Patrón cliente/servidor

TSR 2021, Juansa Sendra, Grupo B

Patró cliente/servidor

- Socket tipo **req** en el cliente
  - Supongo definido `var cs = zmq.socket('req') // cs = cliente Socket`
  - `cs.connect(URL)` (URL = `'ftp://IPserver:PortServer'`)
  - Envío: `cs.send(msg) // msg puede ser multisegmento`
  - Para recibir (evento) `cs.on('message', callback)`
- Socket tipo **rep** en el servidor
  - Supongo definido `var ss = zmq.socket('rep') // ss = server Socket`
  - `ss.bind(tcp://*:portServer)`
  - Envío: `ss.send(msg) // msg puede ser multisegmento`
  - para recibir `ss.on('message', callback)`

Colas

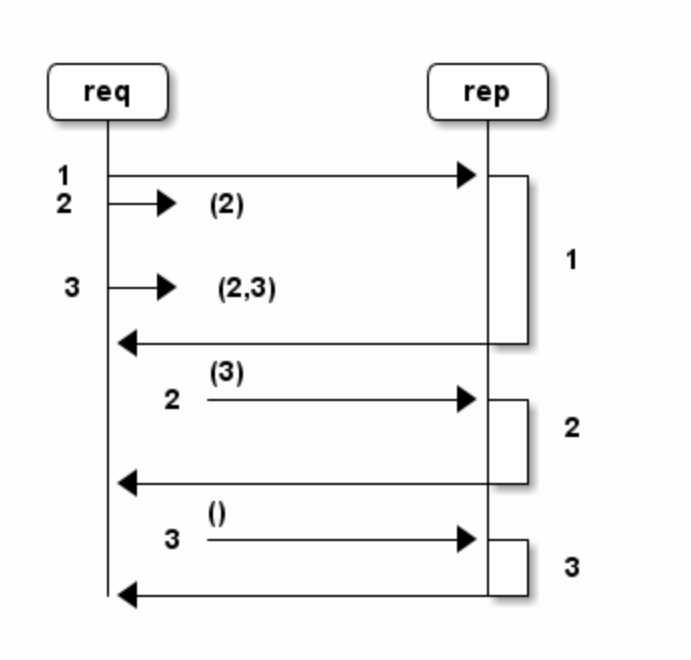
- entrada (recepción)
  - mantiene los mensajes que llegan hasta pasarlos a la aplicación
  - la llegada de un mensaje genera el evento 'message'
- salida (envío)
  - mantiene los mensajes a enviar a otros agentes
  - guarda los mensajes enviados por la aplicación

	entrada	**salida**
req	1 cola	1 cola
rep	1 cola	1 cola

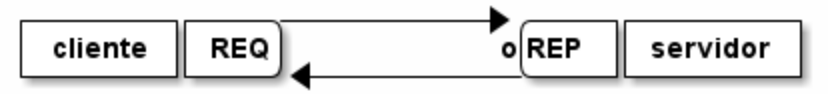
Funcionamento sincrónico

- Envío y recepción NO son bloqueantes $\rightarrow$ emisor y receptor asincrónicos
  - Pero todos los pares petición/respuesta están totalmente ordenados
  - $\rightarrow$ NO podemos tener más de una solicitud pendiente de respuesta
- Parte cliente
  - El cliente envía petición **m1** a través de **req** $\rightarrow$ m1 llega al socket rep (servidor)
  - Si el cliente envia otra solicitud **m2**, se queda en la cola de salida de req
  - Cuando llega la respuesta a **m1**, se envía **m2** desde la cola de salida de req
- Parte servidor
  - Cuando **m1** llega a la cola de entrada de rep, la aplicación lo recoge y procesa
  - Si llega **m3** (de otro cliente), se mantiene en la cola de entrada de rep
  - Cuando se envía la respuesta a **m1**, saca de la cola **m3**
  - Sólo se envía respuesta desde rep si ya hemos recibido solicitud en ese socket
    - el envío se bloquea hasta recibir la petición correspondiente

Funcionamento sincrónico



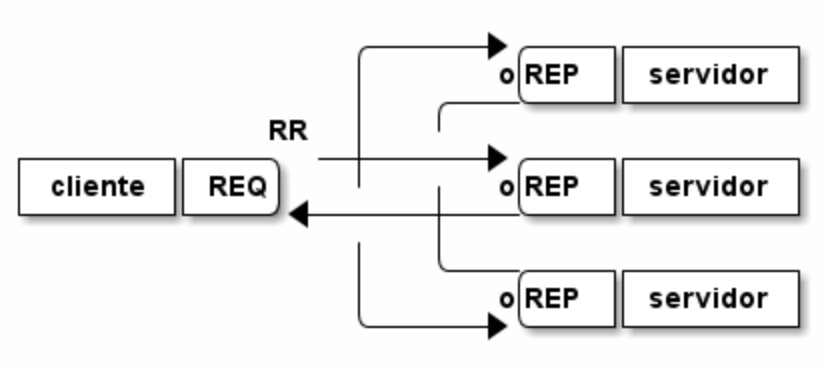
Ejemplo 1:1




```
// server.js
const zmq = require('zeromq')
const rp= zmq.socket('rep')
rp.bind('tcp://*:8888',function(err) {if(err) throw err})
rp.on('message', (msg)=>{console.log(’Request: '+msg); rp.send(’World')})
```

```
// cliente.js
const zmq = require('zeromq')
const rq= zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.send(’Hello')
rq.on('message', (msg)=>{console.log('Response: '+msg)})
```

Examplo 1:n



- Cada petición va a un rep diferente $\rightarrow$ política RR (Round Robin)
- NO envia nueva petición hasta recibir resp a la actual $\rightarrow$ No paraleliza

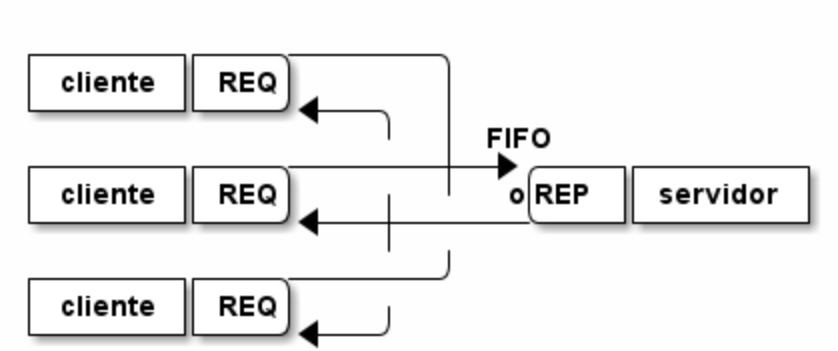
Examplo 1:n

```
// server1.js
const zmq = require('zeromq')
const rp= zmq.socket('rep')
rp.bind('tcp://*:8889',function(err) {if(err) throw err})
rp.on('message', (msg)=>{console.log(’Request: '+msg); rp.send(’World2')})
```

```
// server2.js
const zmq = require('zeromq')
const rp= zmq.socket('rep')
rp.bind('tcp://*:8888',function(err) {if(err) throw err})
rp.on('message', (msg)=>{console.log(’Request: '+msg); rp.send(’World1')})
```

```
// cliente.js
const zmq = require('zeromq')
const rq= zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.connect('tcp://127.0.0.1:8889')
rq.send(’Hello1'); rq.send(’Hello2'); rq.send(’Hello3')
rq.on('message', (msg)=>{console.log('Response: '+msg)})
```

Examplo n:1



- Configuración típica para un servidor
- El socket rep gestiona los mensajes de entrada con una cola FIFO
  - atiende a todos los clientes de manera equitativa (no hay inanición)

Examplo n:1

```
// server.js
const zmq = require('zeromq')
const rp= zmq.socket('rep')
rp.bind('tcp://*:8888',function(err) {if(err) throw err})
rp.on('message', (msg)=>{console.log(’Request: '+msg); rp.send(’World')})
```

```
// cliente1.js
const zmq = require('zeromq')
const rq= zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.send(’Hello1')
rq.on('message', (msg)=>{console.log('Response: '+msg)})
```

```
// cliente2.js
const zmq = require('zeromq')
const rq= zmq.socket('req')
rq.connect('tcp://127.0.0.1:8888')
rq.send(’Hello2')
rq.on('message', (msg)=>{console.log('Response: '+msg)})
```

Modificación del mensaje (formato mensajes)

- Estructura mensaje típico `[envoltorio delimitador cuerpo]`
  - envoltorio = metadades associades al mensaje
    - 1 o mas segmentos
  - delimitador = segmento vacío que separa el envoltorio y cuerpo (en la figura es representa utilizando una ,)
    - 1 o mas segmentos

	envío	**recepción**
req	añade delimitador (1er segmento)	elimina delimitador (1er segmento)
rep	guarda envoltorio, pasa cuerpo a la aplic.	añade envoltorio guardado

Modificación del mensaje (formato mensajes)