# Tema 3.- Middleware. ZeroMQ

TSR 2021. Juansa Sendra, grupo B

# Introducción

- Desarrollamos componentes de un sistema distribuido $\rightarrow$ deben cooperar entre sí
  - Ej. servicio planificación de rutas: depende de servicio GIS (información distancias)
  - Ej. sistema de autorización: necesita servicio de reconocimiento biométrico
- Complicado
  - Pueden desarrollarse por programadores diferentes, con entornos de programación distintos
  - Muchos detalles a considerar, especialmente para solicitar servicios
    - ¿Cómo localizar al servidor correcto? ¿Cuál es el API de un servicio? ¿Cómo construir/interpretar peticiones de servicio? ¿Cómo asociar a cliente la respuesta correcta? ...
  - Depuración compleja
  - ¿Seguridad?
  - ¿Gestión de fallos?

# Introducción.- Soluciones para reducir la complejidad

- Utilización de estándards
  - Facilitan la interoperabilidad
  - Introducen formas racionales de hacer las cosas
  - Proporcionan funcionalidad de alto nivel
- Reutilización de soluciones o componentes previos
  - Menos código que escribir
  - Mas garantías
- Middleware
  - Nivel de software y servicios entre las aplicaciones y el S.O.
  - Introduce transparencia de ... (ubicación, replicación, fallos, ...)

# Middleware

- Perspectiva del programador
  - Implantación sencilla (conceptes claros y bien definidos)
  - Resultado fiable (proporciona metodologia de desarrollo estandarizada y bien definida)
  - Simplifica mantenimento (revisiones APIs)
- Perspectiva del administrador
  - Simplifica instalación, configuración, actualización
  - Facilita interoperabilidad (con productos de terceras partes)

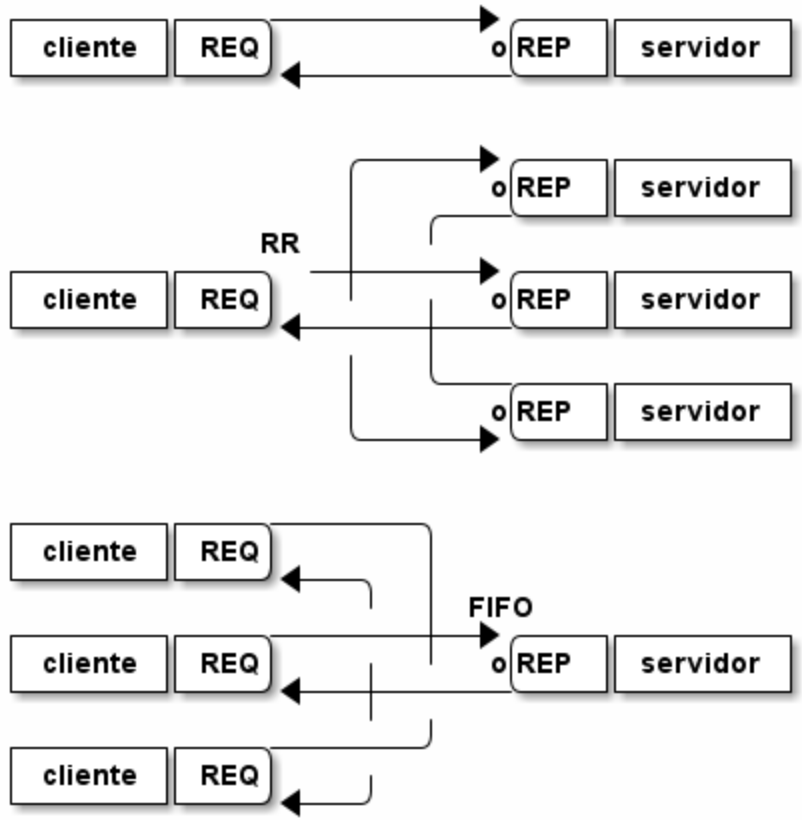
# Middleware orientado a comunicaciones (Sistemas de mensajería)

- Comunicación flexible
  - Transmisión atómica (todo o nada)
  - Tamaño arbitrario, mensajes estructurados
  - Gestión de colas, con ciertas garantías de orden
- No impone visión de estado compartido
  - El estado compartido escala mal y puede provocar problemas de concurrencia
  - Acoplamiento débil
- Encaja con el modelo 'dirigido por eventos'
  - Implícitamente asincrónico (desacopbla emisor/receptor)
- Clasificación de los sistemas de mensajeria
  - Persistentes.- buffers para mensajes. No requieren receptor activo
  - No persistentes.- el receptor ha de estar activo para transmitir el mensaje
    - Con gestor (ej. AMQP, JMS), o sin (ej. 0MQ) ↓ garantías ↑ escalable.

- Simple
  - URLs para identificar a los 'endpoints'
  - API similar a los sockets BSD (familiar): bind/connect, send/receive, ..
    - Muchos tipus de sockets, para implantar distintos patrones de comunicación
  - Únicamente es una biblioteca (no es necesario arrancar ningún servidor, etc.)
    - instalar: `npm install zeromq@5` utilizar: `const zmq = require('zeromq')`
  - Modelo Entrada/salida asincrónica (dirigida por eventos)
- Àmplia disponibilidad (para la mayoría de SO, lenguajes, entornos de programación)
- Soporta los principales patrones de interacción $\rightarrow$ curva de aprendizaje rápida
- Eficiente (compromiso fiabilidad/eficiencia) $\rightarrow$ colas en memoria (en emisor y receptor)
- Reutilizable. Mismo código para comunicar (cambiando solo las URL)
  - Hilos en un proceso, procesos en una máquina (IPC), ordinadores en red IP

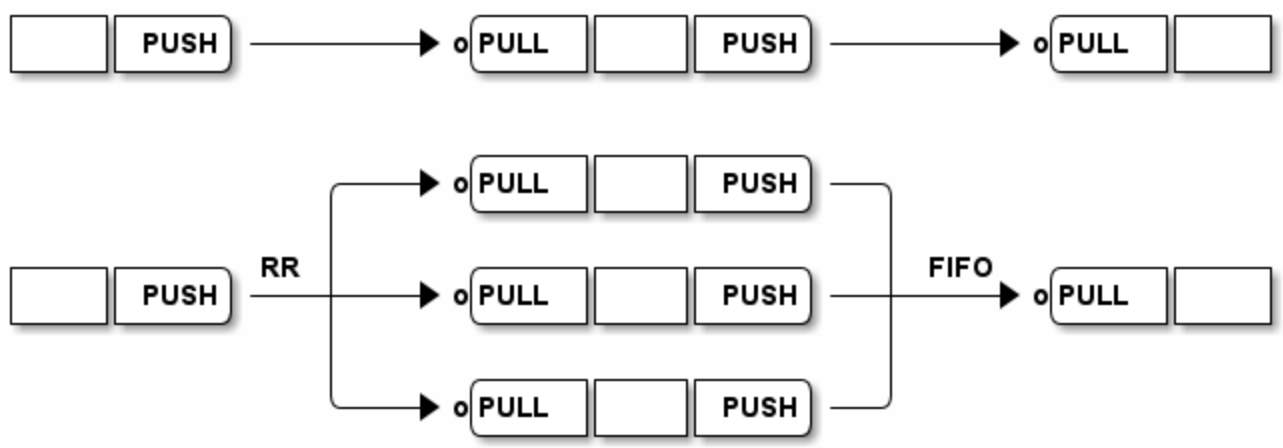
# 0MQ.- Tipos de Sockets y patrones de conexión

- El tipo de socket a utilizar depende de los patrones de conexión
  - cada patrón tiene necesidades diferents (utiliza sockets diferentes)
- Client/Server (sincrónico): req/rep



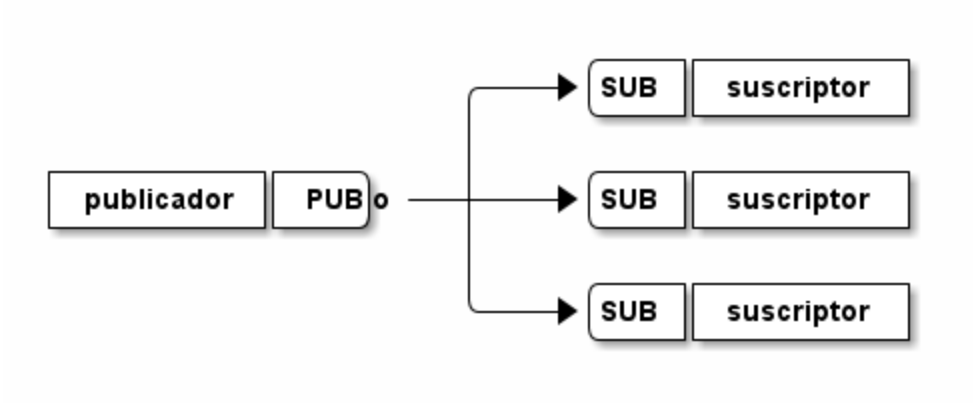
# 0MQ.- Tipos de Sockets y patrones de conexión

- Pipeline (unidireccional): push/pull



# 0MQ.- Tipos de Sockets y patrones de conexión

- Publicación/subscripción (multienvío de mensajes, porque los receptores pueden decidir a qué mensajes se suscriben): pub/sub



# 0MQ.- Tipos de Sockets y patrones de conexión

- Otros patrones (ej. broker/workers): router/dealer


- 0MQ proporciona otros tipos (pair, xsub, xpub), pero NO los estudiamos

# 0MQ.- Mensajes

- Contenido de los mensajes transparente para 0MQ
  - Soporta serialización (marshalling) y reconstrucción (unmarshalling) de cadenas (tiras de caracteres)
  - El programador decide cómo estructurar el contenido del mensaje
    - Cadena (lo que no sean cadenas se convierten primero a cadena)
      - Las cadenas se convierten a buffers utilizando UTF8
      - Luego las volvemos a convertir en cadenas (toString)
    - Podemos utilizar estandards como JSON o XML
- Los mensajes es entregan de forma atómica (entrega totes les parts o no entrega res)
- Envio y recepción asincrónicos (no bloqueantes)
  - Internamente 0MQ gestiona el flujo de mensajes entre colas de los procesos
- Gestión de conexió/reconexió entre agentes automática

# 0MQ.- Mensajes multisegmento

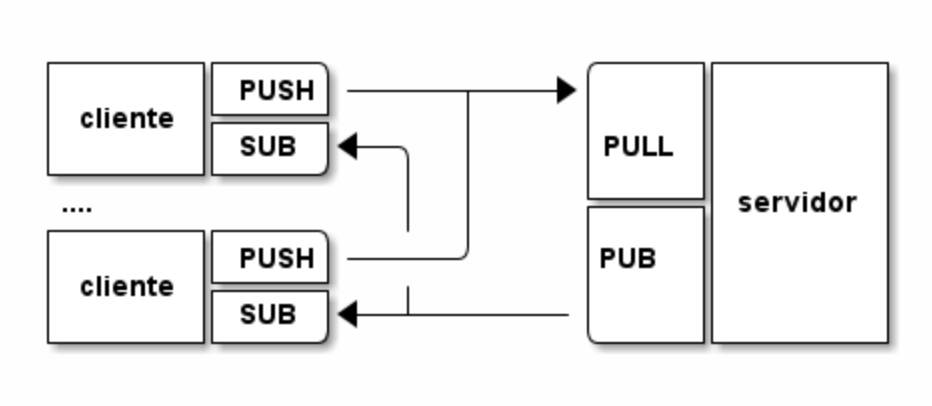
- Los mensajes pueden ser multisegmento
  - `socket.send("text") // 4text (1 segment)`
  - `socket.send(["Hola", "", "Ana"]) // 4Hola03Ana (3 segments)`
    - En la recepción podemos extraer los segmentos automáticamente (los argumentos del callback contienen los segmentos del mensaje)

      `sock.on('message', function(s1,s2,s3) {..})`
    - O los recogemos en un vector

      `sock.on('message', function(...msg) {for (let seg of msg) {..}})`
  - Podemos utilizar cada segmento para una pieza de información diferente
    - Ej. [nomAPI, versioAPI, operació, arg, ..]
  - Denominamos 'delimitador' al primer segmento vacío ("")

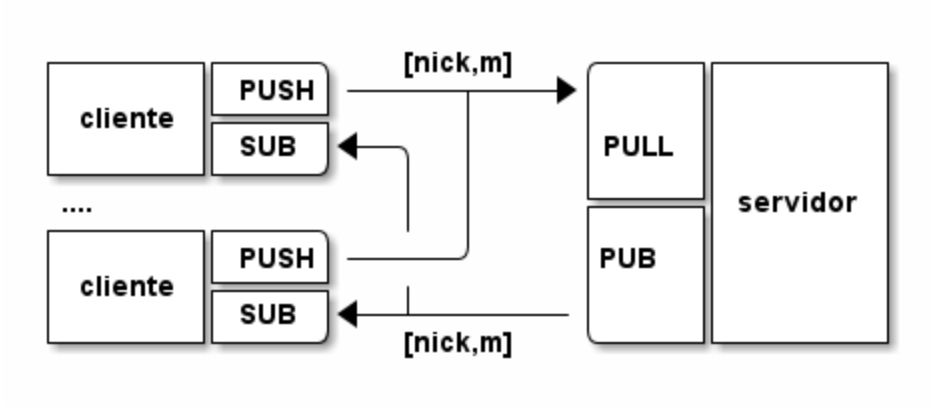
# 0MQ.- Pasos para desarrollar una solución (ej. chat)

- Identificamos los patrones de interacción (de donde derivamos qué tipos de sockets necesitamos, y dónde ubicarlos). Un xat combina:
  - Pipeline (clientes push, servidor pull)
    - Cada cliente envia msg al servidor cuando el usuario introduce una frase
  - Difusión (clientes sub, servidor pub)
    - El servidor difunde a todos los clientes cada nueva frase



# 0MQ.- Pasos para desarrollar una solución (ej. chat)

- Define el formato de los mensajes a intercambiar
  - Cliente a servidor: [remitente (autor de la frase = nick), text]
    - texto 'HI' para darse de alta, texto 'BYE' para darse de baja
  - servidor a clientes: [autor de la frase(= nick), text]
    - nick **server** si la frase la genera el servidor (ej. para avisar alta o baja de un cliente)



# 0MQ.- Pasos para desarrollar una solución (ej. chat)

- Define las respuestas de cada agente ante los eventos generados por los diferentes sockets
  - Cliente
    - `process.stdin.on('data',(str)=>{push.send([nick,str])}) // frase escrita por teclado`
    - `process.stdin.on('end',()=>{push.send([nick,'BYE'])}) // al cerrar stdin`
    - `sub.on('message', (nick,m) => {..}) // al recibir mensaje del servidor`
  - Servidor
    - `pull.on('message', (nick,m) => {..}) // al recibir mensaje del client`

# 0MQ.- Establecimiento connexión (bind/connect sobre transporte TCP)

- Gestión de conexión/reconexión entre agentes automática
- Un proceso (el que debería llegar primero y marcharse el último) realiza un **bind**
  - `sock.bind('tcp://*:5555', function(err) {..})` [5555]
- Otros procesos (uno o mas) realizan un **connect** (usando la IP y socket del que hace **bind**)
  - `sock.connect('tcp://10.0.0.1:5555', function(err) {..})`
- Cuando un agente acaba ejecuta **close** implicito. Podemos invocarlo de forma explícita
  - `sock.close()`
- Además de comunicación 1:1
  - N:1 $\longrightarrow$ N clientes (cada uno 1 connect), 1 servidor (bind)
  - 1:N $\longrightarrow$ 1 cliente (N connects, uno a cada servidor), N servidores (cada uno 1 bind)

# 0MQ.- Establecimiento connexión (bind/connect)

- No es obligatorio un orden entre bind y connect. El primero que llega espera
- Podemos establecer conexiones 1:1, 1:N, N:1
  - `sock.bind('tcp://*:5555', function(err) {...})` [5555]
  - `sock.bind('tcp://*:5555', function(err) {...})` [5556]
  - `sock1.connect('tcp://10.0.0.1:5555', function(err) {...})`
    `sock1.connect('tcp://10.0.0.1:5556', function(err) {...})`
  - `sock2.connect('tcp://10.0.0.1:5555', function(err) {...})`
  - `sock3.connect('tcp://10.0.0.1:5556', function(err) {...})`
- figura

# 0MQ en node

```
const zmq = require('zeromq') // importa biblioteca
let zsock = zmq.socket('tipusSocket')// creació socket (existeixen diversos tipus)
zsock.bind("tcp://*:5555") // bind en el port 5555
zsock.connect("tcp://10.0.0.1:5555") // o connect (host 10.0.0.1, port 5555)
zsock.send([..,..]) // enviament
zsock.on("message", callback) // recepció
zsock.on("close", callback) // resposta al tancament de la connexió
```

# 0MQ.- código ejemplo servidor chat

```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let pull= zmq.socket('pull')
pub.bind ('tcp://*:9998')
pull.bind('tcp://*:9999')

pull.on('message', (id,txt) => {
   switch (txt.toString()) {
     case 'HI' : pub.send(['server',id+' connected']);    break
     case 'BYE': pub.send(['server',id+' disconnected']); break
     default   : pub.send([id,txt])
   }
})
```

# 0MQ.- código ejemplo cliente chat

```
const zmq = require('zeromq')
const nick='Ana'
let sub = zmq.socket('sub')
let psh = zmq.socket('push')
sub.connect('tcp://127.0.0.1:9998')
psh.connect('tcp://127.0.0.1:9999')
sub.subscribe('') // subscrit a tots el missatges

sub.on('message', (nick,m) => {console.log('['+nick+']'+m)})

process.stdin.resume()
process.stdin.setEncoding('utf8')
process.stdin.on('data' ,(str)=> {psh.send([nick, str.slice(0,-1)])})
process.stdin.on('end',()=> {psh.send([nick, 'BYE']); sub.close(); psh.close()})
process.on('SIGINT',()=> {process.stdin.end()})

psh.send([nick,'HI'])
```

# 0MQ.- Colas y opciones

- Los sockets pueden tener colas de mensajes asociadas
  - De entrada (recepción), para mantener los mensajes que llegan
    - generan eventos 'message' cuando llega un mensaje
  - De salida (envío), para mantener los mensajes a enviar a otros
- Podemos asociar algunas opciones a los sockets. Estudiamso únicamente 2:
  - identity.- Fija identificación del agente que se conecta a un router
  - subscribe.- Fija el filtro de prefijos aplicado al socket 'pub'
- Router es el único que puede enviar mensaje a un destino concreto, y sabe quién le ha enviado un mensaje

# 0MQ.- Colas y opciones

| Socket | Cola entrada | Cola salida | opciones | send | receive |
|---|---|---|---|---|---|
| req | 1 | 1 | identity | + delimitador | - delimitador |
| rep | 1 | 1 | identity | + delimitador | - delimitador |
| push | | 1 | identity | | |
| pull | 1 | | identity | | |
| pub | | 1 | | | |
| sub | 1 | | subscribe | | |
| router | 1 per conexió | 1 per conexió | | - destinació | + emisor |
| dealer | 1 | 1 | | | |

# Otros middleware

- Gestión de eventos (ej. JINI)
  - Se incluye con frecuencia en sistemes de mensajería
    - Patrón publicador/subscriptor
- Seguridad
  - Autenticación (ej. OpenID)
    - Una tercera part garanteix la identitat d'un agent
  - Autorización (ej. OAuth)
    - Una tercera parte autoriza una petició
  - Integración con otros protocolos (ej. SSL/ TLS y HTTPS)
- Soporte transaccional
  - Coordinación de modificaciones del estado distribuido (modificaciones atómicas)
    - Soporta las situaciones de fallo

# Conclusiones

- Para gestionar la complejidad de los sistemes distribuidos
  - Gestión adecuada del código y de los servicios
  - Utilización de estandards
  - Utilización middleware
- Principales objetivos middleware
  - Tareas de comunicaciones
  - Petición de servicios
- Principales variantes middleware orientado a comunicaciones
  - Gestión mensajes persistente SI/NO
  - Basados en gestor/sin gestor
- Otros middleware
  - Seguridad
  - Transacciones