TSR - Documentación de referencia sobre **0MQ**

Curso 2019/20

Contents

1. 0MQ

- La asignatura CSD introdujo MOM (Message Oriented Middleware)
  - Comunicación indirecta → asincronía, persistencia, acoplamiento débil
  - Se estudió JMS como ejemplo
- 0MQ es otro ejemplo de MOM, pero muy distinto a JMS:

  - Sin broker para enrutar mensajes, con API similar a sockets
    * No requiere arrancar ningún servidor específico
    * Proporciona sockets para envío/recepción
    * Utiliza URLs para nombrar los endpoints
    * Persistencia débil (colas en RAM)
  - Modelo E/S asincrónica (dirigido por eventos)
  - Amplia disponibilidad (para muchos SO y lenguajes)
    * Biblioteca gratuita (código abierto), enlaza con la aplicación

1.1. Características básicas

- Eficiente (compromiso fiabilidad/eficiencia)
  - Persistencia débil (colas en RAM)
  - Los sockets tienen colas de mensajes asociadas

cola mensajes	descripción	eventos
de entrada (recepción)	mantiene mensajes que llegan	genera evento "message"
de salida (envío)	mantiene mensajes a enviar	

- Define distintos patrones de intercambio de mensajes
  - facilita el desarrollo
- Útil a varios niveles → mismo código para comunicar hilos en un proceso, procesos en una máquina, máquinas en red IP
  - Sólo cambia la configuración del transporte en la URL
  - Nos centramos en comunicación entre máquinas sobre TCP

1.2. Mensajes

- Gestión de buffers transparente
  - Gestiona el flujo de mensajes entre las colas de los procesos y nivel de transporte
  - Contenido del mensaje transparente
- Los mensajes se entregan de forma atómica (todo o nada)
  - Pueden ser multi-segmento (segmento = counted string)
    * 1 segmento: `send('hola')` → `4 h o l a`
    * 3 seg: `send(['hola','','Ana'])` → `4 h o l a 0 3 A n a`

envío mensaje	recepción
`send(['uno','dos'])`	`sock.on("message", (a,b)=>{..})` `a` vale `'uno'`, `b` vale `'dos'`
`send(msg)`	`sock.on("message", (...m)=>{..})` segmentos de `msg` en el vector `m`

1.3. Conexiones

- Gestión de conexión/reconexión entre agentes automática
  - Un agente ejecuta bind: el resto ejecuta connect

    - En cualquier orden
    - Todos los agentes coinciden en algún endpoint
  - Si se ejecuta bind sobre un port que ya está en uso, aparece un error de ejecución

- Conexión/Reconexión en el transporte TCP

  - bind.- La dir IP pertenece a una de las interfaces del socket
    - `s.bind('tcp://*:9999')`
  - connect.- debe conocer la dir IP del socket que realice `bind`
    - `s.connect('tcp://127.0.0.1:9999')`

- Cuando un agente termina ejecuta close de forma implícita

- No sólo comunicación 1:1

  - n:1 → ej. n clientes (cada uno connect), 1 servidor (bind)
  - 1:n → ej. 1 cliente (n connect, uno a cada servidor), n servidores (cada uno bind)

1.4. 0MQ en node

- Instalación biblioteca: `npm install zeromq@4`
- Sintaxis

```
const zmq = require('zeromq')         // importa biblioteca
let zsock = zmq.socket('tipoSocket') // creación socket (existen varios tipos)
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([..,..])                   // envío
zsock.on("message", callback)         // recepción
zsock.on("close", callback)           // respuesta al cierre de la conexión
```

1.5. Sockets 0MQ

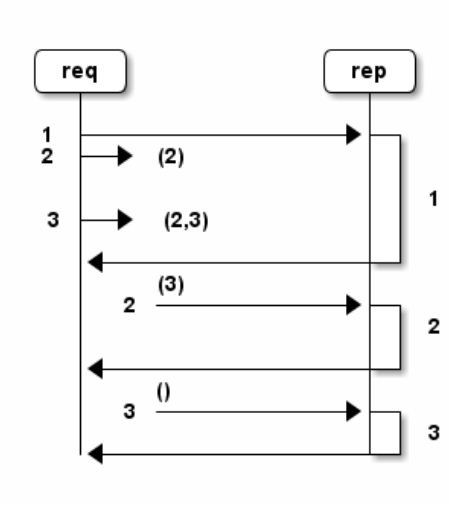
- Existen distintos tipos de sockets, para implementar patrones de diseño (patrones de comunicación) frecuentes en Sistemas Distribuidos

  - Cada patrón tiene necesidades distintas → utiliza sockets diferentes
  - Muchos se resuelven con un par de sockets específicos
  - En otros casos hay que combinar varios tipos de sockets simples

- Cuando conozcamos los tipos de sockets, 3 pasos para diseñar aplicación distribuida:

  1. Decide qué combinaciones de sockets necesitas, y en qué agentes se ubican
  2. Define el formato de los mensajes a intercambiar
  3. Define las respuestas de cada agente ante los eventos generados por los distintos sockets

2. Patrón Cliente/Servidor: req/rep



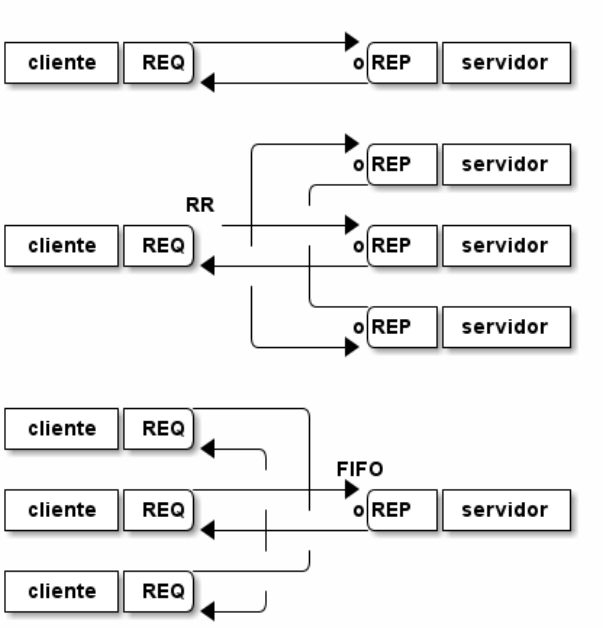
- El servidor usa socket tipo `rep`
  - Ejecuta `bind`
  - Recibe `s.on('message', callback)`
- El cliente usa socket tipo `req`
  - Ejecuta connect
  - Envía con `s.send(msg)`
- Es un **patrón de comunicación sincrónico**
  - Si un cliente envía n peticiones, la segunda, tercera,.. queda en cola local hasta recibir respuesta de la primera
  - Pares pet/resp totalmente ordenados

2.1. Conexiones

En todos los casos

- Solicitud/Respuesta
- En las figuras el símbolo `o` representa `bind`

Son posibles diferentes estrategias de conexión (figura).



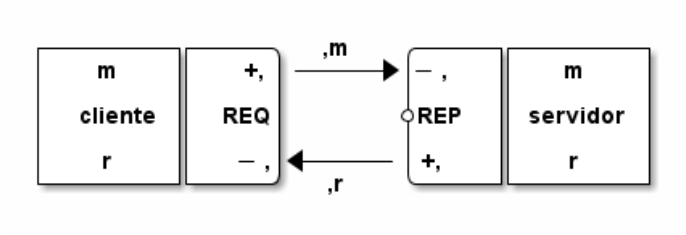
- Conexión `1:1`
- Conexión `1:n`
  - Round-Robin (RR)
  - No paralelización
- Conexión `n:1`
  - Típica servidor
  - Cola FIFO (encolado equitativo)
    - No inanición clientes

2.2. Formato mensajes

1. La aplicación cliente envía un mensaje `m`
2. `req` añade un primer segmento vacío (delimitador) al mensaje
   - Delimitador → segmento vacío que separa envoltorio y cuerpo del mensaje

  - Los segmentos hasta el primer delimitador se denominan envoltorio: permiten indicar metadatos asociados al mensaje (ej.- a quién hay que devolverle la respuesta)
  - Los segmentos siguientes al primer delimitador se consideran el cuerpo del mensaje (datos)
- En la figura representamos el delimitador como ,

3. `rep` guarda el envoltorio y pasa el cuerpo a la aplicación
4. Cuando `rep` envía la respuesta, añade de nuevo el envoltorio
5. Cuando `req` recibe la respuesta, descarta el delimitador y pasa el cuerpo a la aplicación



2.3. Ejemplo Cliente/Servidor 1:1

cliente.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9999')
s.send('Alex')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

Ejecución

```
> node cliente.js
Recibido: Hola, Alex
```

servidor.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9999')
s.on('message', (nom) => {
  console.log('Nombre: '+nom)
  s.send('Hola, '+nom)
})
```

Ejecución

```
> node servidor.js
Nombre: Alex
```

2.4. Ejemplo Cliente/Servidor 1:n

cliente.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.connect('tcp://127.0.0.1:9999')
s.send('Alex1')
s.send('Alex2')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
})
```

servidor1.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9998')
s.on('message', (nom) => {
  console.log('Serv1, '+nom)
  s.send('Hola, '+nom)
})
```

servidor2.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9999')
s.on('message', (nom) => {
  console.log('Serv2, '+nom)
  s.send('Hola, '+nom)
})
```

2.5. Ejemplo Cliente/Servidor n:1

client1.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.send('uno')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

client2.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.send('dos')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

server.js

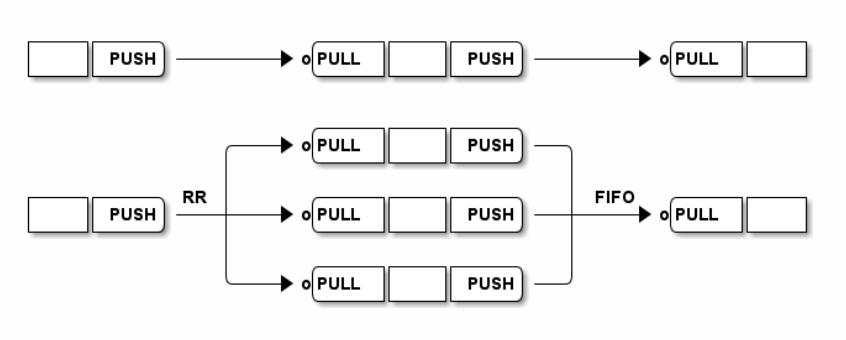
```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9998')
s.on('message', (n) => {
  console.log('Serv1, '+n)
  switch (n) {
      case 'uno': s.send('one'); break
      case 'dos': s.send('two'); break
      default: s.send('mmmmm.. no se')
  }
})
```

3. Patrón Pipeline: push/pull

- Etapas de procesamiento conectadas entre sí
  - El emisor no espera respuesta
  - Envíos concurrentes
- Conexiones
  - `1:1`, `1:n` (RR), `n:1` (encolado equitativo)

  - `1:n:1` (map-reduce)

- `push` y `pull` no alteran el formato de los mensajes



3.1. Ejemplo Pipeline (1:1)

emisor.js

```
const zmq = require('zeromq')
let s = zmq.socket('push')
s.connect('tcp://127.0.0.1:9999')
s.send(['ejemplo','multisegmento'])
```

receptor.js

```
const zmq = require('zeromq')
let s = zmq.socket('pull')
s.bind('tcp://*:9999')
s.on('message', (tipo,txt) => {
    console.log('tipo '+tipo +' texto '+txt)
})
```

3.2. Ejemplo Pipeline (1:n:1)

fan.js

```
const zmq = require('zeromq')
let s = zmq.socket('push')
s.connect('tcp://127.0.0.1:9996')
s.connect('tcp://127.0.0.1:9997')
s.connect('tcp://127.0.0.1:9998')
for (let i=0; i<8; i++)
    s.send(''+i)
```

sink.js

```
const zmq = require('zeromq')
let s = zmq.socket('pull')
s.bind('tcp://*:9999')
s.on('message', (w,n) => {
    console.log('worker '+w +' resp '+n)
})
```

worker1.js

```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9996')
sin.on('message', n => {sout.send(['1',n])})
```

worker2.js

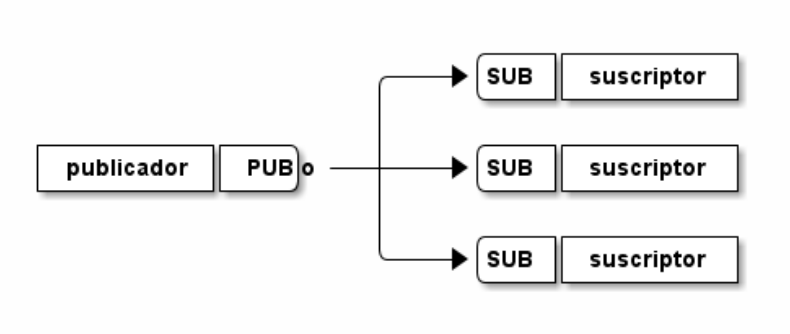
```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9997')
sin.on('message', n => {sout.send(['2',n])})
```

worker3.js

```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9998')
sin.on('message', n => {sout.send(['3',n])})
```

4. Patrón Difusión: pub/sub

- Un publicador (`pub`) difunde mensajes a n suscriptores (`sub`)
  - El suscriptor debe suscribirse al tipo de mensajes que le interesen `socket.subscribe(tipoMsg)`
    - Puede suscribirse a varios tipos de mensajes
  - `s.subscribe('xx')` → el socket `s` únicamente recibe los mensajes que tengan `xx` como prefijo del primer segmento
    - El prefijo `''` (tira vacía) concuerda con todos los mensajes



- El suscriptor empieza a recibir los mensajes de `tipoMsg` a partir del momento en que se suscribe (si el publicador ya había enviado otros previamente, no los recibe)
- `pub` y `sub` no modifican el formato de los mensajes

4.1. Ejemplo Patrón pub/sub

publicador.js

```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let msg = ['uno', 'dos', 'tres']
pub.bind('tcp://*:9999')
function emite() {
    let m=msg[0]
    pub.send(m)
    msg.shift(); msg.push(m) //rotatorio
}
setInterval(emite,1000) // every second
```

suscriptor1.js

```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('uno')
sub.on('message', (m) =>
    {console.log('1',m+'')})
```

suscriptor2.js

```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('dos')
sub.on('message', (m) =>
    {console.log('2',m+'')})
```

suscriptor3.js

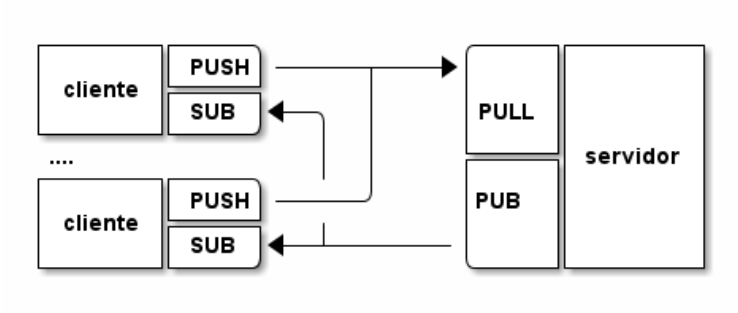
```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('tres')
sub.on('message', (m) =>
    {console.log('3',m+'')})
```

5. Ejemplo de diseño de una aplicación completa: Chat

- 1 servidor, n clientes
  - Los clientes se registran en el servidor (operaciones de alta y baja)
  - Cada cliente puede enviar mensajes al servidor
  - El servidor difunde cada mensaje a todos los clientes registrados
- Pasos para diseñar una aplicación distribuida:
  1. Decide qué combinaciones de sockets necesitas, y en qué agentes se ubican
  2. Define el formato de los mensajes a intercambiar
  3. Define las respuestas de cada agente ante los eventos generados por los distintos sockets

5.1. Paso 1: patrones de interacción

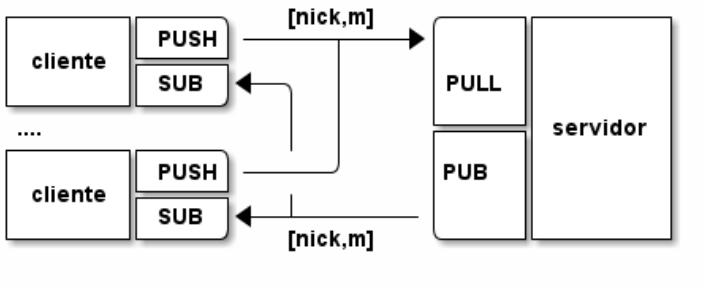
- Un chat combina pipeline y difusión
  - pipeline (cliente `push`, servidor `pull`)
    - Cada cliente envía msg al servidor cuando el usuario introduce una nueva frase
  - difusión (cliente `sub`, servidor `pub`)
    - El servidor difunde a todos los clientes cada nueva frase



NOTA.- no es la única decisión de diseño posible. Por ejemplo, podemos resolver la difusión en base a mantener en el servidor la lista de clientes, y remitir mensajes a cada uno

5.2. Paso 2: formato de los mensajes

- Cliente a servidor.- Indica remitente (nick único) y texto
  - El nick indica el autor de la frase
- Mensaje difundido por el servidor a clientes
  - Debe indicar el autor de la frase y el texto
  - El autor de la frase puede ser el servidor (nick `server`)
    - Ej.- para avisar del alta o baja de un cliente, etc.
- El cliente presenta el mensaje (interfaz texto, interfaz gráfico, color según el nick, …)



5.3. Paso 3: eventos

- Cliente:
  - Inicialmente el cliente ejecuta `push.send([nick,'HI'])`
  - El socket `sub` escucha todos los mensajes publicados por el servidor: `sub.subscribe('')`
  - Llega mensaje del servidor al socket `sub`:
    - `sub.on('message', (nick,m)=> {...})`
  - Remite al servidor toda frase escrita por teclado:
    - `process.stdin.on('data',(str)=>{push.send([nick,str])})`
  - Da de baja al cliente cuando finaliza la aplicación:
    - `process.stdin.on('end',()=>{push.send([nick,'BYE'])})`
- Servidor:
  - Llega msg: `pull.on('message',(id,m)->{..})`
    - si `m` es `'HI'` → alta de `id` (difunde a todos el aviso del alta)
    - si `m` es `'BYE'` → baja de `id` (difunde a todos el aviso de la baja)
    - Para otro `m`, difunde `[id,m]`

5.4. Código chat

client.js

```
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('')
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'])
```

server.js

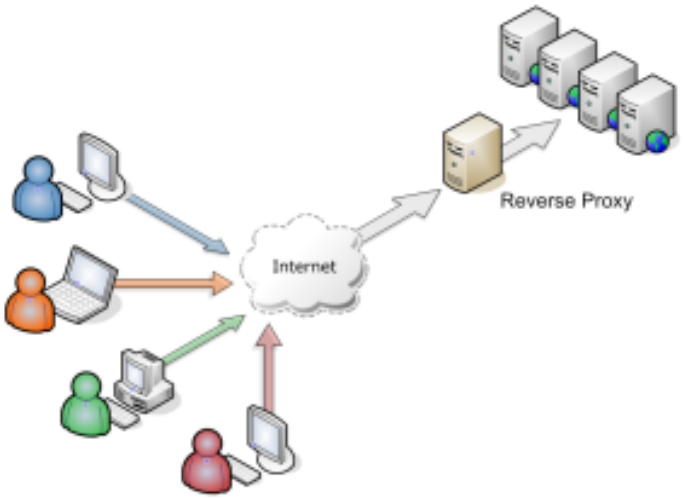
```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let pull= zmq.socket('pull')

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])
    }
})
```

El chat desarrollado es muy rudimentario (limitado), pero puede extenderse fácilmente en distintas direcciones:

- Permitir distintos grupos de usuarios (varias conversaciones). En ese caso cada cliente se suscribe a aquellos grupos en los que está interesado
- Permitir mensajes privados (no se difunden, sino que se remiten de forma directa a destino)
- Los clientes deberían poder seleccionar/modificar su nick, y el servidor verificar que cada nick es único
- El código actual no comprueba posibles fallos (ej. no verifica la entrada del usuario, de forma que se puede escribir un mensaje que coincida con los de alta/baja, etc.)
- El servidor no mantiene estado: un cliente que no está conectado se pierde mensajes, y dichos mensajes no se reenvían cuando conecta el cliente

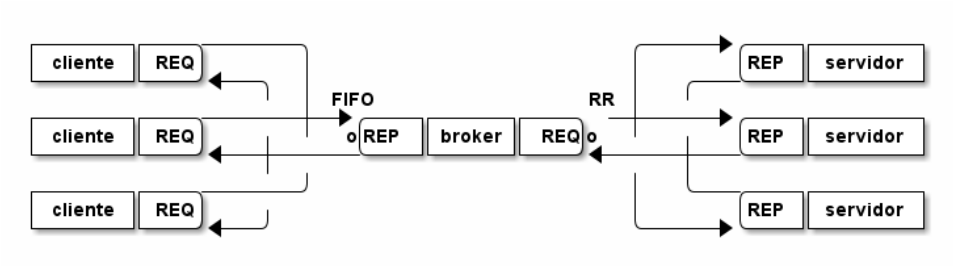
6. Patrón broker (proxy inverso)

- El componente broker (intermediario) se ocupa de la coordinación/comunicación entre componentes
  - Los servidores comunican al broker sus características
  - Los clientes solicitan los servicios al broker
  - El broker redirecciona cada solicitud al servidor adecuado (ej. equilibrando la carga)



6.1. Broker `req/rep`

- Utilizar el patrón `req/rep` para solicitud/respuesta
- El cliente pide al broker, y el broker a un worker



- **Incorrecto**. El broker no atiende a `n` clientes a la vez
  - Con `req/rep` hasta que el broker no devuelve la respuesta al primer cliente no puede procesar la solicitud de otro cliente
  - Tenemos comportamiento sincrónico, y debe ser asincrónico

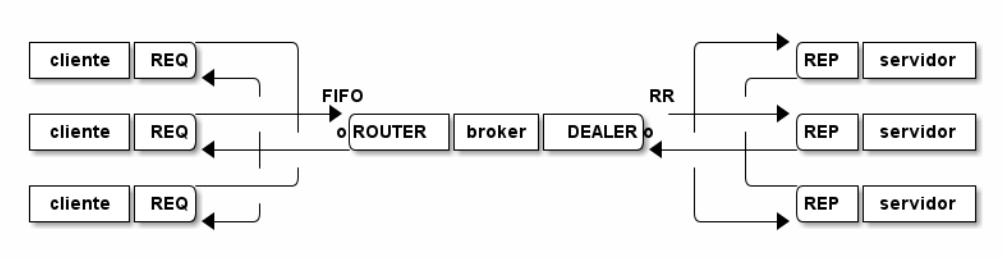
6.2. Broker `router/dealer`

- Introducimos los dos últimos tipos de sockets
- Socket **dealer**
  - Es asincrónico, y permite envío (RR) y recepción (FIFO)
  - No modifica el mensaje ni al enviar ni al recibir
- Socket **router**
  - Es asincrónico, y permite envío y recepción (FIFO)
    - Mantiene una cola por conexión.- cuando recibe, añade al mensaje la identidad de la conexión (id. emisor)

        - `s.identity='xx'` antes de establecer la conexión → al conectar, el identificador de la conexión es 'xx'

    * Al enviar, consume el primer segmento del mensaje, y lo utiliza para determinar la conexión a través de la cual envía el mensaje (puede enrutar)

- Utilizamos la configuración `router/dealer` en el broker



- **Correcto**

  - `router` y `dealer` son asincrónicos (atender la petición de un cliente en un worker no impide atender peticiones de otros en otros workers) → broker asincrónico
  - Peticiones servidas en orden de llegada (FIFO)
  - Reparto de tareas a los workers según turno rotatorio (RR)

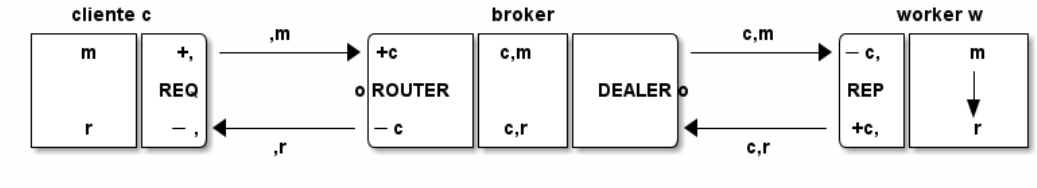
- Formato mensajes

  - Decisión de diseño.- la información de cliente viaja junto al mensaje

    * No requiere guardar estado de cada cliente

  - En la figura `m` es el mensaje, `r` la respuesta, `c` identidad del cliente, `w` identidad del worker, y `','` un delimitador

    * `req` añade delimitador al enviar, lo elimina al recibir
    * `router` añade identidad de la conexión al recibir, y la consume al enviar
    * `rep` guarda el envoltorio al recibir, lo reinserta al enviar



client.js

```
const zmq = require('zeromq')
let req = zmq.socket('req');
req.connect('tcp://localhost:9998')
req.on('message', (msg)=> {
    console.log('resp: '+msg)
    process.exit(0);
})
req.send('Hola')
```

worker.js

```
const zmq = require('zeromq')
let rep = zmq.socket('rep');
rep.connect('tcp://localhost:9999')
rep.on('message', (msg)=> {
    setTimeout(()=> {
        rep.send('resp')
    }, 1000)
})
```

broker.js

```
const zmq = require('zeromq')
let sc = zmq.socket('router') // frontend
let sw = zmq.socket('dealer') // backed
sc.bind('tcp://*:9998')
sw.bind('tcp://*:9999')
sc.on('message',(...m)=> {sw.send(m)})
sw.on('message',(...m)=> {sc.send(m)})
```

El broker se limita a:

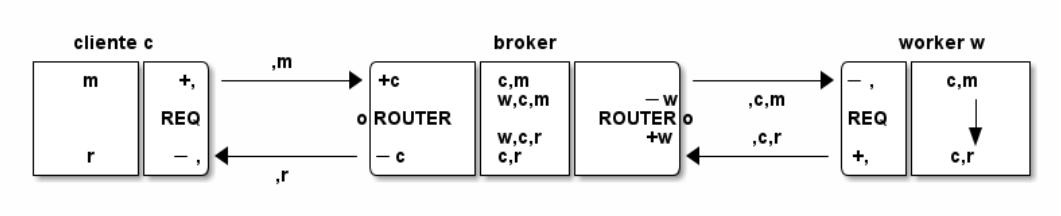
- Reenviar por el backend (hacia un trabajador, según turno RR) cada petición de cliente que le llega por el frontend
- Reenviar al cliente adecuado a través del frontend cada respuesta que le llegue desde un trabajador a través del backend. Para devolver la respuesta al cliente adecuado se utiliza la información de cliente que viaja en el mensaje de respuesta.

6.3. Broker `router/router`

El segundo intento (`router/dealer`) utiliza RR para repartir el trabajo. Esa solución sólo es válida cuando el coste de procesar cada trabajo es muy similar. En general es conveniente poder decidir el destino (enrutar) → ej. para conseguir equilibrado de carga.

Para el caso general se propone un `broker` con `router/router`

- Requiere mantener en el broker la lista de workers disponibles
  - Añadir un nuevo worker (alta), eliminarlo si se da de baja
    - Nuevos mensajes `worker->broker`: alta y baja
  - Cambiamos el socket del worker a req: en lugar de responder a peticiones, solicita trabajos
- Formato mensajes
  - Decisión de diseño.- la info de cliente viaja junto al mensaje
    - No requiere guardar estado de cada cliente
  - En la figura `m` es el mensaje, `r` la respuesta, `c` identidad del cliente, `w` identidad del worker, y `','` un delimitador
    - `req` añade delimitador al enviar, lo elimina al recibir
    - `router` añade identidad de la conexión al recibir, y la consume al enviar
    - `rep` guarda el envoltorio al recibir, lo reinserta al enviar



client.js

```
const zmq = require('zeromq')
let req = zmq.socket('req');
req.connect('tcp://localhost:9998')
req.on('message', (msg)=> {
    console.log('resp: '+msg)
    process.exit(0);
})
req.send('Hola')
```

worker.js

```
const zmq = require('zeromq')
let req = zmq.socket('req')
req.identity='Worker1'+process.pid
req.connect('tcp://localhost:9999')
req.on('message', (c,sep,msg)=> {
    setTimeout(()=> {
        req.send([c,'','resp'])
    }, 1000)
})
req.send(['','',''])
```

broker.js

```
const zmq = require('zeromq')
let cli=[], req=[], workers=[]
let sc = zmq.socket('router') // frontend
let sw = zmq.socket('router') // backend
sc.bind('tcp://*:9998')
sw.bind('tcp://*:9999')
sc.on('message',(c,sep,m)=> {
    if (workers.length==0) {
        cli.push(c); req.push(m)
    } else {
        sw.send([workers.shift(),'',c,'',m])
    }
})
sw.on('message',(w,sep,c,sep2,r)=> {
    if (c=='') {workers.push(w); return}
    if (cli.length>0) {
        sw.send([w,'',
            cli.shift(),'',req.shift()])
    } else {
        workers.push(w)
    }
    sc.send([c,'',r])
})
```

Todos los mensajes que el broker recibe desde frontend tienen la estructura `[c,'',m]`, donde `c` corresponde a la identidad del cliente que realiza la solicitud y `m` al mensaje de petición.

- Si al recibir la petición no hay trabajadores disponibles (`workers.length==0`), anotamos al final del vector `cli` el valor de `c` y al final del vector `req` el valor de `m` (representa un petición pendiente de procesar)
- Si al recibir la petición hay trabajadores disponibles, se selecciona uno de ellos (por simplicidad el primero) y se le envía la petición

Todos los mensajes que el broker recibe desde backend tienen la estructura `[w,'',c,'',r]`, donde `w` corresponde a la identidad del worker que envía la respuesta, `c` al cliente al que se responde, y `r` representa la propia respuesta

- El primer mensaje que envía un trabajador es `['','','']`, que al llegar al broker se recoge como `[w,'','','','']` (ya que el socket `req` añade como prefijo un delimitador y el router la identidad del emisor). Eso significa que la identidad del cliente es `''` (no estamos respondiendo a un cliente), y sirve para diferenciar el primer mensaje de un trabajador de la respuesta a una petición de cliente
- En la respuesta a una petición de cliente (la identidad de cliente no es `''`)

- Comprobamos si hay peticiones pendientes
  - Si hay peticiones pendientes (`cli.length>0`) extrae la primera petición y la pasa a este trabajador
  - Si no había peticiones pendientes, añadimos el trabajador a la lista de trabajadores disponibles.
- Enviamos la respuesta al cliente

Analiza el código proporcionado hasta comprender su funcionamiento.

7. Posibles mejoras sobre el broker `router/router`

Podemos plantear distintas modificaciones sobre el broker `router/router` para mejorar la escalabilidad y/o disponibilidad. En cada propuesta describimos aspectos a resolver y posibles estrategias, aunque no llegamos necesariamente a nivel de código. Un análisis detallado de las estrategias y técnicas planteadas facilita abordar otras posibles mejoras/ampliaciones del broker.

7.1. Patrón broker tolerante a fallos

Queremos implementar tolerancia a fallos de workers. Para ello debemos detectar el fallo de un worker, y reconfigurar el sistema para seguir funcionando sin ese trabajador.

Para detectar el fallo podemos plantear distintas alternativas:

1. sockets adicionales (tipo `router` en el broker, tipo `req` en cada worker), de forma que el broker envía de forma periódica mensajes denominados 'latido' (heartbeat) a los trabajadores, y éstos responden para indicar que siguen vivos: si la respuesta tarda demasiado, suponemos que el trabajador ha caído

2. idem. anterior, pero para ahorrar mensajes sólo enviamos latido a los workers de los que no se tiene noticia reciente (ej. no han devuelto recientemente respuestas a peticiones de trabajos)

3. Utilizar únicamente las peticiones/respuestas habituales: un worker se considera caído si tras enviarle una petición tarda demasiado en devolver la respuesta → al enviar una petición a un trabajador, el broker utiliza `setTimeout(reconfigura, answerInterval)`: si llega respuesta se cancela ese timeout, y si vence el timeout consideramos que el trabajador ha fallado y se ejecuta `reconfigura`

Para no complicar el código, asumimos la alternativa (3)

Queremos transparencia de fallos → cuando el broker detecta el fallo del trabajador `w`, reenvía la solicitud que estaba procesando `w` a otro trabajador.

- Debemos registrar en el broker todas las peticiones pendientes de respuesta, y anotar para cada una qué worker la está procesando
- Si detecta el fallo del worker `w`, el broker dispone de la información sobre la petición que estaba procesando `w`, y puede reenviarla a otro

ftbroker.js (broker que tolera fallos de workers)

```
const zmq = require('zeromq')
const ansInterval = 2000     // answer timeout. If exceeded, worker failed

let who=[], req=[]           // pending request (client,message)
let workers=[], failed={}    // available & failed workers
```

```
let tout={}                  // timeouts for attended requests

let sc = zmq.socket('router') // frontend
let sw = zmq.socket('router') // backend
sc.bind("tcp://*:9998", (err)=>{
   console.log(err?"sc binding error":"accepting client requests")
})
sw.bind("tcp://*:9999", (err)=>{
   console.log(err?"sw binding error":"accepting worker requests")
})

function dispatch(c,m) {
    if (workers.length)             // if available workers,
      sendToW(workers.shift(),c,m)  //    send request to first worker
    else {                          // no available workers
      who.push(c); req.push(m)      //    set as pending
  }
}
function resend(w,c,m) {
    return function() {  // ansInterval finished and not response
        failed[w]=true   // Worker w has failed
        dispatch(c,m)
    }
}
function sendToW(w,c,m) {
    sw.send([w,'',c,'',m])
    tout[w]=setTimeout(resend(w,c,m), ansInterval) }

sc.on('message', (c,sep,m) => dispatch(c,m))

sw.on('message', (w,sep,c,sep2,r) => {
     if (failed[w]) return             // ignore msg from failed worker
     if (tout[w]) {                    // ans received in-time
         clearTimeout(tout[w])         //    cancel timeout
         delete tout[w]
     }
     if (c) sc.send([c,'',r])          // If it was a response, send resp to client
     if (who.length)                        // If there are pending request,
         sendToW(w,who.shift(),req.shift()) //    process first pending req
     else
         workers.push(w)                    // add as available worker
})
```

7.2. Equilibrado de carga

El broker basado en router/router permite remitir cada petición al trabajador que queramos. Por simplicidad, el código proporcionado se limita a aplicar un turno rotativo, pero sería deseable elegir el trabajador con menor carga efectiva (equilibrado de carga).

En un sistema en producción cada trabajador se ejecuta en un nodo distinto, y puede obtener la carga de su nodo para comunicarla al broker. Por su parte el broker debe recibir y mantener

información sobre la carga de cada uno de los trabajadores, y tener en cuenta esos valores a la hora de seleccionar qué trabajador gestiona una petición.

Debemos definir una estrategia para actualizar la información de carga de cada trabajador. Observamos que:

- Los trabajadores no necesitan informar mientras procesan peticiones
- Un trabajador que espera peticiones no modifica su carga mientras está en espera

Con estas premisas, un trabajador debe informar de su carga cada vez que va a pasar a espera (ej.- tras darse de alta o tras responder a una petición):

- Un trabajador notifica su carga en todo mensaje 'convencional' que envía al broker → alta o respuesta a una petición. No necesitamos mensajes específicos para comunicar la carga
- Cuando el broker recibe una petición, la redirige al trabajador que está esperando solicitudes y ha comunicado menor carga

Para representar a los trabajadores en espera y su carga, tenemos distintas alternativas:

1. Mantener array desordenado (ej. insertar al final). Para elegir el de mínima carga hay que recorrer todo el vector (coste lineal)
2. Mantener array ordenado (carga creciente): complica la inserción (coste lineal), pero la elección de mínimo es trivial (el primero del vector)
3. Utilizar una estructura de datos más elaborada (ej min-heap), con costes logarítmicos de búsqueda e inserción

7.3. Tipos de trabajos

Hasta ahora hemos asumido:

- Un único tipo de petición por parte de los clientes
- Workers homogéneos

En conclusión los workers son equivalentes → cualquier worker puede aceptar cualquier petición
Pero en la práctica:

- Podemos tener distintos tipos de solicitudes
- Distintos workers pueden tener capacidades específicas

de forma que según el tipo de solicitud es preferible dirigirlo a un tipo de worker u otro

Para implementar esa especialización de los workers, asumimos que:

- Los clientes indican en cada solicitud el tipo de petición. Al arrancar un cliente le pasaremos como argumento el tipo de peticiones que realiza (ej suponemos tipos A,B,C)
- Cuando un trabajador se da de alta, indica el tipo de peticiones que puede atender. Al arrancar el trabajador, le pasaremos como argumento el tipo de peticiones que puede atender (ej A,B o C)

El broker clasificará los trabajadores según su tipo, y dirige cada petición a uno de los trabajadores que pueden atender ese tipo de petición. Una posibilidad es sustituir la lista de trabajadores por varias listas, una por tipo de trabajador: cuando llega un trabajo, se busca el trabajador en la lista del tipo correspondiente.