Sistemas distribuidos

Grado en Ingeniería Informática

Tema 2.2: Comunicación entre procesos

Departamento de Ingeniería Informática
Universidad de Cádiz

Escuela Superior de Ingeniería
Dpto. de Ingeniería Informática

Versión 1.2

Indice

Sección 1 | Introducción

Introducción (I)

- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.
- Las entidades que se comunican son procesos cuyos papeles determinan cómo se comunican ("patrones de comunicación").
- La aplicación se comunica con:
  - UDP con "paso de mensajes".
  - TCP con "flujo de datos".

Introducción (II)

- Los patrones de comunicación principales son:
  - Comunicación cliente-servidor.
  - Comunicacion en grupo (multidifusión).
- Es necesario el diseño de protocolos de alto nivel que soporten dichos patrones.
- Tipos de comunicación:
  - Recurso compartido.
  - Paso de mensajes.

Sección 2 | API para los Protocolos de Internet

Comunicación entre procesos (I)

- El paso de un mensaje se puede llevar a cabo mediante 2 operaciones de comunicación:
  - *send*: Un proceso envia un mensaje a un destino.
  - *receive*: Un proceso recibe el mensaje en el destino.
- Cada destino tiene asociada una cola de mensajes:
  - Los emisores añaden mensajes a la cola.
  - Los receptores los extraen.
- Características de la comunicación entre procesos:
  - Comunicación síncrona y asíncrona.
  - Destinos de los mensajes.
  - Fiabilidad.
  - Ordenación.

Comunicación entre procesos (II)

Comunicación síncrona

- El emisor y el receptor se sincronizan en cada mensaje.
- *send* y *receive* son operaciones bloqueantes:
  - El emisor se bloquea hasta que el receptor realiza la operación *receive*.
  - El receptor se bloquea hasta que le llegue un mensaje.

Comunicación entre procesos (III)

Comunicación asíncrona (I)

- La operación *send* es no bloqueante:
  - El mensaje se copia a un búfer local.
  - El emisor continúa aunque todavía no exista un *receive* (bloqueante o no bloqueante).
- *Receive* no bloqueante:
  - El proceso receptor sigue con su programa después de invocar la operación *receive*.
  - Proporciona un búfer que se llenarán en segundo plano.
  - El proceso debe ser informado por separado de que su búfer ha sido llenado (sondeo o interrupción).

Comunicación entre procesos (III)

Comunicación asíncrona (II)

- *Receive* bloqueante:
  - En entornos que soportan múltiples hilos en un proceso:
    - Este *receive* puede ser invocado por un hilo mientras que el resto de hilos del proceso permanecen activos.
    - Simplicidad de sincronizar los hilos receptores con el mensaje entrante.
- El *receive* no bloqueante es más eficiente, pero más complejo (necesidad de capturar el mensaje entrante fuera de su flujo de control).

Comunicación entre procesos (IV)

Destino de los mensajes

- Los mensajes son enviados a direcciones construidas por pares (dirección Internet, puerto local).
- Un puerto local:
  - Es el destino de un mensaje dentro de un computador (número entero).
  - Tiene exactamente un receptor pero puede tener muchos emisores.
  - Los procesos pueden utilizar múltiples puertos desde los que recibir mensajes.
  - Cualquier proceso que conozca el número de puerto puede enviarle un mensaje.

Comunicación entre procesos IV)

Fiabilidad

- Comunicación punto a punto fiable:
  - Se garantiza la entrega, aunque se pierda un número razonable de paquetes.
- Comunicación no fiable:
  - La entrega no se garantiza, aunque sólo se pierda un único paquete.

Ordenación

Algunas aplicaciones necesitan que los mensajes sean entregados en el orden de su emisión.

Sockets (I)

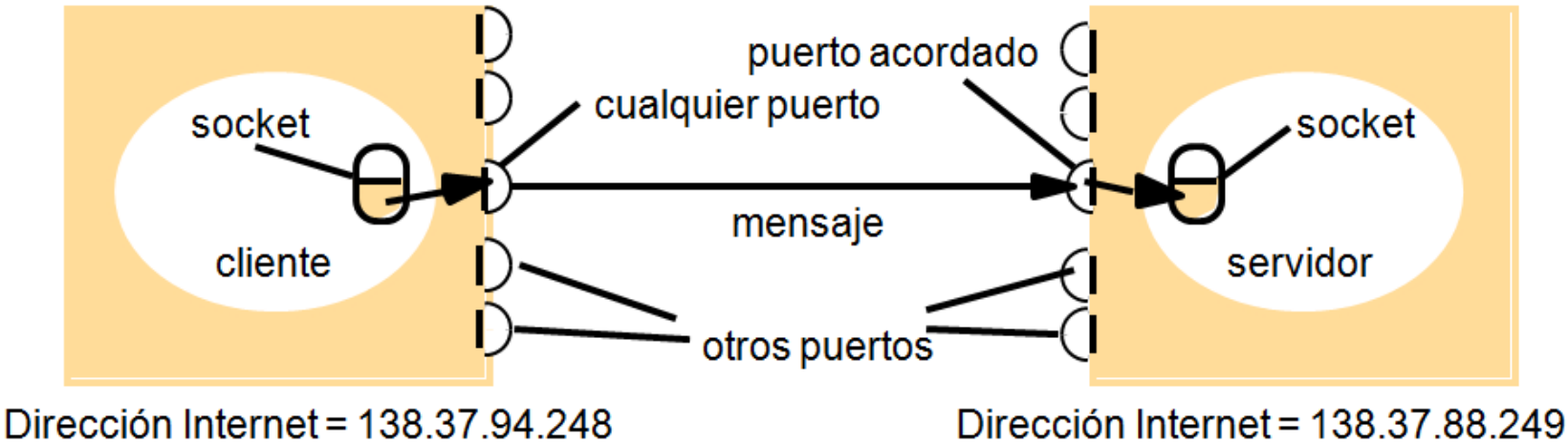
- El mecanismo original de *Unix Interprocess Communication* (IPC) son *pipes*:
  - Cauce unidireccional (flujo), sin nombre.
  - Enlazan filtros, sin sincronización explícita.
- *pipeline*: un mismo padre crea los procesos filtro y *pipes*. Ej.: *gunzip -c fichero.tar.gz | tar xvf -*.
- Útil en entornos productor-consumidor.
- No es útil en sistemas distribuidos:
  - No hay un nombre.
  - Sin posibilidad de enviar mensajes discretos.
- IPC implementada como llamadas al sistema, a partir de UNIX BSD 4.2.

Sockets (II)

- *Socket* (destino de mensajes):
  - Se pueden enviar mensajes mediante un socket.
  - Y permite recibir mensajes.
- La IPC se da entre 2 sockets.
- Cada socket debe estar asociado a:
  - Un puerto local de su máquina.
  - Una dirección IP de esa máquina.
  - Un protocolo (UDP o TCP).
- La interfaz de los sockets proporciona un conjunto de funciones para la comunicación por paso de mensajes:
  - Basada en el modelo cliente/servidor.
  - Tiene 2 servicios en TCP/IP:
    - Mensajes de conexión (TCP).
    - Mensajes sin conexión (UDP).
  - Interfaces: Sockets (Unix) y WinSock (Windows).

Sockets (III)

El proceso que posee el socket es el único que puede recibir mensajes destinados al puerto asociado.



Sockets (IV)

Interfaz Sockets: funciones UDP

- Cliente:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enviar/recibir.
  - Cerrar socket.

Sockets (V)

Interfaz Sockets: funciones TCP

- Cliente:
  - Crear socket.
  - Conectarse.
  - Enviar/recibir.
  - Cerrar socket.
- Servidor:
  - Crear socket.
  - Enlazar a *IP* y *port* (*bind*).
  - Escuchar.
  - Aceptar conexiones.
  - Enviar/recibir.
  - Cerrar socket.

Comunicación de datagramas UDP (I)

- Mensaje autocontenido no fiable desde un emisor a un receptor.
- Único mensaje sin asentimientos ni reenvíos (no hay garantía de entrega).
- Trasmisión entre 2 procesos: *send* y *receive*.
- Cada proceso debe crear un socket y enlazarlo a un puerto local:
  - Los clientes deben enlazarlo a cualquier puerto local libre.
  - Los servidores, a un puerto de servicio determinado.
- La operación *receive* entrega el mensaje transmitido y el puerto al que está enlazado el socket emisor.
- Se utiliza comunicación asíncrona con *receive* bloqueante:
  - *send* devuelve el control cuando ha dirigido el mensaje a las capas inferiores UDP e IP (responsables de su entrega en el destino).
  - Estas capas lo transmiten y lo dejan en la cola del socket asociado al puerto de destino.

Comunicación de datagramas UDP (II)

- Se utiliza comunicación asíncrona con *receive* bloqueante (cont.):
  - *receive* extrae el mensaje de la cola con un bloqueo indefinido (por defecto), a menos que se haya establecido un tiempo límite (*timeout*) asociado al conector.
  - En general, se puede enviar y recibir de cualquier puerto, aunque se puede limitar a uno concreto.
- No hay garantía de la recepción:
  - Los mensajes se pueden perder por errores de *checksum* o falta de espacio.
  - Los procesos deben proveer la calidad que deseen.
- Se puede dar un servicio fiable sobre uno no fiable, si se añaden asentimientos.
- No se suele utilizar una comunicación totalmente fiable:
  - No es imprescindible.
  - Provoca grandes cargas administrativas:
    - Almacena información de estado en origen y destino.
    - Transmite mensajes adicionales.
    - Puede existir latencia para emisor o receptor.

Comunicación de flujos TCP (I)

- La abstracción de flujos (*streams*) oculta:
  - Tamaño de los mensajes: los procesos leen o escriben cuanto quieren y las capas inferiores (TCP/IP) se encargan de empaquetar.
  - Mensajes perdidos: a través de asentimientos y reenvíos.
  - Control de flujo: evita desbordamiento del receptor.
  - Mensajes duplicados y/o desordenados: a través de identificadores de mensajes.
  - Destinatarios de los mensajes: tras la conexión, los procesos leen y escriben del cauce sin tener que utilizar nuevamente sus respectivas direcciones.
- Se distinguen claramente las funciones del cliente y servidor:
  - Cliente: crea un socket encauzado y solicita el establecimiento de una conexión.
  - Servidor: crea un socket de escucha con una cola de peticiones de conexión, asociado a un número de puerto y se queda a la espera de peticiones de conexión.

Comunicación de flujos TCP (II)

- Al aceptar una conexión el servidor:
  - Se crea automáticamente un nuevo socket encauzado conectado al del cliente.
  - Cada proceso lee de su cauce de entrada y escribe en su cauce de salida.
  - Si un proceso cierra su socket, los datos pendientes se transmitirán y se indicará que el cauce está roto.
- Puede haber bloqueo:
  - Lectura: no hay datos disponibles.
  - Escritura: la cola del socket de destino está llena.
- Opciones para atender a múltiples clientes:
  - Escucha selectiva.
  - Multitarea.
- La conexión se romperá si se detectan errores graves de red.

Sección 3 | Ejemplos de programación TCP/UDP

API Python (I)

Módulo *socket*

- Este módulo proporciona acceso a la interfaz de sockets de BSD (Berkeley sockets).
- Está disponible en todos los sistemas Unix modernos, Windows, MacOS y probablemente en otras plataformas.

Familias de sockets

- Dependiendo del sistema y de las opciones de construcción, este módulo admite varias familias de sockets.
  - `AF_INET` comunicación con IP v4.
  - `AF_INET6` comunicación con IP v6.
  - `AF_UNIX` comunicación entre procesos en el mismo host, siendo una alternativa ligera a un socket INET mediante loopback.
  - `AF_TIPC` un protocolo de red abierto, no basado en IP, diseñado para su uso en entornos informáticos agrupados.
  - `AF_BLUETOOTH` comunicación con un controlador Bluetooth.

API Python (II)

Tipos de sockets

- `SOCK_STREAM` proporciona flujos de bytes secuenciados y bidireccionales con un mecanismo de transmisión de datos de flujo. Este tipo de socket transmite datos de forma fiable y en orden. Se implementa en el protocolo Transmission Control Protocol/Internet Protocol (TCP/IP).
- `SOCK_DGRAM` proporciona datagramas, que son mensajes sin conexión de una longitud máxima fija. Este tipo de socket se utiliza generalmente para mensajes cortos, como un servidor de nombres o un servidor de tiempo, porque el orden y la fiabilidad de la entrega de los mensajes no están garantizados. Se implementa en el protocolo User Datagram Protocol/Internet Protocol (UDP/IP).

API Python (III)

Creación socket TCP IPv4

```
socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Creación socket UDP IPv4

```
socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

- La función `socket()` devuelve un objeto `socket` cuyos métodos implementan las diversas llamadas al sistema socket.
- Los tipos de parámetros son algo más de alto nivel que en la interfaz de C: al igual que con las operaciones de read() y write() en archivos de Python, la asignación de búfer en las operaciones de recepción es automática, y la longitud del búfer es implícita en las operaciones de envío.

API Python (IV)

Métodos básicos del objeto `socket`:

- `.bind()` se utiliza para asociar el socket con una interfaz de red y un número de puerto específicos.
- `.listen()` permite que un servidor acepte conexiones.
- `.connect(addr)` conecta con un socket remoto en la dirección `addr`.
- `.accept()` acepta una conexión. Devuelve una tupla `(conn, addr)` donde `conn` es un nuevo objeto `socket` para enviar y recibir datos en la conexión, y `addr` es la dirección vinculada al socket en el otro extremo de la conexión.
- `.close()` marca el socket como cerrado.

API Python (V)

- `.recv(bufsize)` recibe datos del socket. Devuelve los datos recibidos.
- `.recvfrom(bufsize)` Recibe datos del socket. El valor de retorno es una tupla `(bytes, addr)` donde `bytes` representa los datos recibidos y `addr` es la dirección del socket que envía los datos.
- `.send(bytes)` envía datos al socket. El socket debe estar conectado a un socket remoto. Puede enviar menos bytes de los solicitados, pero devuelve el número de bytes enviados.
- `.sendto(bytes, addr)` envía datos al socket. El socket no debe estar conectado a un socket remoto, ya que el socket de destino está especificado por la dirección `addr`.
- `.sendall(bytes)` envía datos al socket. El socket debe estar conectado a un socket remoto. A diferencia de `send()`, este método continúa enviando datos hasta que se hayan enviado todos los datos o se produzca un error.

Ejemplo UDP Python

Servidor

```
import socket
localIP = "127.0.0.1"
localPort = 20001
bufferSize = 1024
msgFromServer = "Hello UDP Client"
bytesToSend = str.encode(msgFromServer)
# Create a datagram socket
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
# Bind to address and ip
UDPServerSocket.bind((localIP, localPort))
print("UDP server up and listening")
# Listen for incoming datagrams
while(True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]
    clientMsg = "Message from Client:{}".format(message)
    clientIP  = "Client IP Address:{}".format(address)
    print(clientMsg)
    print(clientIP)
   # Sending a reply to client
    UDPServerSocket.sendto(bytesToSend, address)
```

Ejemplo UDP Python

Cliente

```
import socket
msgFromClient = "Hello␣UDP␣Server"
bytesToSend = str.encode(msgFromClient)
serverAddressPort = ("127.0.0.1", 20001)
bufferSize = 1024
# Create a UDP socket at client side
UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)
# Send to server using created UDP socket
UDPClientSocket.sendto(bytesToSend, serverAddressPort)
msgFromServer = UDPClientSocket.recvfrom(bufferSize)
msg = "Message␣from␣Server␣{}".format(msgFromServer[0])
print(msg)
```

Ejemplo TCP Python

Servidor

```
import socket
# Standard loopback interface address (localhost)
HOST = "127.0.0.1"
# Port to listen on (non-privileged ports are > 1023)
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Ejemplo TCP Python

Cliente

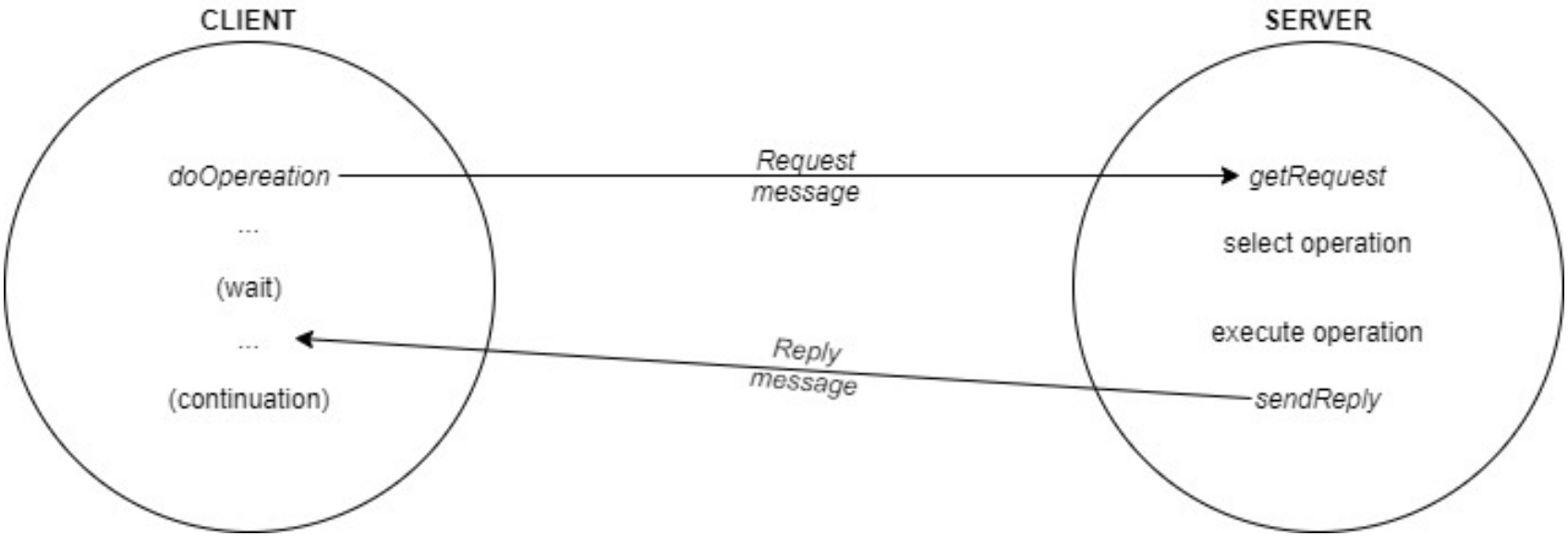
```
import socket

# The server's hostname or IP address
HOST = "127.0.0.1"
# The port used by the server
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

Sección 4 | Comunicación cliente-servidor

Protocolo petición-respuesta (I)

- Protocolo:
  - Identificadores de mensaje: tipoMensaje (0=petición, 1=respuesta), idPetición (int), referenciaObjeto (RemoteObjectRef), idMétodo (int o Method) y argumentos (cadena de bytes).
  - UDP.
- Modelo de fallos:
  - Tiempo de espera límite.
  - Eliminación de mensajes de petición duplicados.
  - Pérdida de mensajes de respuesta.
  - Historial.
- La comunicación cliente-servidor es síncrona (el cliente se bloquea hasta recibir una respuesta).
- *send-receive* requiere 4 llamadas al sistema para un intercambio.
- Existen protocolos (Chorus, Amoeba, Mach y V) que requieren sólo 3 llamadas al sistema: *doOperation*, *getRequest* y *sendReply*.

Protocolo petición-respuesta (II)



Identificadores de la invocación (I)

- El cliente genera una *idInvocacion* única donde:
  - Se añade al mensaje de solicitud.
  - El servidor la copia a su respuesta.
  - El cliente comprueba si es la esperada.
- Deben ser únicos y tienen 2 partes:
  - *IdInvocacion*: entero secuencial elegido por el emisor que ofrece unicidad entre invocaciones de un mismo emisor.
  - Un identificador del remitente que ofrece unicidad global. Por ejemplo, la dirección IP y el número de puerto donde se espera la respuesta.

Fallos en la entrega

- Los mensajes se pueden perder.
- Puede haber rotura: parte de la red aislada.
- Los procesos pueden fallar.
- Dificultad de elegir un N reintentos.
- Los datos si se reciben serán correctos.
- Se incorpora un temporizador al *receive* del cliente.

Temporizadores

- Cuando vence el temporizador del *receive* del cliente, el módulo de comunicaciones puede:
  - Retornar inmediatamente indicando el fallo ocurrido (poco común).
  - Reintentarlo repetidamente hasta obtener una respuesta o cuando exista probabilidad de que el servidor ha fallado.

Solicitudes duplicadas

- El servidor puede recibir solicitudes duplicadas, si los reintentos llegan antes de tiempo (servidor lento o sobrecargado).
- Para evitar estas ejecuciones repetidas:
  - Reconocer los duplicados del mismo cliente (igual *idInvocacion*).
  - Filtrarlos (descartarlos)

Respuestas perdidas

- Provocan que el servidor repita una operación.
- No es un problema cuando las operaciones del servidor son idempotentes:
  - Se pueden de realizar de forma repetida.
  - Los resultados son los mismos que si se ejecutasen una sola vez.

Historial

- Objetivo: retransmitir una respuesta sin volver a ejecutar la operación.
- El historial es el registro de las respuestas enviadas:
  - Mensaje con *idInvocacion* y su destinatario.
  - Gran consumo de memoria: se podrían descartar las respuestas tras algún tiempo.

Protocolos de intercambio RPC (I)

- 3 protocolos que se suelen utilizar:
  - R (*request* o petición).
  - RR (*request-reply* o petición-respuesta).
  - RRA (*request-reply-acknowledge reply* o petición-respuesta-confirmación de la respuesta).

Nombre	Mensajes enviados por Cliente	Mensajes enviados por Servidor	Mensajes enviados por Cliente
R	Petición		
RR	Petición	Respuesta	
RRA	Petición	Respuesta	Confirmación respuesta

Protocolos de intercambio RPC (II)

Protocolo R

- Sólo es útil cuando no hay valor de retorno del procedimiento y el cliente no necesita información.
- El cliente continúa tras enviar la solicitud.

Protocolo RR

- Común en entornos cliente-servidor.
- La respuesta asiente la solicitud.
- Una solicitud posterior del mismo cliente asiente la respuesta.

Protocolos de intercambio RPC (III)

Protocolo RRA

- La respuesta asiente la solicitud.
- El asentimiento de la respuesta lleva la *idInvocacion* de la respuesta a la que se refiere, asiente dicha respuesta y la de *Idvocacion* anterior, y permite vaciar entradas del historial.
- El envío del asentimiento de respuesta no bloquea al cliente pero consume recursos de procesador y red.

Ejemplo de protocolo de comunicación: HTTP

- La longitud de UDP podría no ser adecuada.
- Gracias a TCP se asegura que los datos sean entregados de forma fiable.
- El protocolo de petición-respuesta más conocido que utiliza TCP es HTTP:
  - Métodos: GET, PUT, POST, HEAD, DELETE, OPTIONS.
  - Permite: autenticación y negociación del contenido.
  - Establece conexiones persistentes, y abiertas durante el intercambio de mensajes.

Ejemplo en Python

socketHTTP.py

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ip = socket.gethostbyname("www.marca.com")
sock.connect((ip, 80))
sock.sendall( b"GET␣/␣HTTP/1.1\nHost:␣www.marca.com\n\n" )
bufsize = 4000
output = "␣"
buf = sock.recv ( bufsize )
while (buf):
        output += buf #buf.encode("utf-8")
        print(output)
        print("--------------------------")
        buf = sock.recv ( bufsize )
```

Sección 5 | Comunicación multicast

Multidifusión

- Multidifusión envía un único mensaje a todos los miembros de un grupo de forma transparente.
- No hay garantía de la entrega del mensaje ni del orden de los mensajes.
- Proporciona la infraestructura para desarrollar sistemas distribuidos con las características:
  - Tolerancia a fallos.
  - Búsqueda de los servidores de descubrimiento de redes espontáneas.
  - Mejores prestaciones basadas en datos replicados.
  - Propagación de notificaciones de eventos.

Bibliografía

Coulouris, G.; Dollimore, J.; Kindberg, T.
Distributed Systems: Concepts and Design (5ª ed.)
Addison-Wesley, 2012.
(Trad. al castellano: Sistemas distribuidos: conceptos y diseño, 3ª ed., Pearson 2001)