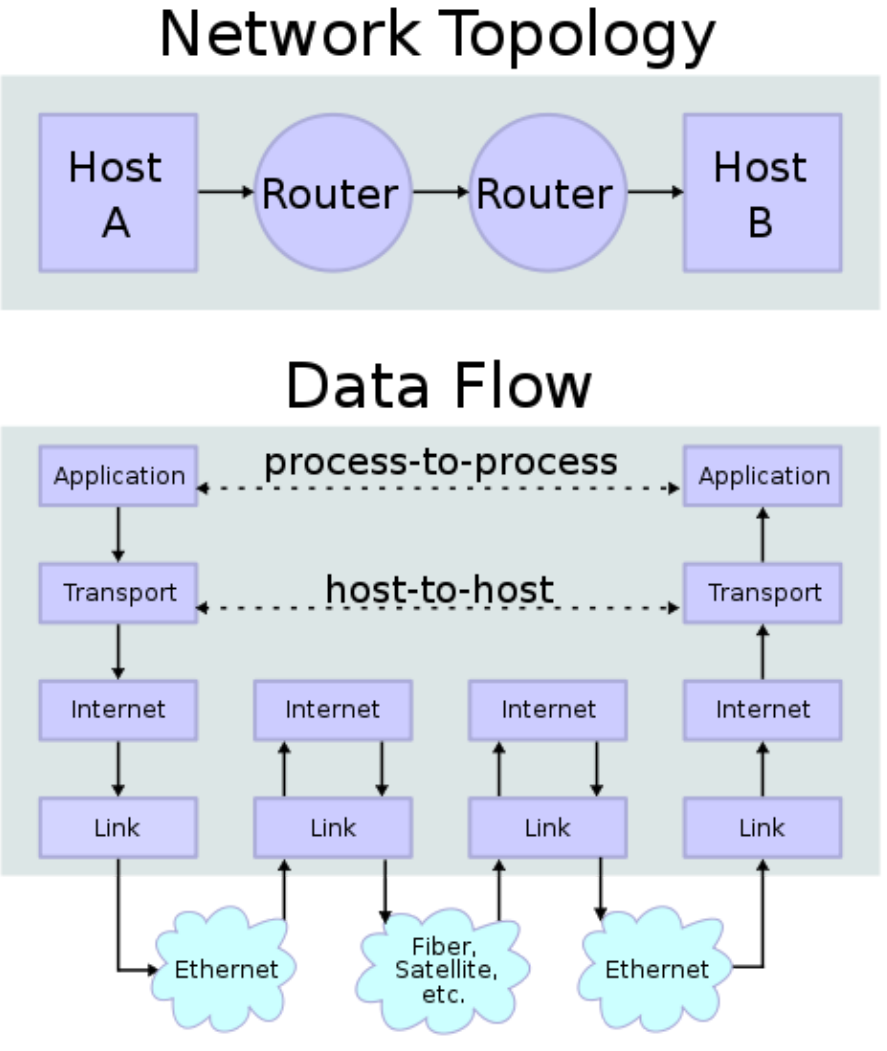
Tema 1. Introducción. Servicios básicos

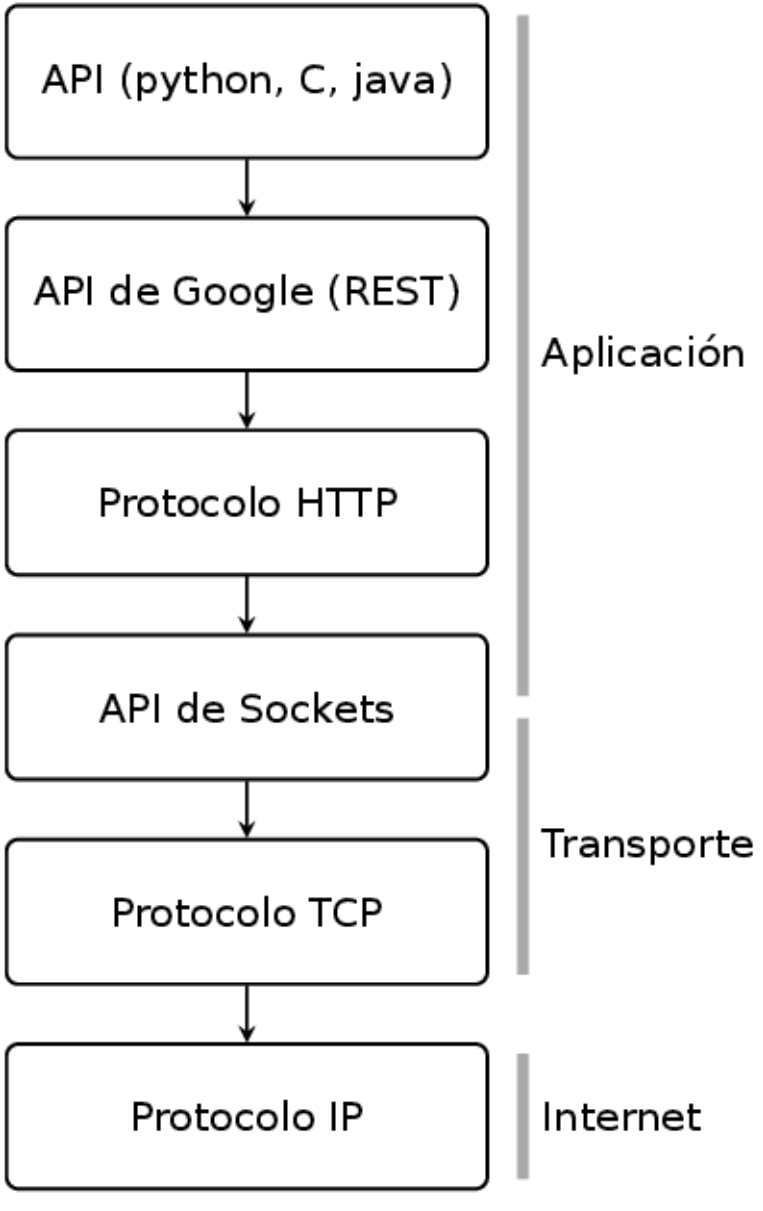
Ingeniería de Servicios

2023-2024

Los servicios se implementan en capas



...capas de bibliotecas y protocolos



APIs y capas

Un ejemplo ilustrativo:

> Queremos usar un servicio (en este caso de LocationIQ) para escribir un cliente que convierta el nombre de una calle en sus coordenadas de latitud, longitud

Por ejemplo, se usaría así:

```
$ python3 Geolocalizar.py
Dame el nombre de la calle: Viesques, Gijón
Sus coordenadas son: (43.5283176, -5.6456681)
```

> Documentación de la API:
> https://locationiq.com/docs

Parámetros de la consulta

La invocación de la API consiste en hacer un `GET` (HTTP) a la URL:

```
http://us1.locationiq.com/v1/search.php?<parameters>
```

Donde:

- `parameters` es una lista de `nombre=valor`, separados por `&`, en la que al menos deben aparecer:
  - `format` con el valor `json` o `xml` (indica el formato de la respuesta)
  - `q` con el nombre del objeto a geolocalizar (calle, ciudad, etc)
  - `key` una cadena que indentifica a quién facturar la consulta

EJEMPLO:

http://us1.locationiq.com/v1/search.php?format=json&q=Madrid&key=pk.dc05...

Nivel 0: Usando la API de sockets

Usamos un socket para conectar al servidor de Google (puerto 80) y "hablar" directamente HTTP con él:

```
# Secuencia de operaciones para conectar al servidor
import socket
s = socket.socket()
s.connect(("us1.locationiq.org", 80))
peticion = """cierto comando HTTP

""".encode("utf8")
s.send(peticion)
respuesta = s.recv(5000)   # 5000 = bytes esperados
print(respuesta.decode("utf8"))
```

Nivel 0: Completando el ejemplo (mal)

```
# Intento (mal) de pedir geolocalización de una calle
host = "us1.locationiq.org"
calle = "Campus de Viesques, Gijón"

peticion = "GET "
peticion += "/v1/search.php?key=pk.dc05..."
peticion += "&format=json&q=" + calle
peticion += " HTTP/1.1\r\n"
peticion += "Host: " + host + "\r\n\r\n"

import socket
s = socket.socket()
s.settimeout(2)
s.connect((host, 80))
a_enviar = peticion.encode("utf8")
s.send(a_enviar)
respuesta = s.recv(32000)
print(respuesta.decode("utf8"))

# Se recibe respuesta, a pesar de que la petición no es HTTP válido
```

Nivel 0: ¿por qué ha fallado?

La URL que hemos solicitado vía `GET` no tiene sintaxis válida (contiene espacios, tildes).

Hay una función en python para convertirla al formato válido:

```
# Cómo codificar correctamente una URL

calle = "Campus de Viesques, Gijón"
import urllib.request, urllib.parse, urllib.error
print(urllib.parse.quote(calle))
```

Nivel 0: Completando el ejemplo (mejor)

```
import urllib.request, urllib.parse, urllib.error
host = "us1.locationiq.org"
calle = "Campus de Viesques, Gijón"

peticion = "GET "
peticion += "/v1/search.php?key=pk.dc05..."
peticion += "&format=json&q=" + urllib.parse.quote(calle)
peticion += " HTTP/1.1\r\n"
peticion += "Host: " + host + "\r\n\r\n"

print(peticion)

import socket
s = socket.socket()
s.settimeout(2)
s.connect((host, 80))
s.send(peticion.encode("utf8"))
respuesta = s.recv(32000)
print(respuesta.decode("utf8"))
```

Problemas de la solución anterior

- Necesitamos conocer la sintaxis HTTP para enviar una petición correcta
- La respuesta del servidor incluye cabeceras HTTP que necesitaríamos saber interpretar
- La respuesta del servidor puede venir por trozos y requerir varios `recv`
- ¿Cómo saber cuántos `recv` son necesarios? De nuevo interpretando las cabeceras previas
- Tras superar estas dificultades, tendremos un `json`, que habría que interpretar también

Analicemos el JSON con alguna herramienta para descubrir dónde está la información que buscamos (las coordenadas)

Nivel 1: biblioteca que maneje HTTP

Podemos abstraernos de la API de sockets y "saltar" directamente a la capa HTTP.

La biblioteca `urllib` (incluida en python3) sirve para ello.

- Codificando URLs (`urllib.parse.urlencode)`
- Manejando toda la comunicación HTTP (`urllib.request.urlopen`)

```
import urllib.request, urllib.parse, urllib.error
import http.client

calle = "Campus de Viesques, Gijón"
url = "/v1/search.php?key={}&format={}&q={}".format(
    "pk.dc05...",
    "json", urllib.parse.quote(calle))

servidor = http.client.HTTPConnection("us1.locationiq.org")
servidor.request("GET",  url)
respuesta = servidor.getresponse().read()

print(respuesta.decode("utf8"))
```

Problemas de la solución anterior

- La interfaz de `urllib` es poco amigable
- No funcionaría correctamente si el protocolo incluyera *cookies*, *cachés*, *proxies* u otros elementos HTTP que compliquen el escenario.
- La respuesta aún está en `json` y necesita ser interpretada

> Del último punto puede ocuparse el módulo `json`

```
# ...
respuesta = urllib.request.urlopen(url).read().decode("utf8")

# Hasta aquí es como antes. Ahora, usamos json
import json
datos = json.loads(respuesta)
coordenadas = float(datos[0]["lat"]), float(datos[0]["lon"])
print(coordenadas)
```

Nivel 2: biblioteca más potente para HTTP

La biblioteca `requests` soluciona los problemas del caso anterior:

- Interfaz amigable
- Uso sencillo también en escenarios más complicados
- Trae un *parser* `json` incorporado

```
import requests

calle = "Campus de Viesques, Gijón"
parametros = {"key": "pk.dc05...",
              "format": "json", "q": calle}
resultado = requests.get(
    "https://us1.locationiq.org/v1/search.php?",
    params=parametros
)
datos = resultado.json()
coordenadas = float(datos[0]["lat"]), float(datos[0]["lon"])

print(coordenadas)
```

Problemas de la solución anterior

- Es una biblioteca de terceros (no viene con python)
  - Instalarlo es fácil: `pip install requests`
  - Pero mejor hacerlo en un entorno virtual `virtualenv`
- Sigue siendo necesario que el programador conozca la especificación de la API de google
  - Para construir la URL con la petición
  - Para encontrar lo que queremos dentro del json

Nivel 3: biblioteca de cliente para esa API concreta

- Una biblioteca que dialogue directamente con el servidor ocultaría el hecho de que por debajo hay HTTP, URLs y JSON
- El propio proveedor del servicio proporciona una biblioteca para Python (`locationiq`)

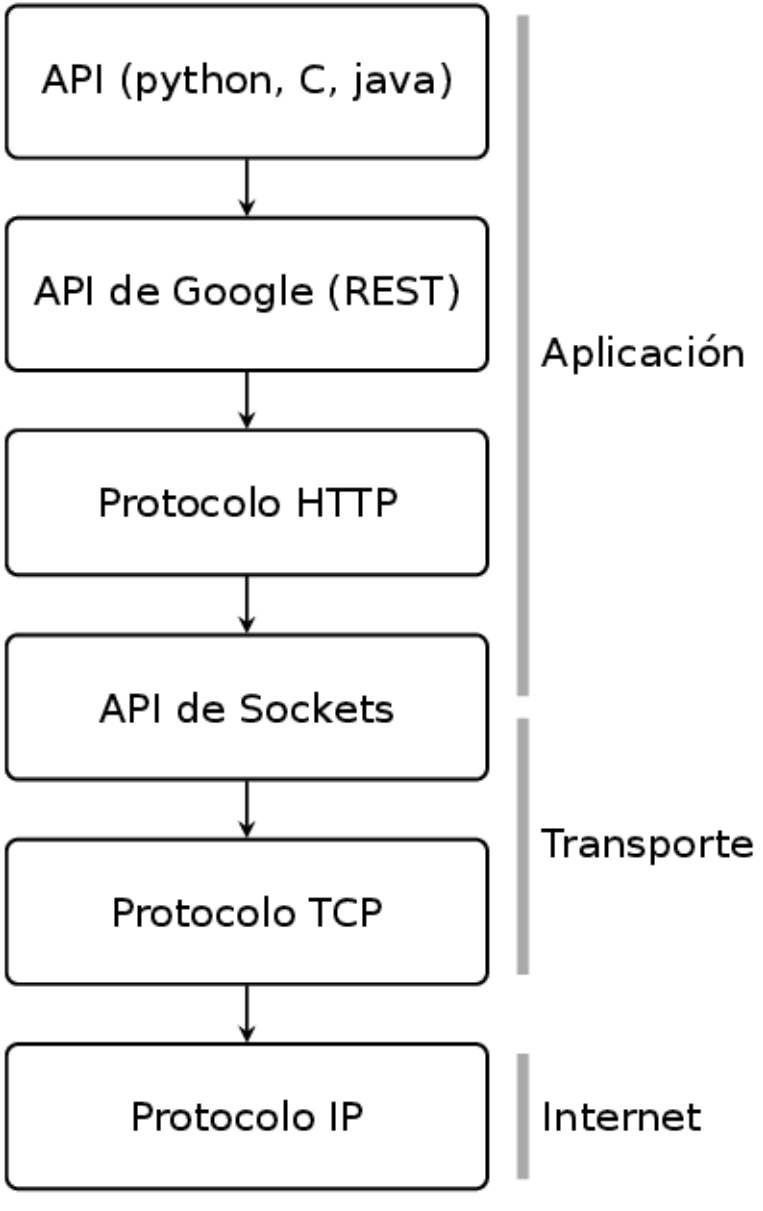
```
from locationiq import (Configuration, ApiClient,
                        SearchApi)

config = Configuration()
config.api_key['key'] = "pk.dc05..."

api = SearchApi(ApiClient(config))
result = api.search("Campus de Viesques, Gijón",
                    "json", 1)
coord = float(result[0].lat), float(result[0].lon)

print(coord)
```

...capas de bibliotecas y protocolos



En esta asignatura...

- Repasaremos brevemente el nivel de internet
- Veremos los protocolos ofrecidos por el nivel de transporte
- Y trabajaremos fundamentalmente con protocolos y bibliotecas en el nivel de aplicación

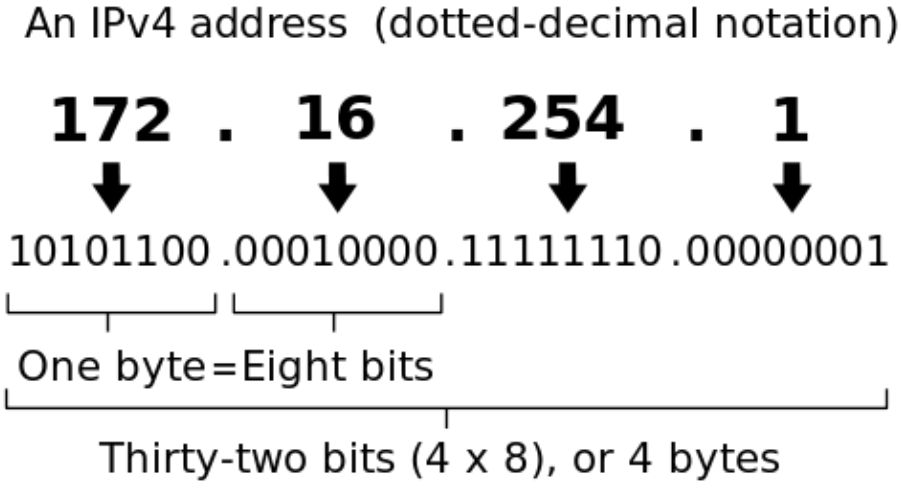
Nivel de Internet (IP)

IP

- A este nivel cada máquina se identifica por una IP
  - IP v4 → 32 bits
  - IP v6 → 128 bits
- El nivel se ocupa de hacer llegar un **paquete** de un nodo a otro, *enrutándolo* por los nodos intermedios necesarios.
- El paquete transportable por IP tiene un tamaño máximo de 64k.
  - La red bajo IP puede limitar más el tamaño del paquete (MTU)
  - IP puede fragmentar o descartar el paquete que no quepa en la MTU.
  - El paquete puede perderse y no llegar.

Direcciones IP (v4)

- Las direcciones son 32 bits. Típicamente se agrupan de 8 en 8.



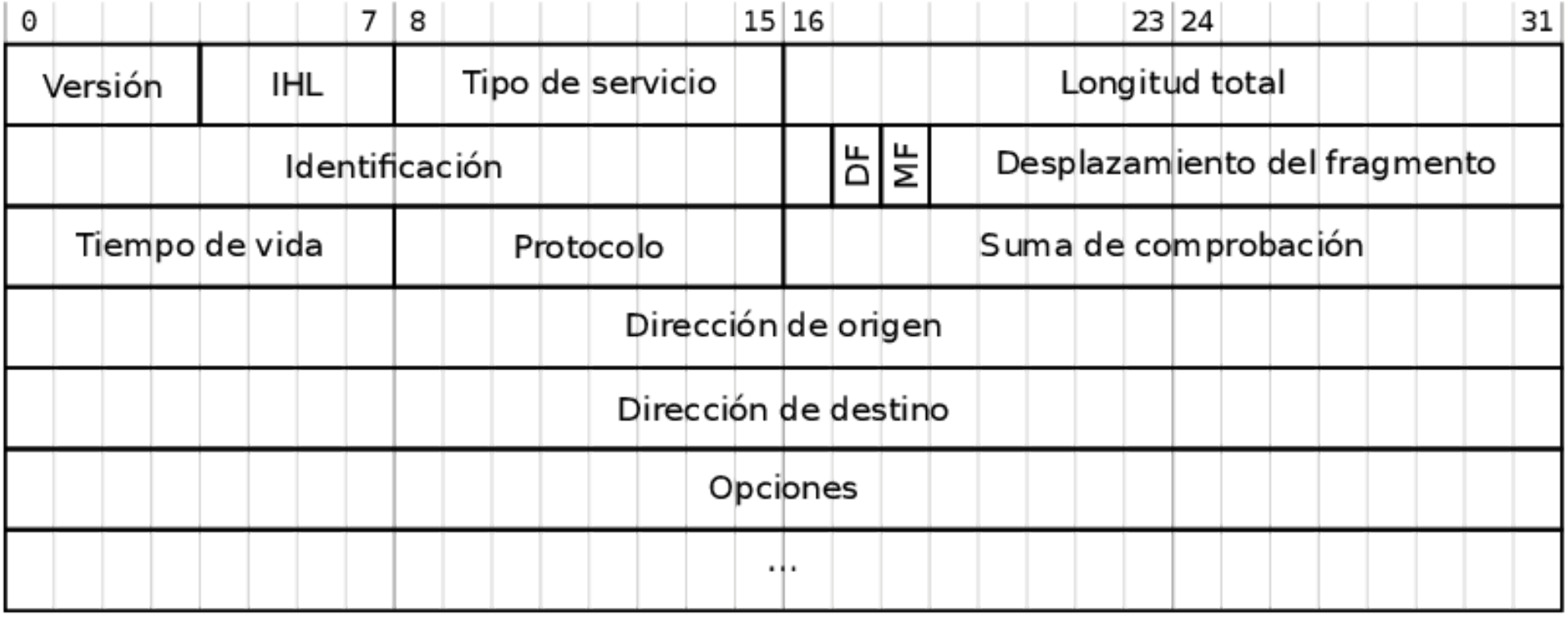
Direcciones especiales

Algunos rangos están reservados para usos especiales.

Rango	Uso
`192.168.0.0/16`	Direcciones privadas locales
`172.16.0.0/12`	más direcciones privadas locales
`10.0.0.0/8`	más direcciones privadas locales
`127.0.0.0/8`	direcciones locales
`0.0.0.0/8`	broadcast
`224.0.0.0/4`	multicast
`240.0.0.0/4`	uso futuro

hay más...

Recordatorio

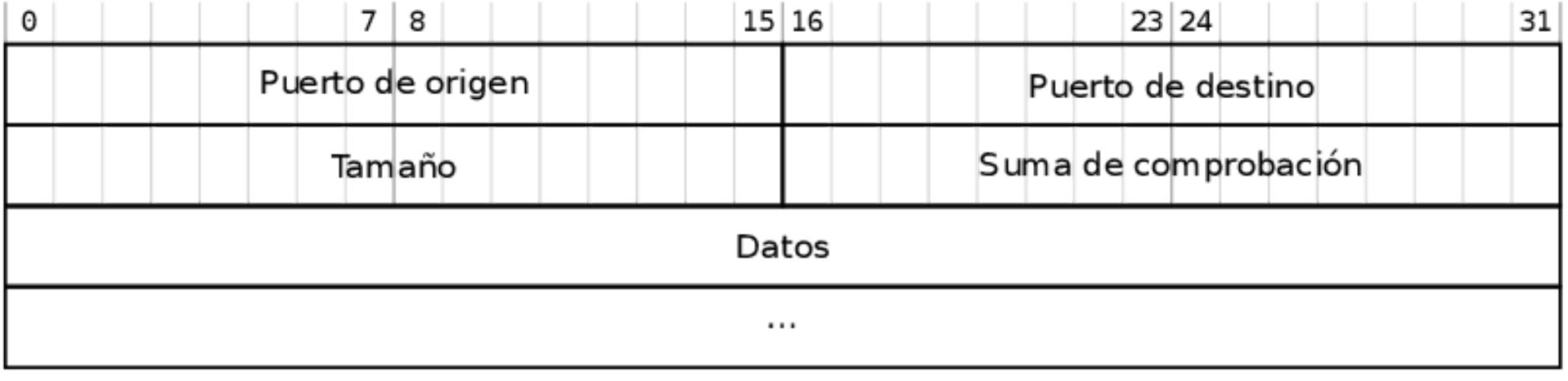


Nivel de Transoporte (UDP)

Transporte: UDP

UDP es un protocolo simple de transporte sobre IP. Añade:

- Números de **puerto** para identificar un (proceso) destino dentro de la máquina destino.
- Otra suma de comprobación (opcional)



Problemas con UDP

Tene **carencias**

- ¿Protocolo de turnos? (quién inicia la comunicación)
- El primero en comunicar ¿cómo conoce la IP y puerto de destino?
- ¿Qué números de puerto son válidos?
- Máximo tamaño de datagrama: 64k (en realidad 65507) ¿si se necesita más?
- ¿Cómo influye un MTU menor?
- Si el datagrama no llega ¿cómo saberlo?
- Orden de paquetes no garantizado

Cómo resolver algunas de ellas

- **Turnos:** definir un protocolo que indique quién debe comenzar con `sendto()`
- **IP y puerto:**:
  - Configuración por línea de comandos o fichero de opciones
  - Configuración automática (DHCP)
  - Descubrimiento por *broadcast*
- **Números de puerto válidos:** Estandarización (RFCs, IANA)
  - Puertos *bien conocidos*: 0 -- 1023 (protegidos)
  - Puertos registrados: 1204 -- 49151 (estandarizados pero no protegidos)
  - Restantes: 49152 -- 65535 (libres)

Puertos bien conocidos

Están asociados a protocolos concretos (generalmente sobre TCP, pero algunos también sobre UDP).

Algunos ejemplos de **puertos bien conocidos**

Puerto	Protocolo	Descripción
7	ECHO	"Eco" para comprobar la comunicación
21	FTP	Transferencia de ficheros
22	SSH	Comunicación cifrada (para terminal remota o transferencia)
23	TELNET	Conexión con terminal remoto (sin cifrar)
25	SMTP	Transferencia de correo electrónico hacia el servidor
80	HTTP	Web
110	POP3	Transferencia de correo electrónico desde el servidor
...	...	etc.

Confirmación de recepción

- Un datagrama de respuesta por cada datagrama enviado
- Fijar un *timeout*
- Repetición del envío si se agota el *timeout*
  - Evitar colapsar la red
  - Intervalos entre repeticiones crecientes
- Filtrar dirección desde donde llega recepción
- Numerar cada datagrama
  - Para casar respuesta con petición
  - Para descartar duplicados

Fragmentación (a nivel IP)

- Si un datagrama no cabe en el MTU,
  - El protocolo IP lo romperá...
  - ...y reconstruirá
  - pero si un fragmento se pierde, el datagrama completo se invalida
- Aumentan las probabilidades de perder datagramas.

Broadcast (difusión)

UDP permite enviar un mensaje *broadcast*:

- Es recibido por todos los nodos en el mismo segmento de red
- **Para recibirlo** es necesario poner el *socket* en modo *broadcast* (`setsockopt()`)
- **Para enviarlo** se usa como dirección de destino una especial (se averigua con `ifconfig`)

Ejemplo de broadcast

Recepción

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
...
# 1) Preparar el socket en modo broadcast
s.setsockopt(sockt.SOL_SOCKET, socket.SO_BROADCAST, 1)
# 2) Asignarle un número de puerto, p.ej, 9999
s.bind(("", 9999))
# 3) Esperar mensajes
datos, origen = s.recvfrom(65535)  # Tamaño máximo esperado
```

Envío

```
# Para enviar
s.setsockopt(sockt.SOL_SOCKET, socket.SO_BROADCAST, 1)
# Y simplemente se envían a la dirección de broadcast de la red
# y al puerto en que espera el servidor
red = "192.168.1.255" # (por ejemplo)
s.sendto(bytes("Mensaje enviado por broadcast", "ascii"), (red, 9999))
```

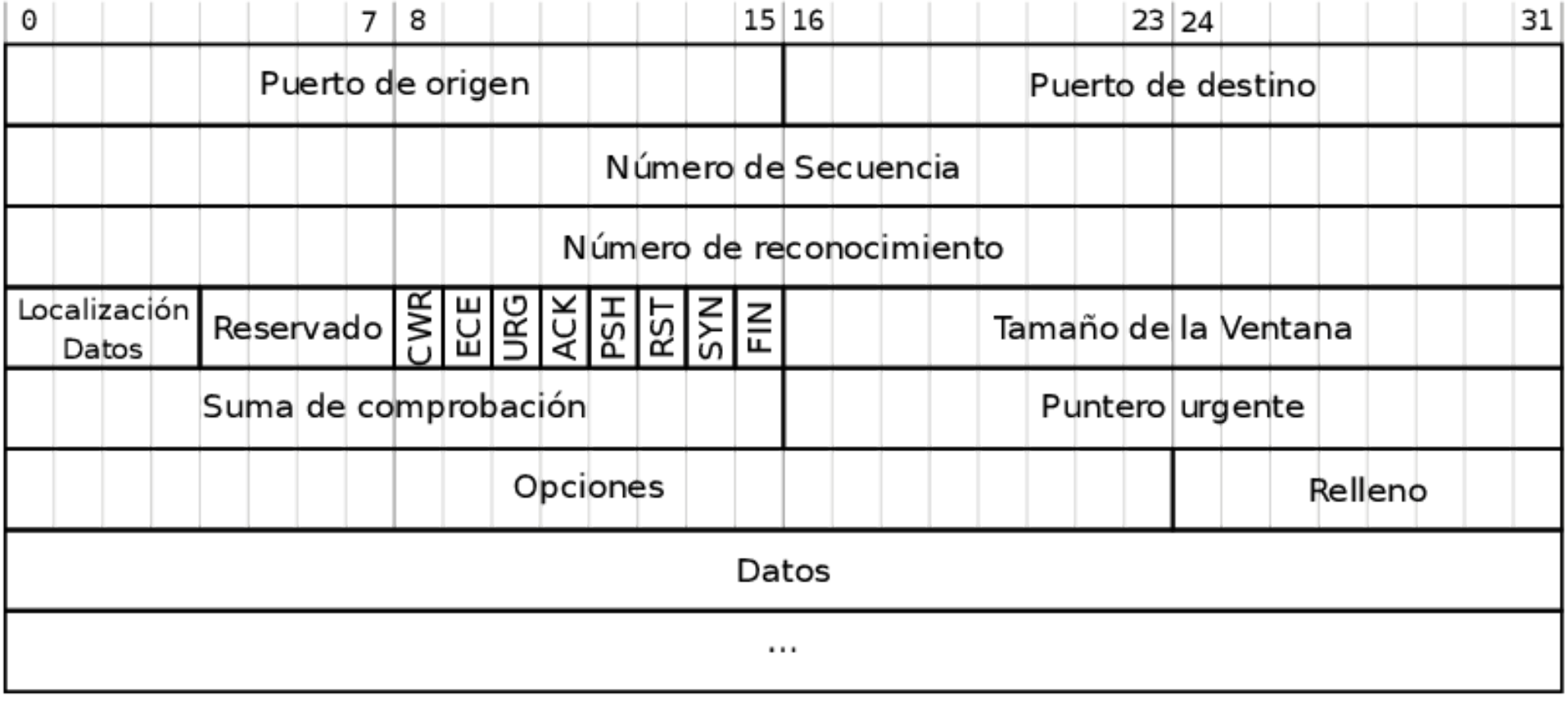
Nivel de transporte (TCP)

TCP

Suple las carencias de UDP, implementando:

- Confirmación de recepción de cada paquete
- Detección de errores (basado en *checksum*)
- Autonumeración de paquetes, que permite:
  - Reensamblar datos en orden correcto
  - Reenviar paquetes de los que no hubo confirmación, o se detectó error
- Control de flujo, basado en *ventana deslizante* (el extremo que recibe puede variar la velocidad del que envía, o detenerlo)
- Autodetección de congestión (y reducción de la velocidad)

Esto requiere más información en cada paquete



Como resultado TCP proporciona un **canal virtual** por el que se puede transmitir un **flujo de bytes**.

Nuevo problema: *framing*

Si lo que se recibe es un *flujo* de bytes,

¿Cómo sabe el receptor dónde termina un mensaje y empieza el siguiente?

Recordar que:

- Un `recv()` puede retornar menos datos de los enviados por el otro extremo.
- Por tanto pueden ser necesarios varios `recv()` hasta recibir todo el mensaje.
- Pero si se hace un `recv()` cuando el mensaje ya estaba completo, el receptor se bloqueará.
- ¿Cómo saber cuándo parar de hacer `recv()`?

Observar que este problema **no existe en UDP** (cada datagrama va completo)

Soluciones

1. Cerrar el socket tras enviar el último byte

- El receptor lo detecta porque `recv()` retorna 0 bytes
- Puede cerrarse sólo en un envío, para permitir la recepción de una posible respuesta
- Válido para protocolos muy simples

Soluciones

2. Usar mensajes de tamaño prefijado (por el protocolo)

- Un estándar o convenio define el tamaño del mensaje
- El receptor sabe que ha terminado cuando ha recibido ese número de bytes
- Válido para protocolos muy simples. Poco habitual.

¿Y si el emisor no envía todos los bytes que componen ese entero?

El receptor quedaría bloqueado...

Soluciones al *framing*

3. Usar un byte especial como "terminador"

- Por ejemplo, el carácter `\n` o `\0`
- Esto implica que ese byte no puede formar parte del mensaje
- En caso de que pueda aparecer en el mensaje, debe ser "marcado" (códigos de escape)

Soluciones al *framing*

4. Especificar longitud

- Delante del mensaje el emisor envía un entero que indica su longitud.
- Este entero debe ser delimitado a su vez de alguna forma
- Típicamente suele ser de tamaño fijo (ej: 4 bytes)

¿Y si el emisor desconoce de antemano la longitud del mensaje?
Por ejemplo, lo está leyendo de otro flujo.

Soluciones al *framing*

5. Enviar el mensaje en varios "trozos"

- Cada uno de ellos con su longitud como prefijo.
- Un último bloque de longitud cero (sólo contendría el entero con la longitud, igual a cero) marcaría el final.
- Es la solución más flexible.

Soluciones al *framing*

Técnicas híbridas (mezcla de las anteriores)

Por ejemplo HTTP

- Usa la técnica del delimitador para la cabecera (el delimitador es la secuencia `\r\n\r\n`)
- Y la técnica de enviar primero la longitud para el cuerpo (campo `Content length`)

Tipos de datos estructurados

- El dato usa un lenguaje de representación que deja claro dónde termina.
- Ejemplo: XML, JSON.

Concurrencia

Para atender varios clientes a la vez existen diferentes alternativas:

- Creación de más procesos servidores (`fork()`)
- Creación de varios hilos trabajadores (*threads*)
- Uso de multiplexación (`select()`, `poll()`)

Si el número de clientes es muy grande (*C10k problem*) la creación de procesos o hilos es prohibitiva

Debe usarse por tanto multiplexación, pero la lógica del programa se complica (patrón reactor, programación orientada a eventos, *futures*, *promises*, *corutinas*, ...)

Servicios

Un servicio:

- Es proporcionado por la capa de aplicación.
- Implementado sobre TCP o UDP (u otros protocolos específicos)
- Define e implementa su propio protocolo (generalmente estandarizado en un RFC)
- El servicio puede ser usado:
  - Por el usuario final
  - Por otro servicio (servicios básicos)

Servicios orientados al usuario:

- HTTP (Navegación web)
- SMTP, POP3, IMAP (correo electrónico)
- TELNET, SSH, RDP, RFB (terminal remota)
- FTP, SFTP, WebDAV, Bittorrent (transferencia de ficheros)
- RTSP (*streaming* de medios)
- IRC, SIP, XMPP (mensajería interactiva)
- etc...

Veremos varios de ellos en temas futuros

Servicios básicos

Funcionamiento de la red:

- **DHCP**: Asigna direcciones IP automáticamente
- **NAT**: Permite conectar una sub-red privada a la red exterior, sin consumir IPs públicas
- **SNMP**: Monitorización de la red. Detección de problemas

Nombres:

- **DNS**: Permite asignar nombres lógicos a los nodos y resolverlos en IPs
- **LDAP**: Servicio de directorio (listado de nombres, personas, organizaciones, etc)

Otros:

- **NTP**: Sincronización con la hora exacta.

DHCP

Dynamic Host Configuration Protocol

El protocolo tiene lugar en cuatro fases, cada una de las cuales se concreta en un datagrama UDP: Descubrimiento, Oferta, Solicitud, y Reconocimiento (**D**iscovery, **O**ffer, **R**equest, **A**cknowledgement).

Discovery. El cliente:

- Usa un *broadcast* UDP al puerto 67 para **descubrir** un servidor
- En el datagrama puede especificar qué IP prefiere (la última usada)
- Queda a la espera de respuestas en el puerto UDP 68

Offer. El servidor:

- Maneja un *pool* de IPs e información de configuración (DNS, router de salida, etc.)
- Elige una IP para el cliente
- Envía una **oferta** al cliente conteniendo la IP elegida

DHCP (sigue)

Request. El cliente:

- Si no recibe respuestas puede reintentar el paso 1
- Puede recibir varias ofertas. Cada una lleva un *identificador del servidor* que la hizo.
- El cliente se queda con una (y sólo una)
  - Crea un datagrama de **solicitud** que contiene el *id del servidor* que hizo la oferta elegida
  - Envía por *broadcast* dicho datagrama

Acknowledgement. El servidor, al recibir un datagrama de solicitud:

- Si el *id* no era el suyo, ignora el datagrama y devuelve la IP ofrecida al *pool*
- Si lo era, da la IP por ocupada durante un tiempo (*leasing*)
- Envía un datagrama de **reconocimiento** al cliente, indicándole el *leasing* y otra información de configuración.

DHCP liberación

- Cuando una máquina va a apagarse, debería liberar su IP
- Para ello el protocolo prevé un paquete específico: DHCP **Release**
- El servidor retornará la IP al *pool*

Sin embargo un cliente puede apagarse "de golpe" sin ocasión de enviar ese detagrama

- Por ello no es mandatorio el enviarlo
- Y por eso hay un tiempo de *lease* máximo

NAT

NAT (*Network Address Translation*)

Es la **modificación de la IP de un paquete** a su paso por un *router*.
(También será necesario cambiar los *checksums* y otras partes de la cabecera)

- En su versión más simple (*one to one*), una IP puede cambiarse por otra
- En su versión más habitual (*one to many*), todo un conjunto de IPs (privadas) se cambian por una (pública)

Cualquier paquete que salga de la IP privada llevará como IP origen la del router

Pero entonces, es el router quien recbirá las respuestas

¿Cómo sabe a qué máquina de la subred privada debe redirigir el paquete respuesta?

NAT (cont.)

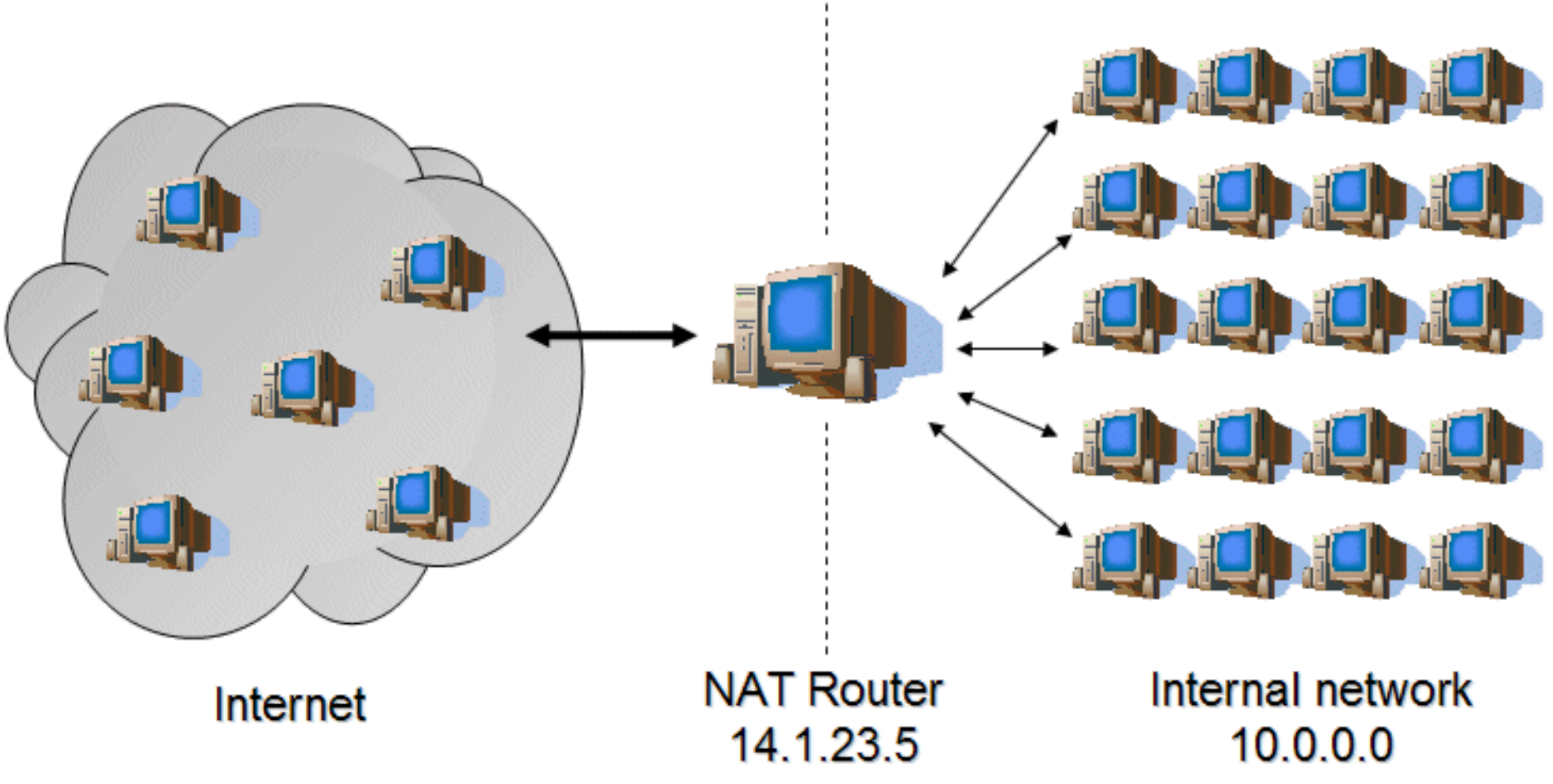
El método más habitual consiste en:

- El router cambia la IP origen (llamémosla IP_O) por la suya propia (IP_R)
- Y cambia también el puerto origen (incialmente N) por otro (pasa a ser M)
- Mantiene una tabla que relaciona el nuevo puerto M con el antiguo origen $IP_O{:}N$
- Cuando el router recibe un paquete dirigido $IP_R{:}M$, cambia la IP y puerto destino por $IP_O{:}N$
- Y así llega la respuesta a la máquina correcta

Este método se denomina NAPT o también simplemente NAT.

NAT (cont.)

Esto permite tener muchas máquinas clientes que "desde fuera" usan una sola IP



DNAT

¿Y si hay servidores dentro de la red privada?

No serían accesibles desde el exterior...

DNAT (*Destination Network Address Translation*) es la solución para este problema.

- El router se configura de modo que ciertos puertos sean redirigidos a ciertas IPs internas
- Por ejemplo, el puerto 80 se redirige al servidor web de la red privada
- Esto hace accesible ese servidor desde fuera, a través de la IP externa del router.

Esta técnica también se denomina redirección de puertos (*port forwarding*) o mapeo de puertos (*port mapping*).