución

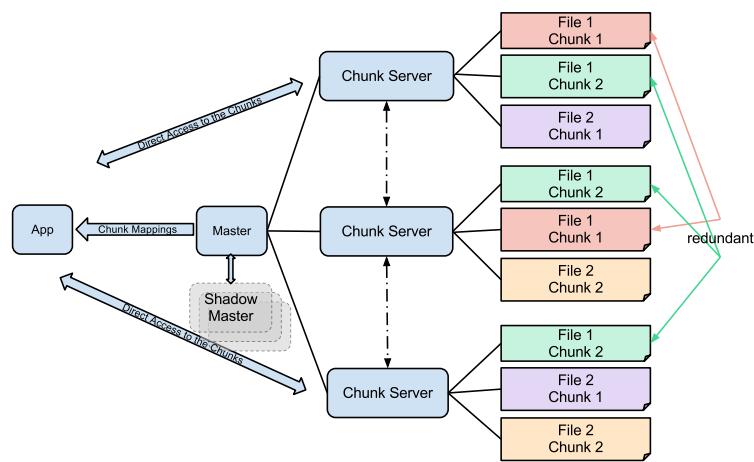
Próximo >

## Casos de estudio

# Casos de estudio

## Google file system

Todos los servicios (apps) de Google que requieren gestión de almacenamiento de archivos se basan en los servicios provistos por el **GFS**.



**Figura 5.1: Arquitectura de GFS.**

El sistema se basa en un conjunto de *clusters* interconectados. Los clientes (apps) se conectan a alguno de ellos.

La figura anterior muestra un esquema de su arquitectura. Cada archivo se partitiona en *chunks* de 64Mb. Cada *chunk* se replica en al menos tres *chunk servers*. Un nodo *master* contiene los metadatos de cada archivo, lo cual incluye la lista de *chunks* y los servidores que los contienen (*chunk mappings*).

Una aplicación (cliente) contacta inicialmente al *master* para obtener los metadatos de un archivo. Luego la aplicación puede requerir *chunks* desde cualquier *chunk server*.

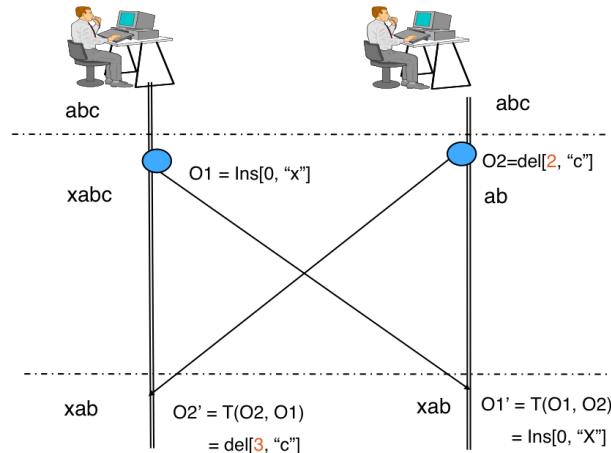
El nodo *master* comúnmente tiene réplicas para distribuir la carga de trabajo y proveer tolerancia a fallas.

[GFS paper](#).

# Edición de documentos en tiempo real

La edición simultánea de documentos es un servicio provisto por varias aplicaciones. Uno de los problemas a resolver es la edición en redes con alta latencia como Internet.

Una de las técnicas mas utilizadas es la *transformación de operaciones*, ilustrada en la siguiente figura.



**Figura 5.2: Ejemplo de transformación de operaciones.**

Las operaciones son secuencias de `insert(p, c)` y `delete(p, c)`, donde `c` es un carácter y `p` es una posición o índice.

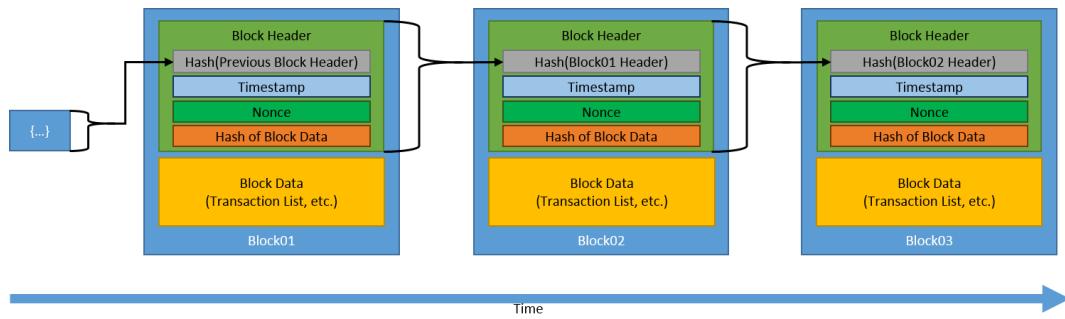
La idea consiste en que dada una operación ejecutada localmente se transmite a los demás editores. En la recepción de operaciones, cada proceso secuencializa las operaciones locales y remotas recibidas *modificando las posiciones* de algunas operaciones, tal como se muestra en la figura.

Dependiendo del ordenamiento usado, se puede lograr diferentes tipos de consistencias. Cada operación puede incluir un *timestamp* que determina el orden basado en relojes lógicos. Si se usan relojes vectoriales puede lograrse *consistencia secuencial*.

## Blockchain

Un *blockchain* es una secuencia de bloques de datos extensible (sólo permite operaciones *append*). Cada bloque comúnmente contiene *transacciones*, un *hash criptográfico* de los datos del bloque mas el hash del bloques anterior, y un *nounce* (valor usado por única vez).

El *hash* forma un *timestamp* distribuido ya que prueba que el bloque es el último.



**Figura 5.3: Secuencia de bloques (blockchain).**

Esta secuencia de bloques comúnmente representa un *log de transacciones* y se usa en *criptomonedas* como [bitcoin](#) y [ethereum](#) permitiendo replicar el *log* en una red *peer-to-peer* abierta usando un protocolo que garantiza su *inviolabilidad (integridad)*.

Cada bloque incluye *transacciones*. En el caso de criptomonedas, representan transferencias entre dos usuarios o cuentas. Cada transacción generalmente está firmada digitalmente por el *ejecutor* de la transacción.

Cada usuario, quien utiliza un programa cliente conocido como su *billetera (wallet) virtual*, tiene su par de claves pública y privada. La clave pública actúa como su identificador.

Los clientes generan transacciones que son comunicadas a los demás nodos de la red. Cada nodo recolecta las transacciones para ser incluidas en un nuevo bloque. Un bloque tiene un límite de transacciones.

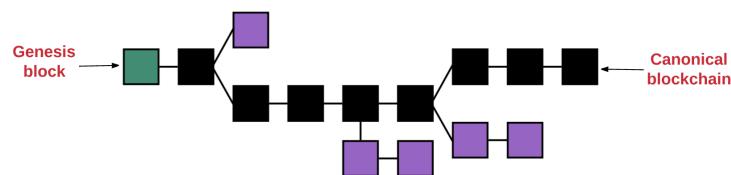
Recordar que una *firma digital* de un documento (transacción en éste caso) consiste en un *hash criptográfico* del mismo cifrado con la *clave privada del firmante*. Esto permite que se pueda validar descifrando la firma con la *clave pública del firmante* y comparando el hash obtenido con el cómputo del hash del documento. Si estos hashes coinciden, el documento es íntegro y ha sido emitido por el firmante.

En algunos frameworks de blockchain (Bitcoin, Ethereum, ...) las transacciones se generan en base a la ejecución de *scripts* definidas en la aplicación distribuida, conocidas como *dapps*. Estos frameworks también permiten definir *smart contracts*, los cuales se basan en eventos o tareas programadas que disparan la ejecución de *scripts* y permiten implementar las reglas (dependientes de la aplicación) del sistema.

Un protocolo basado en blockchain usa algún algoritmo de consenso por mayoría (quorum). Un nodo que ha recolectado un número dado de transacciones puede incluirlas en un nuevo bloque y proponiéndolo (broadcast) a los demás.

Cada nodo que recibe una propuesta, su aceptación se representa por la inclusión del nuevo bloque en su copia local del blockchain, previa verificación del bloque, lo cual incluye la validación de cada transacción y los metadatos (nro, nonce y hashes) del bloque.

En el caso que dos o más nodos propongan concurrentemente nuevos bloques a los demás *peers*, el bloque que posteriormente logre mayor aceptación gana. Un nodo que recibe más de un bloque válido concurrentemente, los incluye (acepta temporalmente) en diferentes *ramas* (*forks*). Luego, al recibir un nuevo bloque, éste se adiciona a la rama correspondiente (en base al hash del bloque anterior). Con el tiempo, luego de pocas rondas, la rama más larga ganará y se transforma en la *cadena canónica*, como se muestra en la siguiente figura.



**Figura 5.4: Diferentes ramas (temporales) en un blockchain.**

Los nodos que proponen nuevos bloques se denominan *mineros* ya que deben lograr generar un bloque cuyo valor *hash* tenga ciertas propiedades, típicamente una cierta cantidad de ceros al comienzo. Esto hace que un *minero* tenga que encontrar un *nonce* que permita lograr el hash requerido. Esta computación se logra mediante un algoritmo de *fuerza bruta* (prueba y error), por lo que esto se conoce como *prueba de trabajo*, dado que el nodo que propone un nuevo bloque válido ha invertido un considerable poder de cómputo para lograrlo.

En este modelo la aceptación de un nuevo bloque por el sistema no se logra por un sistema de votación, sino que se logra implícitamente por la adición del bloque en cada copia local de cada participante, logrando así mantener las réplicas sincronizadas.

La integridad del blockchain se logra ya que si un atacante quiere modificar algún bloque intermedio del blockchain, deberá modificar los subsiguientes, y deberá lograr su validación de la mayoría. En el caso de *proof of work* esto significa que deberá tener un poder de cómputo superior a la mayoría de los mineros. El modelo hace que no convenga ser un nodo malicioso, ya que si tiene suficiente poder de cómputo, es conveniente ser un minero lícito.

Cada minero al lograr que su bloque propuesto sea aceptado por el sistema, obtiene una recompensa, mediante la creación de nuevas monedas (si el sistema está aún en un estado inflacionario) y/o por el cobro de *comisiones* a los usuarios que participan en las transacciones incluidas en el bloque.

La técnica *blockchain* puede usarse en cualquier escenario que requiera un sistema de seguimiento de transacciones distribuidas en logs replicados inviolables como por ejemplo, sistemas de expedientes, trazabilidad de productos y otros.

Para lograr comprender mejor el funcionamiento de un *blockchain* ver los videos en [Blockchain demo](#) y experimentar en las diferentes secciones del sitio (hash, block, blockchain, ...).

## Algoritmos de consenso

El algoritmo descripto (conocido como *proof of work*) requiere grandes consumos de energía dado que se basa en una competencia de poder de cómputo entre los mineros. Existen alternativas de consenso basados en otros conceptos.

- *Proof of stake/importance (prueba de apuesta/importancia)*: Se basa en que un nodo tiene una mayor participación (confianza) basado en las cantidad de *tokens* que posee. En las cripto-monedas tiene la desventaja que induce a una estrategia de ahorro más que en operaciones de gastos.
- *Proof of Authority*: Se basa en que un conjunto de nodos son *validadores* que ganan su rol por medio de la *confianza* de los demás nodos. Se basa en una idea similar a *web o trust* usada en PGP aunque requiere una organización algo más centralizada.

- *Basados en Byzantine Agreement:* Algoritmos basados en consenso de mayoría (votación) como los estudiados previamente, como por ejemplo Paxos o algunas de sus versiones.

Se debe hacer notar que estos protocolos proveen *consistencia eventual temporal* ya que en un momento dado puede haber varias ramas diferentes en las réplicas. Si se toma en cuenta el blockchain descartando las posibles ramas temporales (vistas como un estado interno temporal) y asumiendo que la operación *write (append)* conceptualmente se realiza luego de la selección de la rama más larga, es posible concluir que ofrece *consistencia secuencial fuerte*.

## Referencias

- [Concurrency Control in Groupware Systems](#). Ellis, Gibbs. ACM 1989.
- [Bitcoin](#). Satoshi Nakamoto. Bitcoin whitepaper.
- [A survey of Consortium Blockchain Consensus Mechanisms](#). Wei Yao, et al. 2021.

---

< Anterior

Tolerancia a fallas

Próximo >

Especificación y Análisis

# Especificación y Análisis

En este capítulo se describen algunos métodos y formalismos para la especificación y análisis de sistemas distribuidos. Estos formalismos comúnmente se utilizan generalmente para la especificación de sistemas concurrentes, independientemente si se usan modelos de memoria compartida o pasaje de mensajes.

Existen numerosos enfoques y aquí se describen algunos de los mas utilizados como son los basados en *finite state machines (FSM)*, *lógicas temporales*, *redes de Petri* y *session types*.

## Máquinas de estados finitos

Los autómatas finitos (FSMs) se utilizan comúnmente para describir programas o máquinas de estados secuenciales y concurrentes. Una de las herramientas utilizadas en el ámbito académico es [Labeled Transition System Analyzer \(LTSA\)](#) que soporta modelos en Java. La herramienta se complementa con el libro[\[5\]](#) de los mismos autores.

Para modelar sistemas distribuidos se han propuesto varias extensiones. Una de las más utilizadas son el modelado con *máquinas de estado finitos comunicantes (CFSM)*. La idea del formalismo es describir un conjunto de sistemas de transición de estado que interactúan por medio de eventos/mensajes. Cada autómata representa un proceso del sistema.

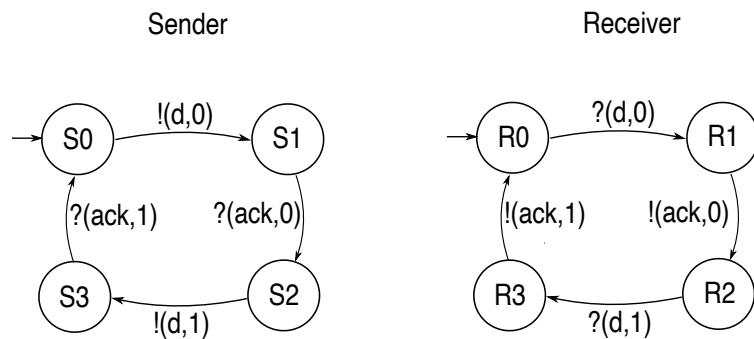
Formalmente, un *CFSM* representa un *protocolo* en el que intervienen  $N$  procesos  $P_1, \dots, P_N$  y se define como una tupla  $(S, I, M, T, F)$ , donde

- $S = \{S_1, \dots, S_n\}$  donde cada  $S_i$  ( $1 \leq i \leq N$ ) es el conjunto de estados del proceso  $i$ .
- $I = \{I_1, \dots, I_n\}$  donde cada  $I_i \in S_i$  ( $1 \leq i \leq N$ ) es el *estado inicial* de  $P_i$ .
- $M = \{M_1, \dots, M_n\}$  donde cada  $M_i$  ( $1 \leq i \leq N$ ) es el conjunto de *tipos de mensajes* que puede enviar  $P_i$ .
- $T = \{T_1, \dots, T_N\}$  donde cada  $T_i : S_i \times \cup_{j=1}^N (M_{j,i} \cup M_{i,j}) \rightarrow S_i$ , para  $1 \leq i \leq N$ . El conjunto denotado po  $M_{j,i}$  representa los mensajes

que pueden ser enviados de  $P_j$  a  $P_i$ .

- $F = \{F_1, \dots, F_N\}$  donde cada  $F_i$  ( $1 \leq i \leq N$ ) es el *conjunto de estados finales* de  $P_i$ .

La siguiente figura muestra una posible especificación del *two phase handshake protocol* simplificado en forma gráfica (similar al usados en autómatas finitos), donde los rótulos de los arcos con las formas  $!msg$  y  $?msg$  representan un *envío* o *recepción* de un mensaje, respectivamente. Cuando hay más de dos procesos involucrados las operaciones de envío y recepción pueden especificar el o los procesos de destino u origen de la forma  $P!msg$  y  $Q?msg$ , respectivamente.



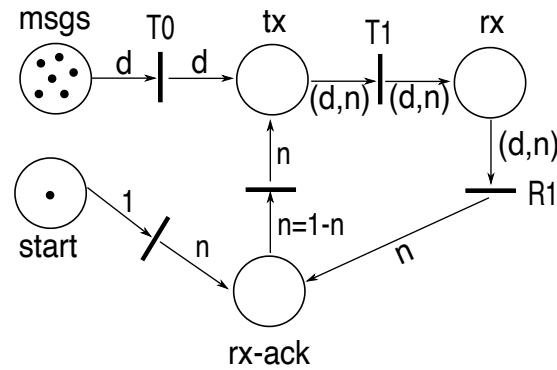
**Figura 6.1: Ejemplo de CFSM del *two-phase handshake protocol*.**

En este formalismo es posible analizar algunas propiedades automáticamente, como por ejemplo, *alcanzabilidad*, es decir, determinar si un estado es alcanzable desde otro y *deadlock*. También es posible determinar algunos tipos de *invariantes*.

## Redes de Petri

Este formalismo, propuesto en 1939 por Carl Adam Petri, forma parte de la clase de modelos de eventos discretos en sistemas dinámicos. Una *red de Petri* es un grafo *bipartito* con dos clases de nodos, *places* (*lugares*) y *transiciones*. Un *place* puede contener un número arbitrario de *tokens*. Una transición está *habilitada* si todos los *places* conectadas como *inputs* tienen al menos un token. Una transición al ser *disparada*. Un *disparo* consume *tokens* de cada *places* de entrada y genera *tokens* en su salida. Cada *arco* determina el número de tokens extraídos y generados, lo que se conoce como la *multiplicidad del arco* y en caso de omisión generalmente es 1.

Existen numerosas extensiones. Una de las mas utilizadas para modelar sistemas distribuidos son las *redes de Petri coloreadas (CPN)* que permiten que los tokens sean *rotulados* o *tipados* como se muestra en la siguiente figura.



**Figura 6.2: Two-phase handshake protocol modelado con Petri-net.**

Formalmente una red de Petri es una tupla  $(P, T, F, M_0, W)$  donde:

- $P$  es el conjunto de *places*.
- $T$  es el conjunto de *transiciones*.
- $P \cap T = \emptyset$
- $F \subseteq (P \times T) \cup (T \times P)$ : Es el conjunto de *arcos*.
- $M_0 : P \rightarrow \mathbb{Z}$  es la función de *marcado inicial*.
- $W : F \rightarrow \mathbb{Z}$  es la función de asignación de *pesos (multiplicidad)* a arcos.

En la figura 6.2 se muestra el marcado inicial con un conjunto de tokens representando el número de mensajes a transmitir en el *place msgs* y un token en el *place start* que *habilita* la transmisión del primer mensaje.

## Ejecución

La semántica de ejecución de una *Petri net N* se define por medio de una relación de transición entre estados entre dos *marcados*.

$M \xrightarrow{N,t} M'$  donde  $t$  es una *transición habilitada* por  $M$  y  $M' = M - W(p_s, t) + W(t, p_d)$  para cada  $p_s$  y  $p_d$  correspondientes a los *places* de origen y destino de  $t$ , respectivamente. Las operaciones (sobrecargadas) representan el efecto de la transición  $t$ , lo cual quita (-)  $W(p_s, t)$  tokens del *place*  $p_s$  y agrega (+)  $W(t, p_d)$  tokens al *place*  $p_d$ .

La relación  $M \xrightarrow{N} M'$  representa un paso (disparo) de cualquier transición habilitada. Se debe notar que una red de Petri es un modelo *no determinístico* ya que en un estado puede haber múltiples transiciones habilitadas.

Se denota como  $M \xrightarrow{N^*} M'$  la *clausura reflexo-transitiva* (cero o más pasos) de  $\xrightarrow{N}$ .

Una *secuencia de disparos* de una red de Petri  $N$  es una secuencia de transiciones  $< t_1 \dots t_n >$  tal que  $M_0 \xrightarrow{N, t_1} \dots \xrightarrow{N, t_n} M_n$ . El conjunto de todas las secuencias de disparo se denota como  $L(N)$ .

$R(N) = \{M \mid M_0 \xrightarrow{N^*} M\}$  es el conjunto de *markings alcanzados*.

## Análisis de Petri-nets

Las redes de Petri permiten construir un modelo ejecutable con el cual se puede experimentar (simular y probar) su comportamiento. Generalmente también se desea analizar propiedades generales y sus algoritmos de decisión como:

1. **Alcanzabilidad:** Determinar si un marking  $M' \in R(N)$  desde un marking dado. Recientemente (2021) se demostró que este problema es no decidible en general. En la práctica es mas común tratar de mostrar que no es posible alcanzar estados erróneos, lo cual puede realizarse utilizando *lógica temporal lineal* conjuntamente con el método *tableau*.
2. **Liveness (progreso):** Es posible demostrar diferentes niveles:
  - *dead*: No puede disparar desde un marking dado.
  - *L<sub>1</sub> – live*: Existe al menos un disparo en alguna secuencia de  $L(N)$
  - *L<sub>2</sub> – live*: Puede disparar eventualmente.
  - *L<sub>3</sub> – live*: Puede disparar eventualmente infinitamente.
  - *L<sub>4</sub> – live (live)*: Puede disparar siempre en cada estado posible.
3. **Boundeness (acotada):** Una Petri net se dice *k bounded* si no contiene markings posibles con más de *k* tokens. Esto es útil para analizar el uso y gestión de recursos.

Existen numerosas herramientas para la simulación y análisis de redes de Petri y es uno de los modelos más usados en la industria. Muchos lenguajes de especificación de sistemas concurrentes y distribuidos definen su semántica en términos de redes de Petri.

Entre las numerosas extensiones cabe mencionar las *timed Petri nets* que permiten que las transiciones se disparen en base a *eventos temporales*, útiles para modelar *timeouts* y otros escenarios. Las *Petri Nets Jerárquicas* permiten modelar sistemas modulares, haciendo que un *place* represente una *Petri net* o subsistema.

## Lógica Temporal

La lógica temporal contiene *modalidades* para predicar sobre estados en el tiempo (pasos discretos). La fórmula  $\Box F$  denota que  $F$  es verdadera siempre (en cada estado) y la fórmula  $\Diamond F$  especifica que  $F$  es *eventualmente* verdadera (en algún estado futuro). [Temporal Logic of Actions \(TLA+\)](#) es un lenguaje desarrollado por Leslie Lamport para la especificación de sistemas distribuidos. Además del lenguaje, el cual se basa en lógica temporal enriquecido con la definición de tipos de datos y especificación de propiedades a verificar.

Las lógicas temporales permiten una técnica para determinar semi-automáticamente si una fórmula dada es *satisfiable* conocida como *model checking*, el cual se basa en la generación de [autómatas de Büchi](#), los cuales son una versión de autómatas finitos sobre *entradas infinitas*, aunque muchas herramientas actuales de model-checking utilizan SAT solvers proposicionales.

TLA+ *toolset* es un conjunto de herramientas de soporte. Existen lenguajes de alto nivel (como PlusCal) que permite definir algoritmos (especialmente concurrentes y distribuidos) y es más adecuado para desarrolladores de software. Un algoritmo PlusCal se traduce a un modelo TLA+ sobre el cual se pueden realizar análisis de propiedades.

El siguiente listado muestra una especificación al estilo *TLA+* (con pequeños cambios tipográficos) del protocolo de ejemplo.

```

EXTENDS Naturals
CONSTANT Data
VARIABLES val, rdy, ack
TypeInvariant = val in Data and rdy in {0,1} and ack in {0,1}
Init = val in Data and rdy in {0,1} and ack == rdy
Send = rdy == ack and val' in Data and rdy'=(1-rdy) and UNCHANGE
Recv = rdy != ack and ack' = (1-ack) and UNCHANGED <val, rdy>
Next = Send or Recv
Spec = Init and [] [Next]<val, rdy, ack>

THEOREM Spec ==> [] TypeInvariant

```

La especificación define los *estados* `Init`, `Send`, `Recv` y `Next` del sistema.

La notación `v' = v` denota que `v'` es el *nuevo valor* de '`v`' (o su valor en el próximo estado).

La especificación `Spec` define que partiendo del estado inicial, siempre vale `Next`, lo cual representa que el sistema puede transicionar a los estados `Send` y `Recv` infinitamente. La notación `[Next]<val, rdy, ack>` es una abreviación de `Next or (val'=val and rdy'=rdy and ack'=ack)`, es decir que es válida una transición *epsilon* (dejando los valores sin cambiar).

## Session types

Otro enfoque para la especificación e implementación de sistemas distribuidos es el desarrollo de sistemas de tipos que permitan describir programas verificables por un compilador o intérprete de un lenguaje de programación.

El sistema de tipos conocido como *session types* permite definir un protocolo global y la lógica de cada uno de los procesos participantes (o roles). El sistema es verifiable (*type-checked*) automáticamente.

El sistema de tipos define los siguientes constructores de tipos:

- $send_{p,q}\tau$ : Envío de un mensaje de tipo  $\tau$  del rol  $p$  a  $q$ .
- $recv_{q,p}\tau$ : Recepción (en  $q$ ) de un mensaje de tipo  $\tau$  enviado por  $p$ .
- $offer_{p,q}\{t_1, \dots, t_n\}$ : El rol  $p$  selecciona (no determinísticamente) un sub-protocolo (tipo) y se lo comunica a  $q$ .

- $choose_{q,p}\{l_1 : t_1, \dots, l_n : t_n\}$ : El rol  $q$  selecciona la rama rotulada  $l_i$  en base al  $offer t_i$  recibido por  $p$ .
- $\epsilon$ : End protocol.

En los casos de protocolos con sólo dos roles, es posible omitir el emisor y receptor.

A continuación se muestra el protocolo global usado como ejemplo con *session types* donde **S** y **R** son los *roles* de los procesos *Sender* y *Receiver*.

**TPP:**  $send_{S,R}(\text{data}, \text{seq}); recv_{R,S}(\text{data}, \text{seq}); send_{R,S} \text{ ack}(\text{seq}); recv_{S,R} \text{ ack}(\text{seq})$ ; TPP

En este ejemplo el protocolo *TPP* es recursivo, lo cual permite especificar ciclos o comunicación *infinita*.

El protocolo global se puede *proyectar* (automáticamente) en los tipos de cada participante (rol).

```
S: send (data, seq); recv ack(seq); S
R: recv (data, seq); send ack(seq); R;
```

Estas expresiones sobre tipos pueden ser verificadas por un *type checker*. La verificación consiste en que los procesos siguen una coreografía que no lleva a *deadlock* y que los mensajes enviados y recibidos tienen los tipos esperados.

A continuación se muestran dos programas que implementan los tipos definidos.

<pre>Sender: S {     while true {         send (data, seq);         recv ack(n);         seq = 1 - seq;     } }</pre>	<pre>Receiver: R {     while true {         recv (data, seq);         send ack(seq);     } }</pre>
---	--

El *type checker* debe verificar que un proceso es *DUAL* del otro, usando las siguientes reglas:

1.  $dual(\epsilon) = \epsilon$
2.  $dual(send_{tx,rx}T; \alpha) = recv_{rx,tx}T; dual(\alpha)$
3.  $dual(recv_{rx,tx}T; \alpha) = send_{tx,rx}T; dual(\alpha)$
4.  $dual(offer_{p,q}\{l_i : t_i\}); \alpha = choice_{q,p}\{l_i : dual(t_i)\}; dual(\alpha)$  con  $(1 \leq i \leq n)$
5.  $dual(choice_{q,p}\{l_i : t_i\}); \alpha = offer_{p,q}\{l_i : dual(t_i)\}; dual(\alpha)$ , con  $(1 \leq i \leq n)$

Es fácil ver que cuando dos procesos son *duales* (o *co-tipos*) siguen los patones de comunicación adecuadamente, es decir que cuando uno envía un tipo de mensaje, el otro lo espera y viceversa, pero no describe la *lógica* del protocolo, es decir cómo computan o procesan los datos transmitidos y recibidos. Por ejemplo, no es posible especificar cómo transicional los números de secuencia entre los valores 0 y 1.

Es fácil de demostrar que el tipo `S = dual(R)` :

1. `dual(R) = dual(recv (data,seq); send ack(seq); R;) =`
2. `send (data,seq); dual(send ack(seq); R;) =`
3. `send (data,seq); recv ack(seq); dual(R;) =`
4. `send (data,seq); recv ack(seq); S = S`

A continuación se muestra un ejemplo mas interesante que contempla el caso de transmisiones con errores, en cuyo caso el receptor en lugar de responder con *ack(n)* responde con *error*. Esta especificación requiere el uso de *offer* y *choice*.

```
S: send (data,seq); choice {ack(n): S, error: end}
R: recv (data,seq); offer {send ack(seq); R, send error; end}
```

**Ejercicio:** Probar  $S = dual(R)$  (o viceversa).

Existen lenguajes y herramientas para especificar, verificar y generar código para diferentes lenguajes de programación basados en esta idea. Uno de ellas es **Scribble** basado en session types y *Pi calculus*. Este proyecto tiene herramientas para generar código (interfaces o esqueletos) para varios lenguajes de programación y puede *proyectar* el código de cada participante (rol) a partir de la *especificación global* del protocolo.

También se han implementado bibliotecas en algunos lenguajes de programación aprovechando las características de sus sistemas de tipos y mecanismos de meta-programación, como para Haskell, Ocaml, Scala y Rust. En algunos de estos lenguajes es necesario realizar algunas verificaciones en tiempo de ejecución (*runtime*) ya que en general se requiere un sistema de tipos *lineal* (en donde una variable puede modificarse una única vez). El sistema de tipos de Rust (*affine*) es muy adecuado para implementar este sistema de tipos y se han desarrollado varias bibliotecas.

## Referencias

1. Leslie Lamport. *Specifying Systems. The TLA+ Language and Tools for Hardware and Software Engineers*. 2003, Pearson Education. [Online](#) update (July, 2021).
2. N. Yoshida at al. *The Scribble Protocol Language*. [Scribble tutorial](#).
3. Kurt Jensen, Lars Cristensen. *Coloured Petri Nets. Modeling and Validation of Concurrent Systems*. Springer-Verlag. 2009. ISBN 978-3-642-00283-0. e-ISBN 978-3-642-00284-7.
4. N. Yoshida et al. *Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types*. [Download](#).
5. Jeff Magee, Jeff Kramer. *Concurrency. State Models & Java Programs*. Second Edition. Wiley.

---

< Anterior

## Casos de estudio