



## Seminario 3 – 0MQ



Tecnologías de los Sistemas de Información en la Red



# Índice

---

1. Introducción
2. Mensajes
3. 0MQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía



# I. Introducción: Objetivos de 0MQ

---

- ▶ Middleware de comunicaciones simple
  - ▶ Configuración sencilla: URL para nombrar “endpoints”
  - ▶ Uso cómodo y familiar: API similar a los sockets BSD
- ▶ Ampliamente disponible
  - ▶ Implementación migrable
- ▶ Soporta patrones básicos de interacción
  - ▶ Elimina la necesidad de que cada desarrollador “reinvente la rueda”
  - ▶ Fácil de usar (de manera inmediata)
- ▶ Rendimiento
  - ▶ Sin sobrecargas innecesarias
  - ▶ Compromiso entre fiabilidad y eficiencia
- ▶ El mismo código puede utilizarse para comunicar
  - ▶ Hilos en un proceso
  - ▶ Procesos en una máquina
  - ▶ Ordenadores en una red IP
  - ▶ Sólo se necesitan cambios en la configuración



# I. Introducción: Características principales

---

- ▶ Comunicación basada en mensajes
  - ▶ Persistencia débil: colas en memoria principal
- ▶ Es sólo una biblioteca
  - ▶ No se necesita arrancar ningún servidor específico (middleware)
  - ▶ Implantada en C++
  - ▶ Disponible en la mayoría de los sistemas operativos
    - ▶ Linux\_XYZ
    - ▶ Windows
    - ▶ BSD
    - ▶ MacOS X
  - ▶ *Bindings* disponibles para muchos lenguajes y entornos de programación

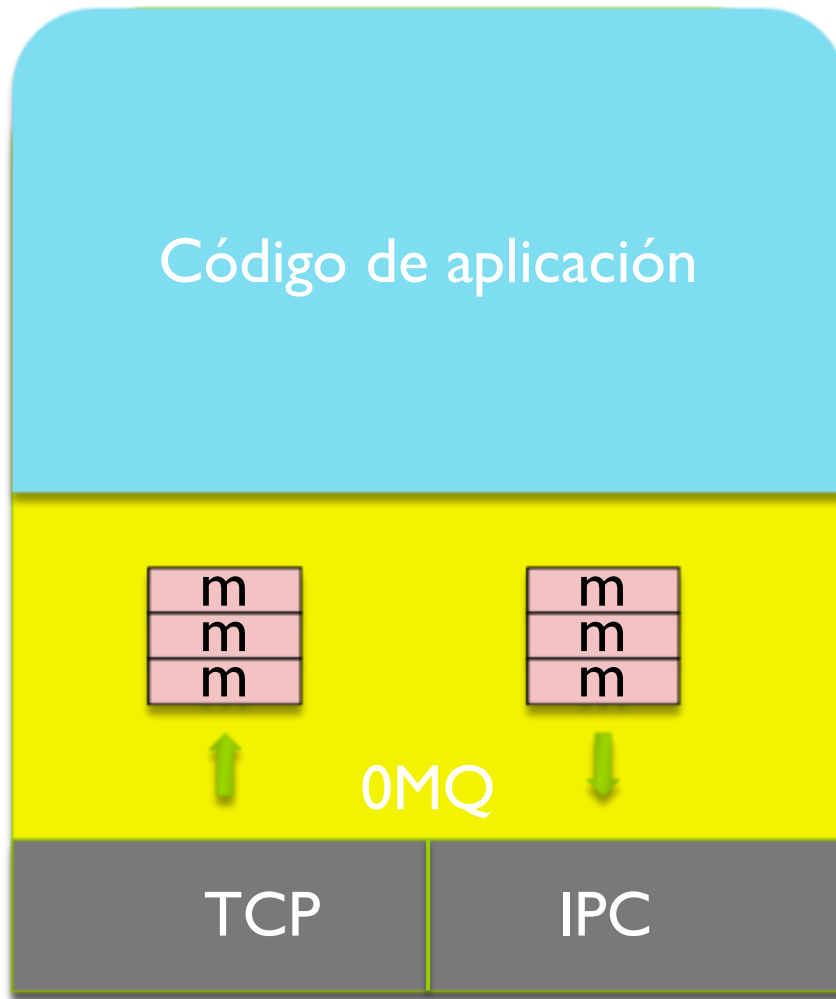


# I. Introducción: Tecnología

---

- ▶ Proporciona sockets para enviar y recibir mensajes
  - ▶ send/receive, bind/connect interfaz para los sockets
- ▶ Puede utilizar estos transportes:
  - ▶ Entre procesos
    - ▶ TCP/IP
    - ▶ Multicast fiable (pgm)
    - ▶ IPC (Sockets Unix)
  - ▶ Intra-proceso
    - ▶ Colas (en memoria)
    - ▶ Útiles para hilos concurrentes
- ▶ Transporte utilizado para instanciar un socket
  - ▶ Fácilmente modificable mediante un cambio en la configuración

# I. Introducción: Vista de un proceso 0MQ



- ▶ La aplicación enlaza con la biblioteca 0MQ
- ▶ 0MQ mantiene colas en memoria
  - ▶ En el emisor
  - ▶ En el receptor
- ▶ 0MQ usa niveles de comunicación
- ▶ Potencialmente, con más de un contexto
  - ▶ P.ej., para comunicación entre hilos



# I. Introducción: Instalación

## ▶ Desde los fuentes, en Linux

- ▶ `sudo apt-get install build-essential libtool autoconf automake uuid-dev`
- ▶ `wget http://download.zeromq.org/zeromq-3.2.3.tar.gz`
- ▶ `tar xvzf zeromq-3.2.3.tar.gz`
- ▶ `./configure`
- ▶ `make`
- ▶ `sudo make install`
- ▶ `sudo ldconfig`

## ▶ Hay también instaladores de los paquetes binarios

## ▶ En node

- ▶ `npm install zmq`

```
var zmq = require('zmq');
```



# Índice

---

1. Introducción
2. Mensajes
3. 0MQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía





## 2. Mensajes: Middleware orientado a mensajes

---

- ▶ Los mensajes es lo que se envía
  - ▶ No hay problemas de empaquetado (“framing”) para la aplicación
  - ▶ La gestión de “buffers” también está resuelta
- ▶ Los mensajes pueden ser “multi-parte”
  - ▶ El soporte para estructurar los mensajes resulta sencillo
- ▶ Los mensajes se entregan atómicamente
  - ▶ Se entregan todas las partes o nada se entrega
- ▶ Tanto el envío como la recepción son asincrónicos
  - ▶ Internamente, 0MQ gestiona el flujo de mensajes entre las colas (de los procesos) y los transportes
- ▶ La gestión de la conexión y reconexión entre agentes es automática



## 2. Mensajes

- ▶ El contenido de los mensajes resulta transparente para 0MQ
- ▶ No se necesita soporte para “marshalling”
  - ▶ No hay que preocuparse por la codificación
  - ▶ Los mensajes son “blobs” para 0MQ
  - ▶ Pero el API de 0MQ soporta una serialización sencilla de cadenas en los mensajes

```
zsock.send("Esto es", "un", "mensaje");
```

7	Esto es
2	un
7	mensaje

- ▶ **NOTA**
  - ▶ Algunos tipos de socket utilizan el primer segmento



## 2. Mensajes: Consecuencias

- ▶ El programador debe decidir cómo estructurar el contenido del mensaje
- ▶ En muchos casos, puede ser tan sencillo como una cadena
- ▶ Se puede utilizar CUALQUIER codificación
  - ▶ Binaria, por ejemplo
- ▶ Aproximación sencilla: mensajes XML
  - ▶ Se utilizarán parsers XML
- ▶ Aproximación algo más sencilla: mensajes JSON
- ▶ La aproximación más sencilla
  - ▶ Utilizar cada segmento para una pieza de información distinta, con su propia codificación. Ejemplo:
    - ▶ Segmento 1: nombre de la interfaz invocada, en formato cadena
    - ▶ Segmento 2: versión de la API de la interfaz, como una cadena
    - ▶ Segmento 3: nombre de la operación
    - ▶ Segmento 4: primer argumento (un entero)
    - ▶ Segmento ...



# Índice

---

1. Introducción
2. Mensajes
3. 0MQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía



## 3.API 0MQ

---

### 1. Sockets

- ▶ Envío y recepción utilizan sockets
- ▶ Varios tipos de sockets
- ▶ Operaciones bind/connect

### 2. Patrones de comunicación

- ▶ Soportados por tipos específicos de socket



## 3.1. Sockets 0MQ

- ▶ La creación de un socket es sencilla:

```
var zmq = require('zmq');  
  
var zsock = zmq.socket(<TIPO SOCKET>);
```

- ▶ Donde <TIPO SOCKET> será uno de los siguientes

req	push	pub
rep	pull	sub
dealer	pair	xsub
router		xpub

- ▶ Qué tipos utilizar dependerá de los patrones de conexión en los que intervenga

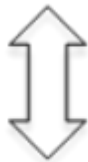


## 3.1. Sockets: Estableciendo vías de comunicación

- ▶ Un proceso realiza un **bind**
- ▶ Otros procesos realizan **connect**
- ▶ Cuando terminen, **close**
- ▶ **bind/connect** están desacoplados: no hay requisitos sobre su ordenación

```
sock.bind("tcp://10.0.0.1:5555",  
  function(err) { .. });
```

5555



10.0.0.1

```
so.connect("tcp://10.0.0.1:5555",  
  function(err) { .. });
```





## 3.1. Sockets: Múltiples conexiones son posibles

```
so1
.connect("tcp://10.0.0.1:5555",
  function(err) { .. });
so1
.connect("tcp://10.0.0.2:5556",
  function(err) { .. });
```



10.0.0.1



```
sock.bind("tcp://10.0.0.1:5555",
  function(err) { .. });
```

5555

```
so2
.connect("tcp://10.0.0.1:5555",
  function(err) { .. });
```



10.0.0.2



```
sock.bind("tcp://10.0.0.2:5556",
  function(err) { .. });
```

5556

```
so3
.connect("tcp://10.0.0.2:5556",
  function(err) { .. });
```







## 3.1. Sockets: Conexiones y colas

---

- ▶ Los sockets tienen colas de mensajes asociadas
  - ▶ De entrada (recepción), para mantener los mensajes que hayan llegado
    - ▶ Generan el evento “message” cuando mantienen algún mensaje
  - ▶ De salida (envío), manteniendo los mensajes a enviar a otros agentes
    - ▶ Donde se guardan los mensajes enviados por la aplicación
- ▶ Los sockets “router” mantienen un par de colas (entrada/salida) por agente conectado
  - ▶ El resto de los sockets no distinguen entre agentes
  - ▶ Los sockets “pub” quedan fuera de esta discusión
- ▶ Los sockets “pull” y “sub” solo mantienen una cola de entrada
- ▶ Los sockets “push” y “pub” solo mantienen una cola de salida



## 3.1. Sockets: bind / connect

- ▶ ¿Cuándo realizar un **bind** y cuándo un **connect**?
  - ▶ En la mayoría de los casos no importa: gestionado en la configuración
- ▶ Observaciones
  - ▶ Todos los agentes coincidirán en algún “endpoint”
  - ▶ Los “endpoints” se referencian mediante sus URL
  - ▶ En el transporte TCP
    - ▶ La dirección IP debe pertenecer a una de las interfaces del socket (bind)
      - bind: El socket solo necesita una configuración IP local (o ninguna)
        - No necesita conocer dónde están los demás agentes
    - ▶ El socket que realice un “connect” necesita conocer la dirección IP del socket que realice un “bind”



## 3.1. Sockets: Transportes: TCP

- ▶ URL: `tcp://<dirección>:<puerto>`
- ▶ Tres maneras de especificar la dirección

```
sock.bind("tcp://192.168.0.1:9999");
```

```
sock.bind("tcp://*:9999");
```

```
sock.bind("tcp://eth0:9999");
```

- ▶ \*: **bind** sobre todas las interfaces
- ▶ “eth0”: **bind** sobre todas las direcciones asociadas a la interfaz “eth0”



## 3.1. Sockets: Transportes: IPC

---

- ▶ *Inter Process Communication* (Sockets Unix)
- ▶ URL: `ipc://<ruta-del-socket>`

```
sock.bind("ipc:///tmp/myapp");
```

- ▶ Se necesita permiso `rw` (lectura y escritura) sobre el socket en `<ruta-del-socket>`



## 3.1. Sockets: Transportes: intra-proceso

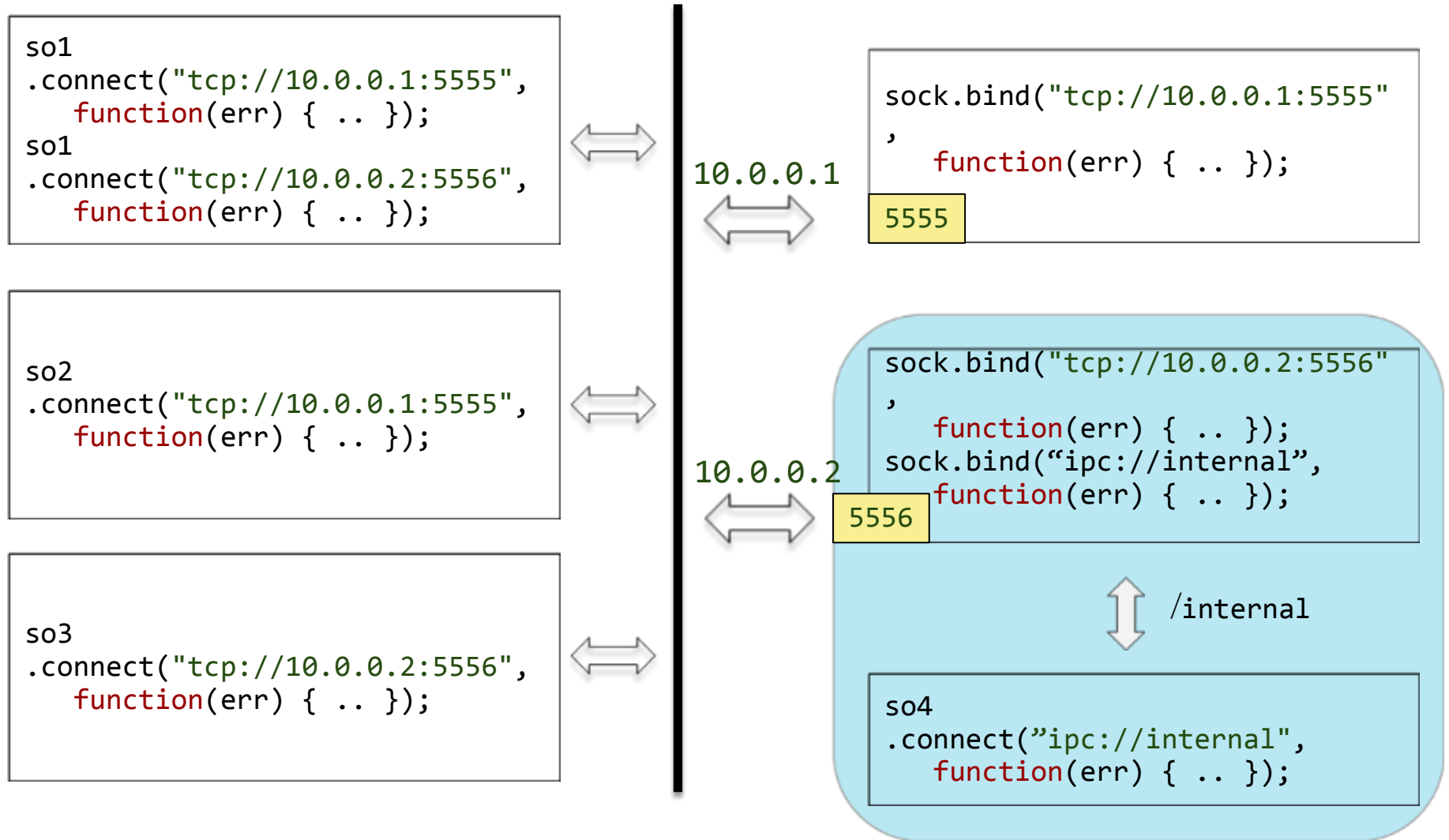
- ▶ Mecanismo de comunicación intra-proceso
- ▶ Soportado directamente por 0MQ
- ▶ Utiliza estructuras de datos compartidas por todos los hilos del proceso
  - ▶ En nodejs solo hay un hilo
- ▶ URL: `inproc://<nombre-cola>`

```
sock.bind("inproc://appqueue");
```

- ▶ Restricciones
  - ▶ Longitud máxima para el nombre: 256
  - ▶ Se DEBE hacer un bind ANTES DE cualquier connect



# 3.1. Sockets: el uso de múltiples transportes es posible





## 3.1. Sockets: Envío de mensajes

- ▶ Envío multi-segmento, en varias llamadas

```
sock.send("Segmento 1", zmq.ZMQ_SNDMORE);  
sock.send("Segmento 2");
```

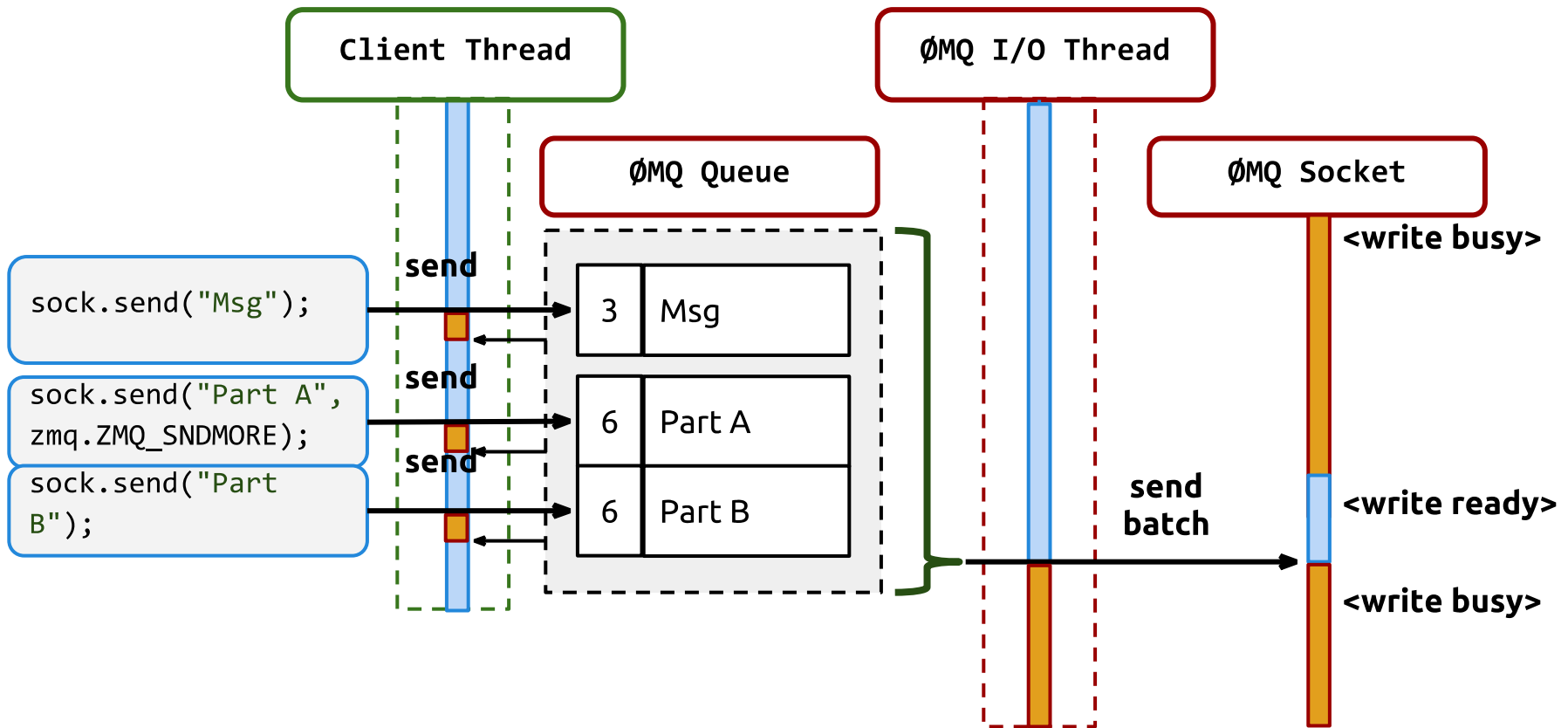
- ▶ Los segmentos pueden extraerse de un vector en una misma llamada

```
sock.send(["Segmento 1", "Segmento 2"]);
```

- ▶ Los segmentos deben ser **buffers** o **cadena**s.
  - ▶ Las cadenas se convierten en buffers, utilizando codificación UTF8
    - ▶ Lo que no sea cadena se convierte primero a cadena



### 3.1. Sockets: aspectos internos del envío (batching)







## 3.1. Sockets: Recepción

- ▶ Basado en eventos “message” del socket
  - ▶ Los **argumentos** del manejador contienen los segmentos del mensaje
  - ▶ **NOTA:** Los segmentos son buffers binarios

```
sock.on("message", function(first_part, second_part){  
    console.log(first_part.toString());  
    console.log(second_part.toString());  
});
```

- ▶ Para un número variable de segmentos, usar “**arguments**” directamente...

```
sock.on("message", function() {  
    for (var key in arguments) {  
        console.log("Part" + key + ": " + arguments[key]);  
    };  
});
```

- ▶ ... o convertir antes en vector

```
var segments = Array.prototype.slice.call(arguments);  
segments.forEach(function(seg) { ... });
```



## 3.1 Sockets: Opciones

- ▶ Hay muchas.
- ▶ Dos importantes: **identity**, y **subscribe**

```
sock.setsockopt('identity', 'frontend');  
sock.setoption('subscribe', 'SOCKER');
```

- ▶ Pueden también expresarse así:

```
sock.identity = 'frontend';  
sock.subscribe('SOCKER');
```

- ▶ **identity** es conveniente a la hora de conectar con sockets “router”
  - ▶ Fija el ID del agente que se conecte al “router”
- ▶ **subscribe**, utilizado por sockets “sub”
  - ▶ Fija el filtro de prefijos aplicado al socket “pub”



## 3.2. Patrones básicos

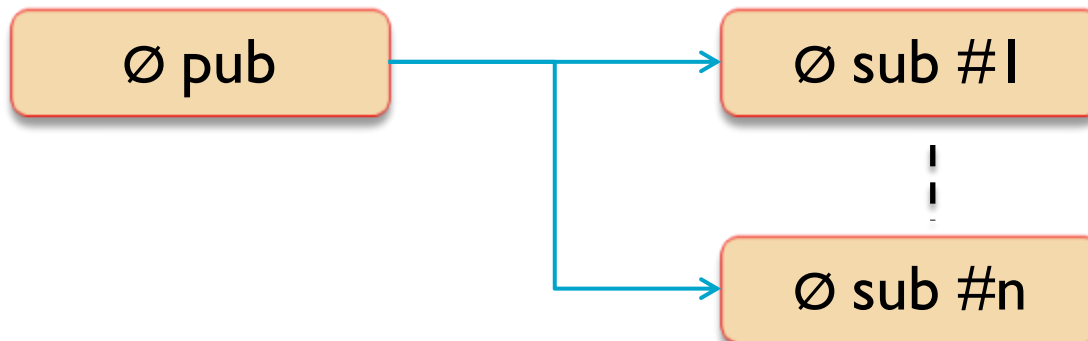
### ► Request/Reply (sincrónico)



### ► Push-pull



### ► Pub-Sub



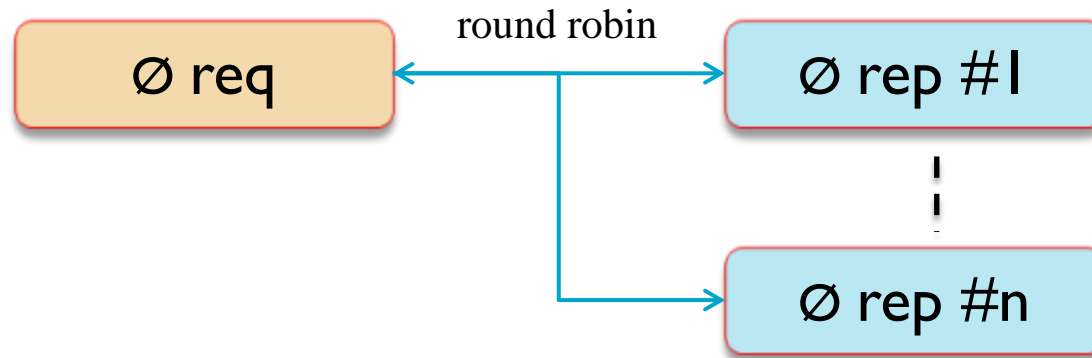


## 3.2. Patrones básicos: **request/reply**

- ▶ Implantado mediante sockets **req** en el cliente
  - ▶ sockets **rep** en el servidor
- ▶ Cada mensaje enviado vía **req** necesita asociarse a una contestación desde el socket **rep** del servidor
- ▶ Patrón de comunicación sincrónico
  - ▶ Todos los pares petición/respuesta están totalmente ordenados
  - ▶ Los “endpoints” pueden reaccionar asincrónicamente
- ▶ Cuando se ha enviado un mensaje a través de un socket **req**, otro envío posterior por ese socket será encolado localmente
  - ▶ Hasta que el mensaje de respuesta sea recibido
    - ▶ Entonces el mensaje encolado se enviará
- ▶ Cuando se ha enviado un mensaje a través de un socket **rep**, otro envío posterior por ese socket será encolado localmente
  - ▶ Hasta que un nuevo mensaje de petición sea recibido
    - ▶ Entonces el mensaje encolado se enviará: será su respuesta

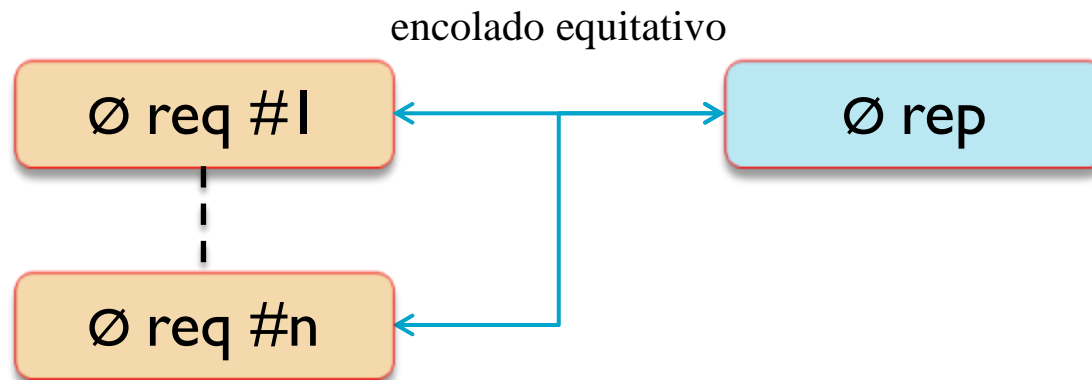


## 3.2. Patrones básicos: **request/reply** con distribución



- ▶ Cuando un **req** conecta con más de un **rep**, cada mensaje de petición se envía a un **rep** diferente
  - ▶ Se sigue una política “round-robin”
- ▶ La operación continúa siendo sincrónica:
  - ▶ OMQ no envía nuevas peticiones hasta que cada respuesta sea recibida
  - ▶ No hay paralelización de peticiones

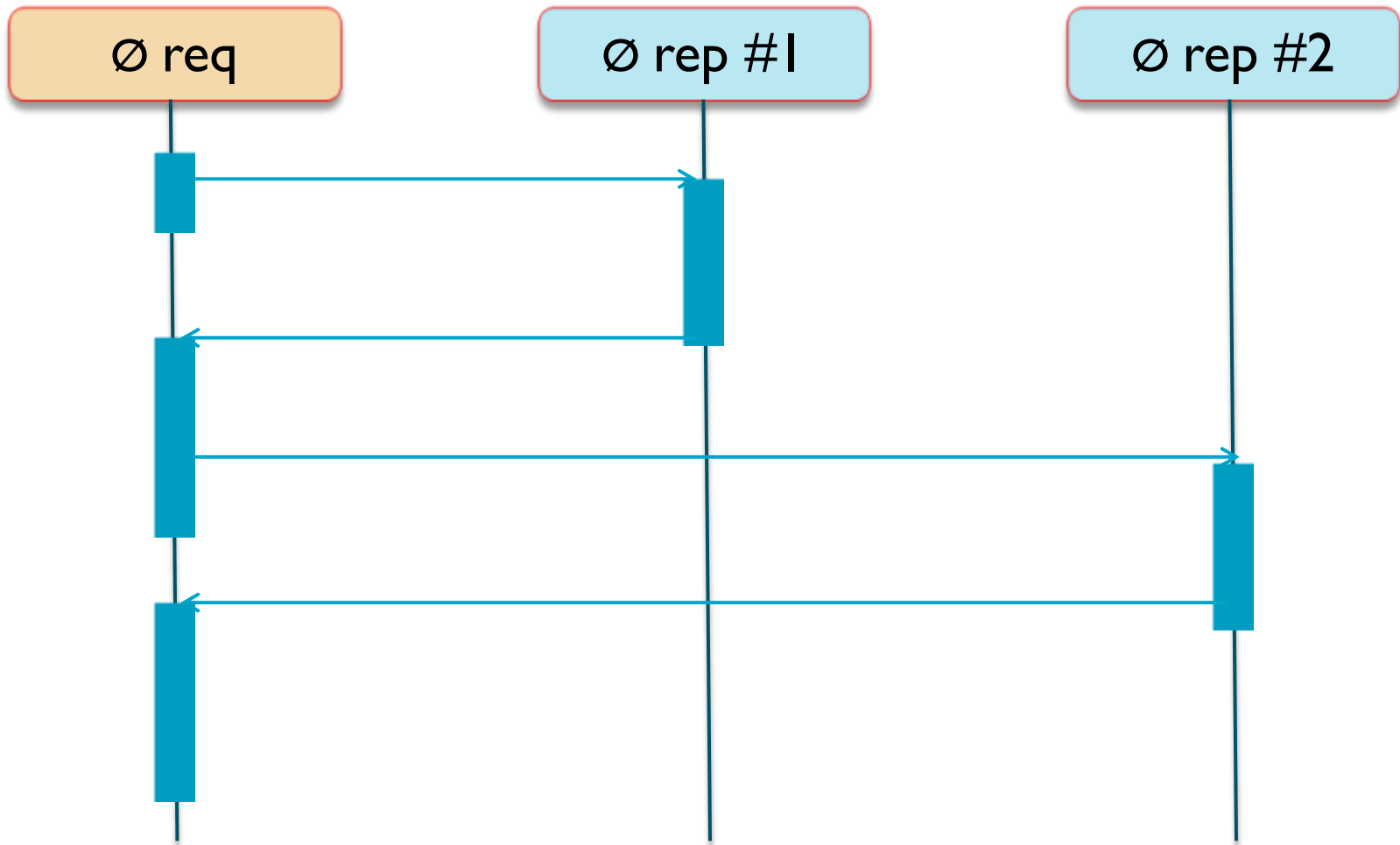
## 3.2. Patrones básicos: múltiples peticionarios



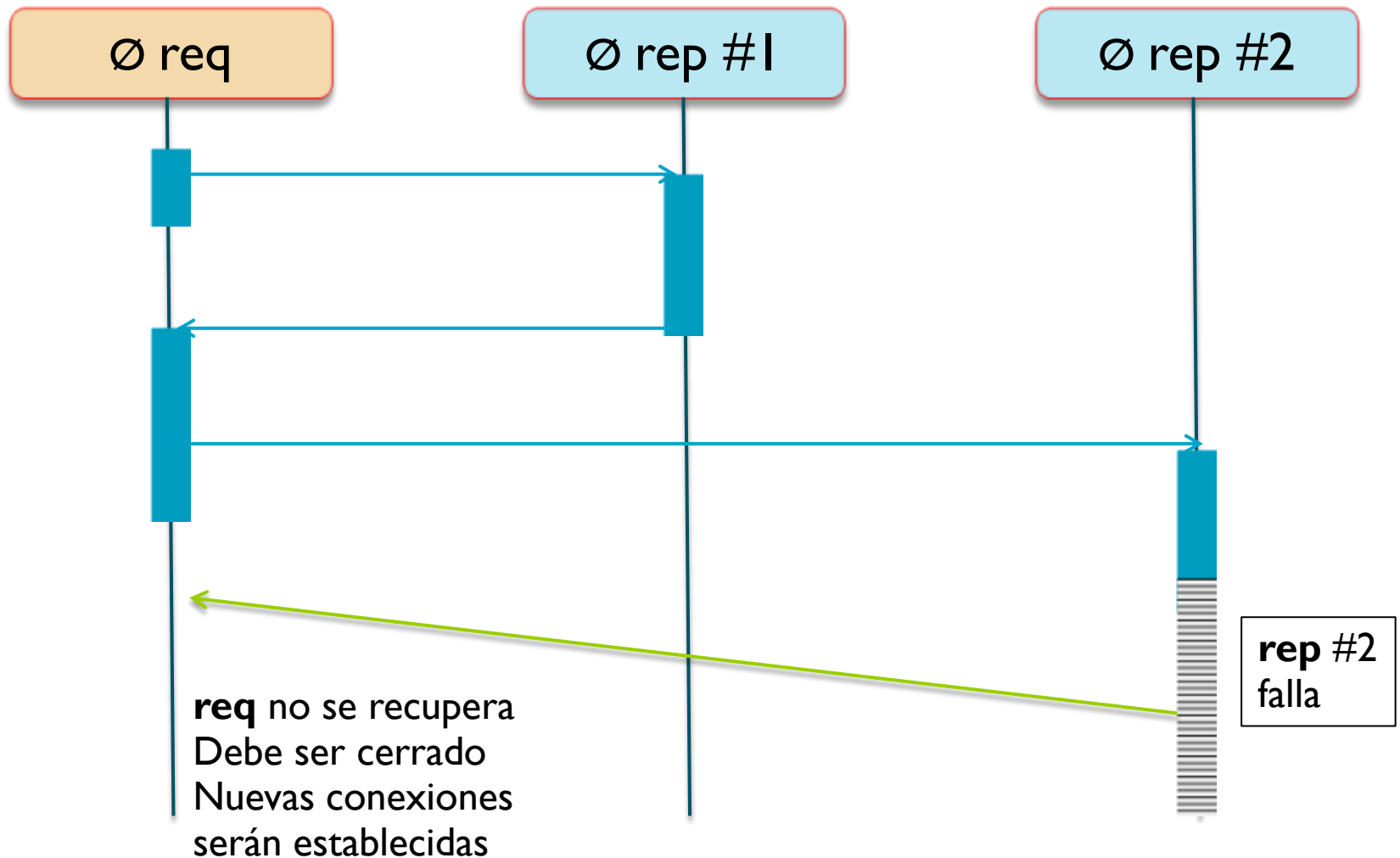
- ▶ Configuración típica para un servidor
- ▶ El socket **rep** gestiona los mensajes de entrada con una cola
  - ▶ Ningún socket **req** sufrirá inanición



## 3.2. Patrones básicos: Secuencia petición/respuesta



## 3.2. Patrones básicos: Fallos petición/respuesta







## 3.2. Patrones: req/rep básico

```
var zmq = require('zmq');  
var rq = zmq.socket('req');  
rq.connect('tcp://127.0.0.1:8888');  
rq.send('Hello');  
rq.on('message', function(msg) {  
  console.log('Response: ' + msg);  
});
```

```
var zmq = require('zmq');  
var rp = zmq.socket('rep');  
rp.bind('tcp://127.0.0.1:8888',  
  function(err) {  
    if (err) throw err;  
  });  
rp.on('message', function(msg) {  
  console.log('Request: ' + msg);  
  rp.send('World');  
});
```



## 3.2. Patrones básicos: req/rep, dos servidores

```
var zmq = require('zmq');
var rq = zmq.socket('req');
rq.connect('tcp://127.0.0.1:8888');
rq.connect('tcp://127.0.0.1:8889');
rq.send('Hello');
rq.send('Hello again');

rq.on('message', function(msg) {
  console.log('Response: ' + msg);
});
```

```
var zmq = require('zmq');
var rp = zmq.socket('rep');
rp.bind('tcp://127.0.0.1:8888',
  function(err) {
    if (err) throw err;
  });
rp.on('message', function(msg) {
  console.log('Request: ' + msg);
  rp.send('World');
});
```

```
var zmq = require('zmq');
var rp = zmq.socket('rep');
rp.bind('tcp://127.0.0.1:8889',
  function(err) {
    if (err) throw err;
  });
rp.on('message', function(msg) {
  console.log('Request: ' + msg);
  rp.send('World 2');
});
```



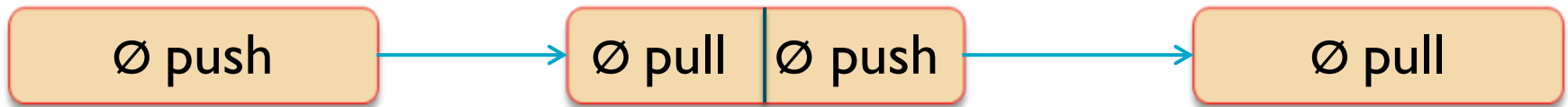
## 3.2. Patrones: req/rep, estructura de los mensajes

- ▶ Los mensajes tienen un primer segmento vacío
- ▶ Es el “delimitador”
- ▶ El socket **req** lo añade, sin que intervenga la aplicación
- ▶ El socket **rep** lo elimina antes de pasarlo a la aplicación
  - ▶ Pero lo añade de nuevo en la contestación
- ▶ El socket **req** lo eliminará de la contestación

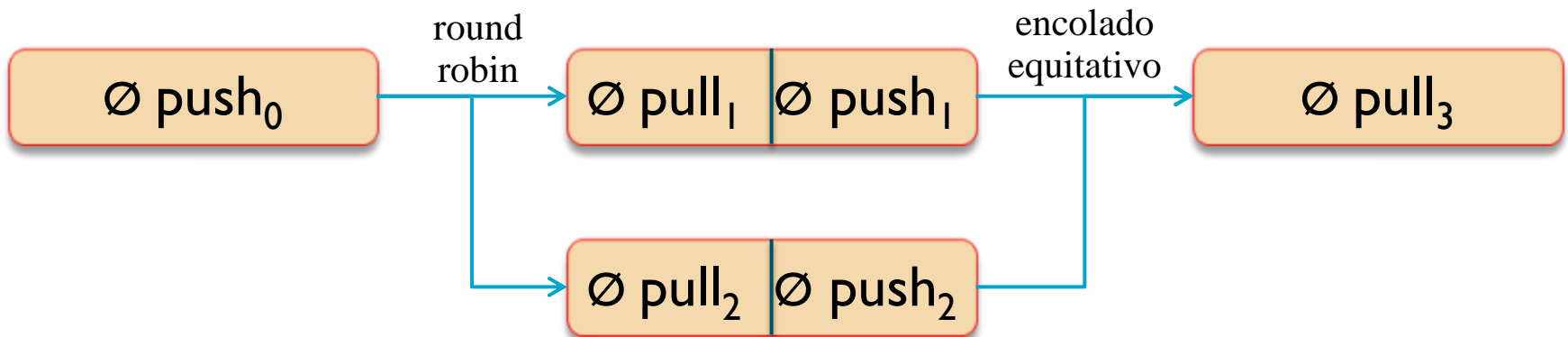
0	“”
7	“esto es”
2	“una”
8	“petición”

## 3.2. Patrones básicos: push/pull

- Distribución de datos unidireccional
- El emisor no espera ninguna respuesta
  - Los mensajes no esperan respuestas: envíos concurrentes



- Se aceptan multiples conexiones
  - P.ej., organización típica map-reduce:





## 3.2: Patrones: ejemplo push/pull, productor/consumidores

```
var zmq = require("zmq");
var producer = zmq.socket("push");
var count = 0;

producer.bind("tcp://*:8888", function(err) {
  if (err) throw err;

  setInterval(function() {
    var t = producer.send("msg nr. " + count++);
    console.log(t);
  }, 1000);
});
```

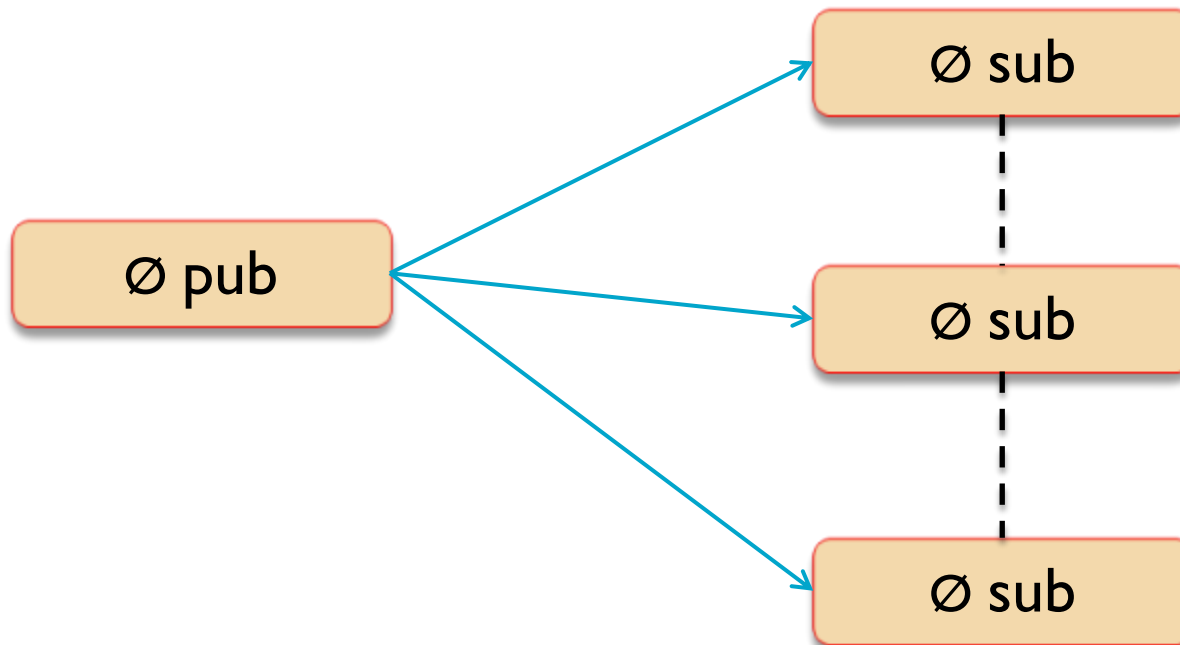
```
var zmq = require("zmq");
var consumer = zmq.socket("pull");

consumer.connect("tcp://127.0.0.1:8888");

consumer.on("message", function(msg) {
  console.log("received: " + msg);
});
```

## 3.2. Patrones básicos: Publish/Subscribe (pub/sub)

- Este patrón implanta la difusión de mensajes...

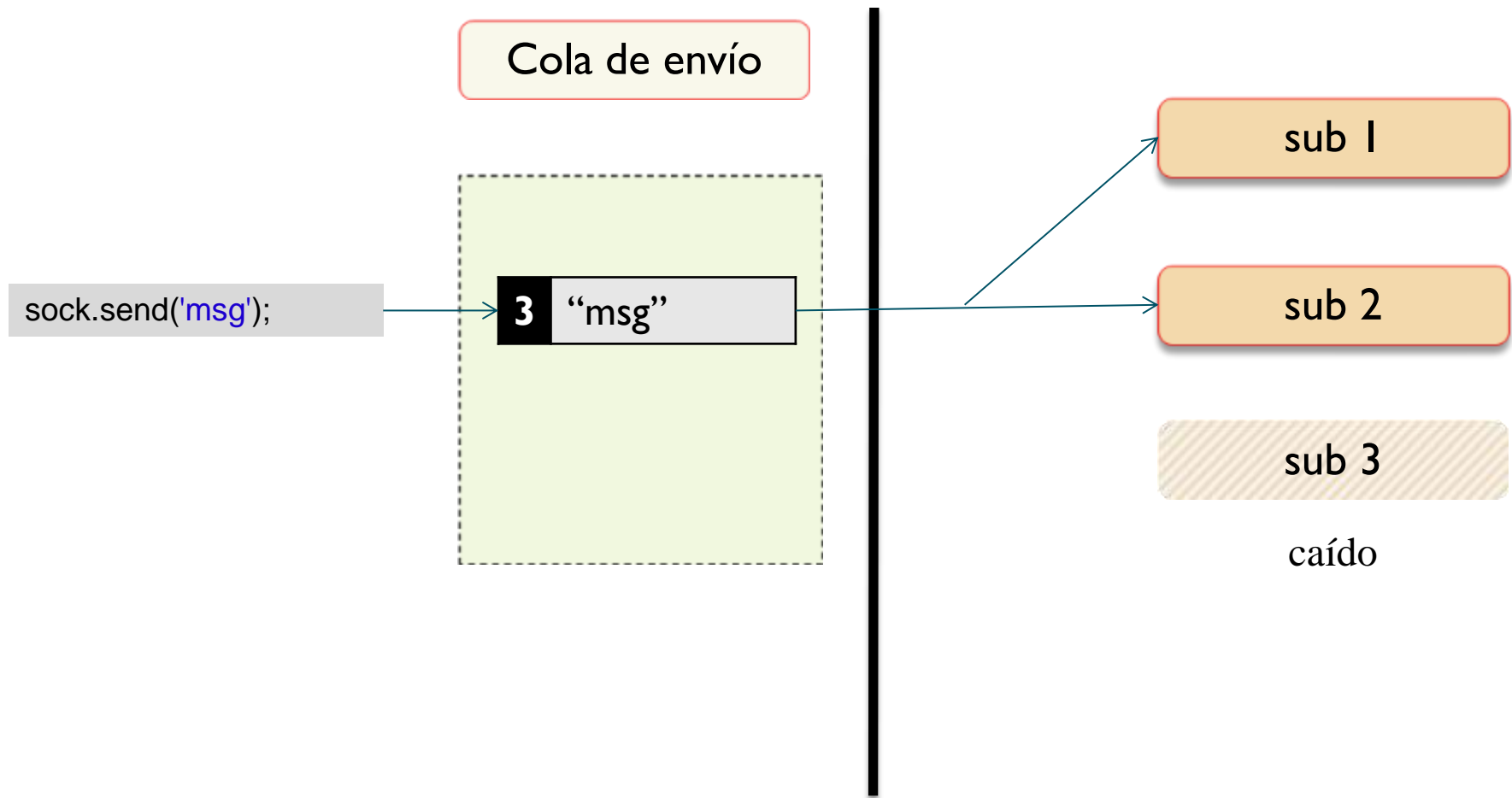


- ... con una condición: los receptores pueden decidir que se suscriben solo a ciertos mensajes
  - Entonces es un multienvío



## 3.2. Patrones básicos: pub/sub

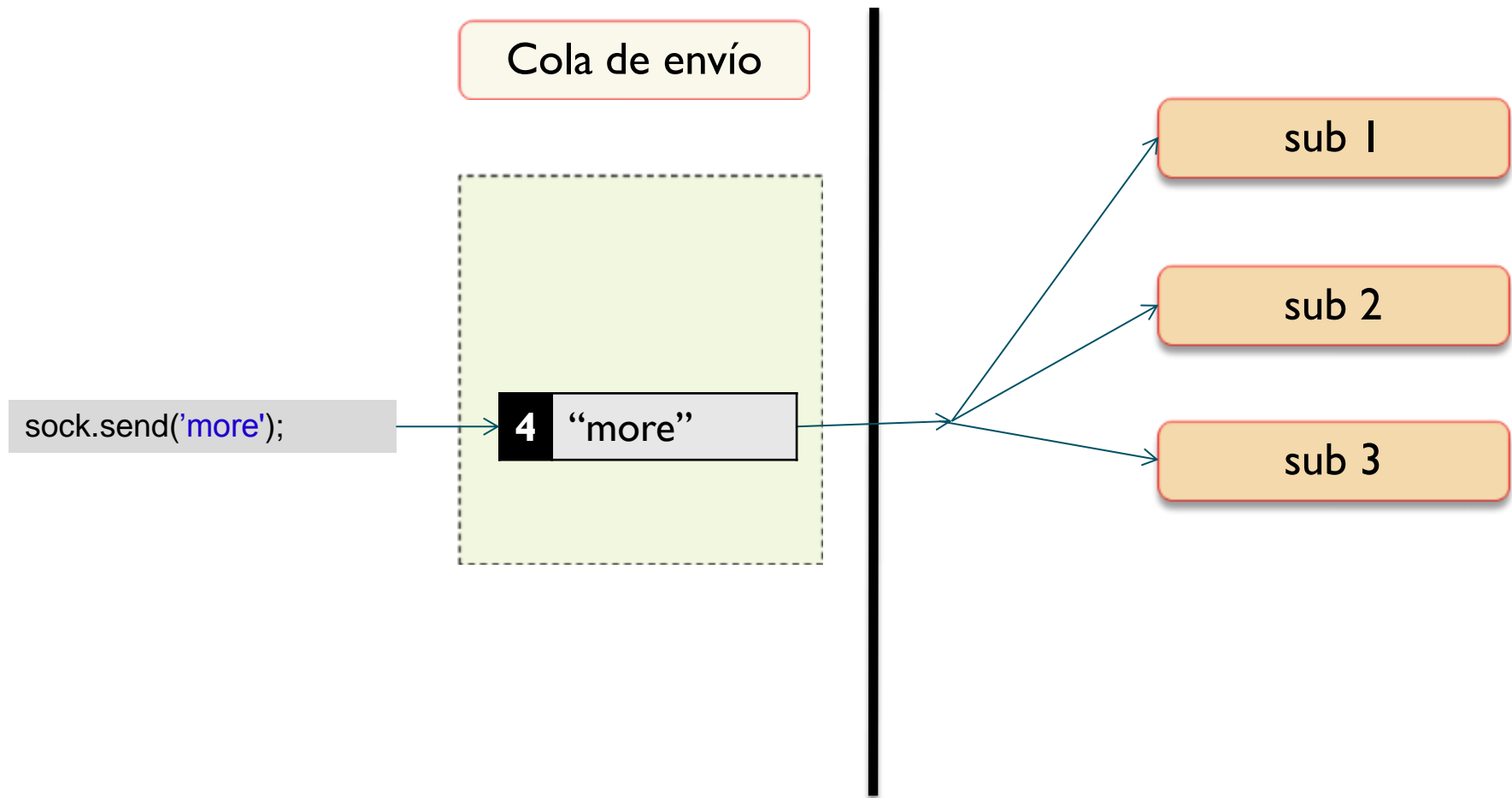
- Los mensajes son enviados a todos los agentes disponibles y conectados





## 3.2. Patrones básicos: pub/sub

- Los mensajes son enviados a todos los agentes disponibles y conectados

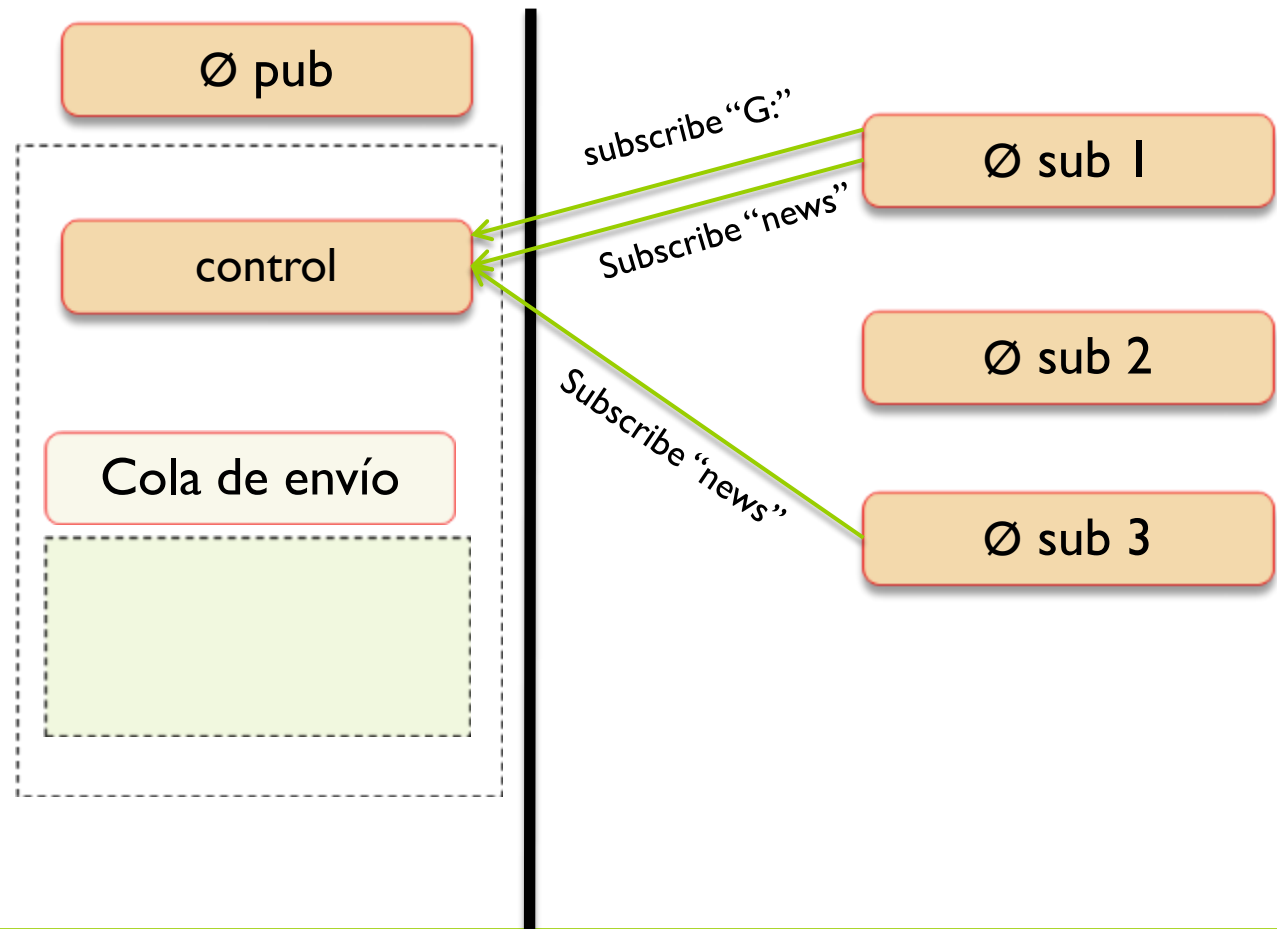






## 3.2. Patrones básicos: pub/sub: Suscripción/filtrado

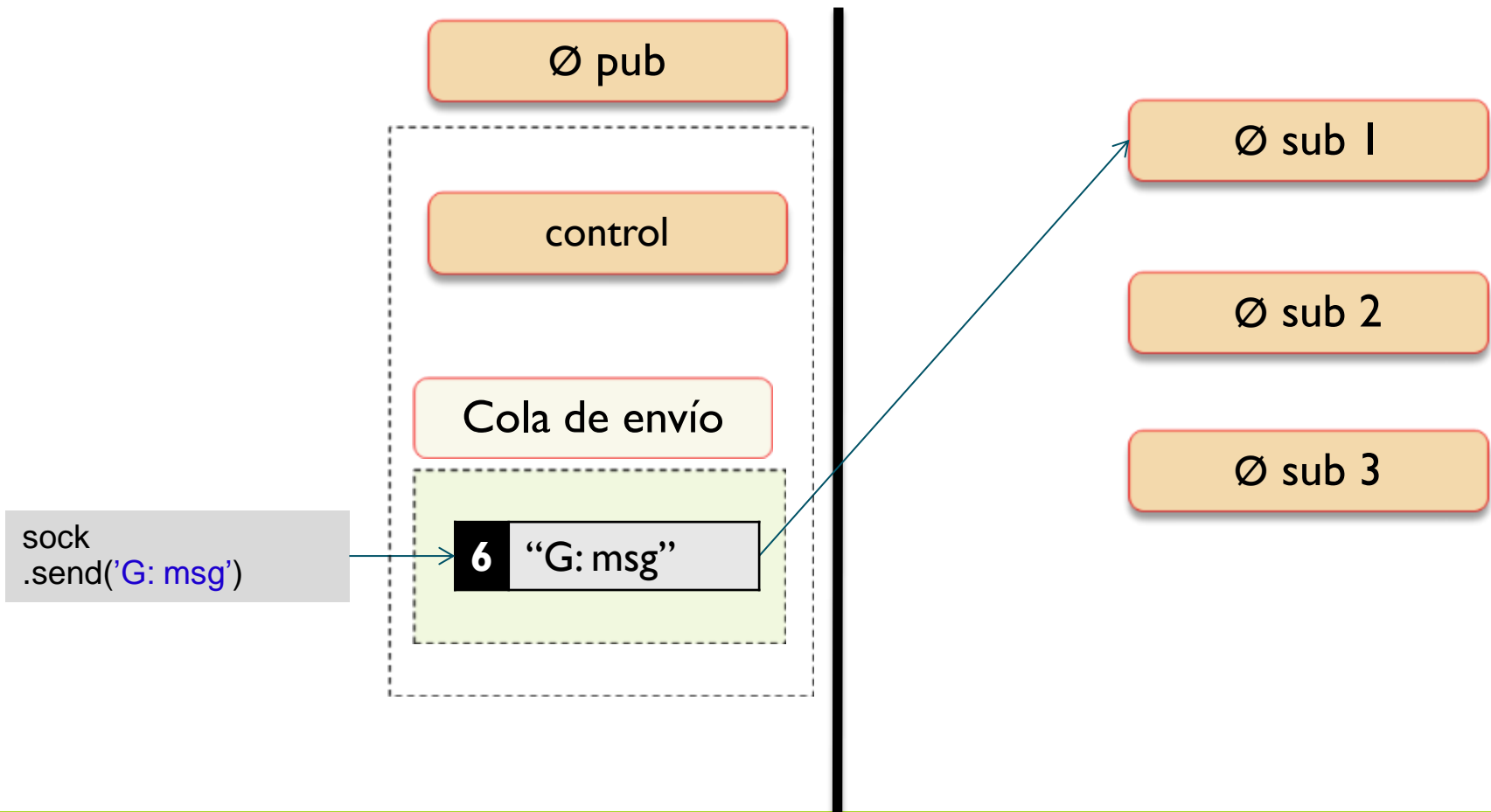
- ▶ Los suscriptores pueden especificar filtros, como prefijos de los mensajes
  - ▶ Pueden especificar varios prefijos





## 3.2. Patrones básicos: pub/sub: Suscripción/filtrado

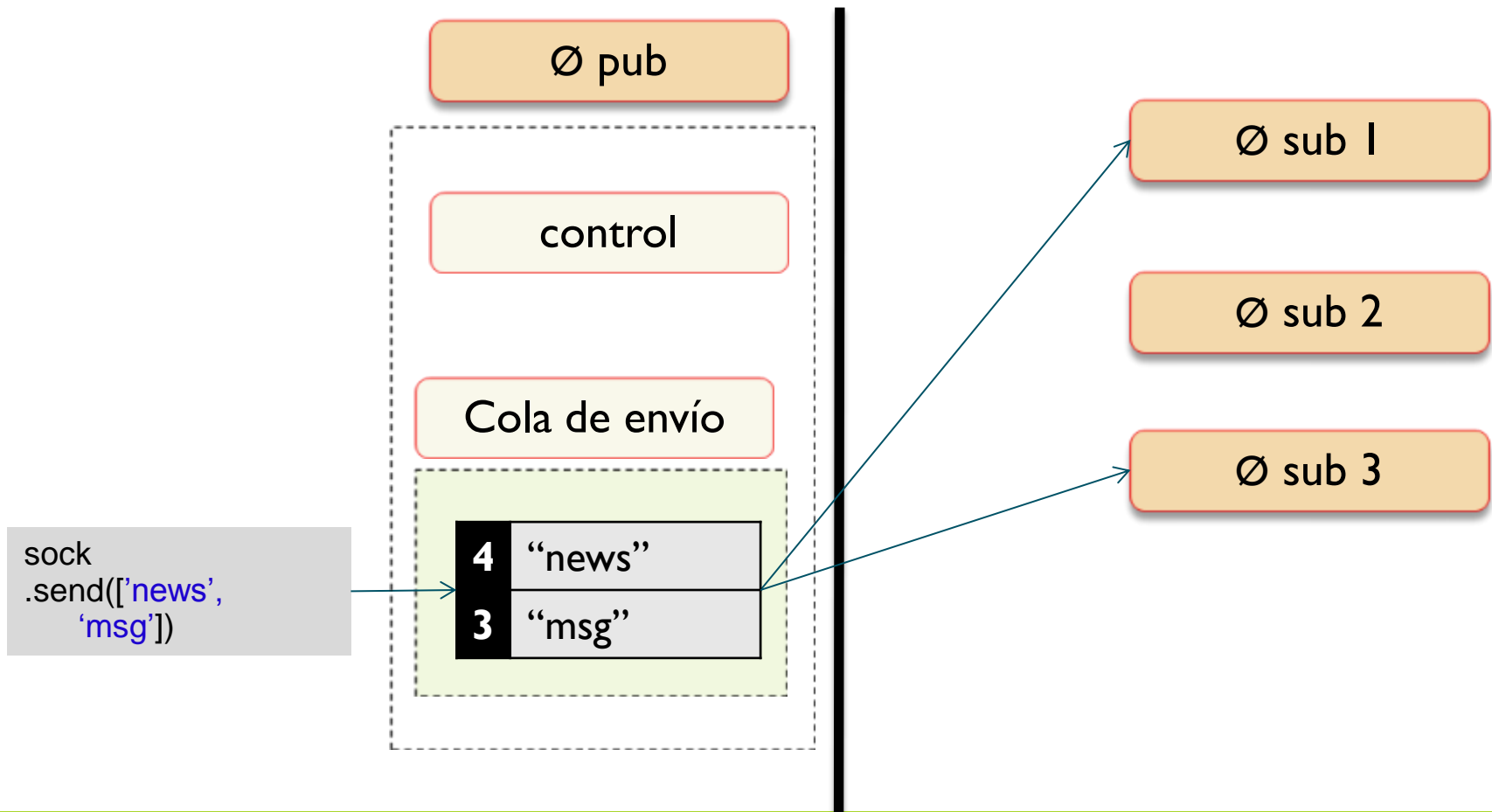
- ▶ Los suscriptores pueden especificar filtros, como prefijos de los mensajes
  - ▶ Recibirán solo los mensajes con esos prefijos





## 3.2. Patrones básicos: pub/sub: Suscripción/filtrado

- ▶ Los suscriptores pueden especificar filtros, como prefijos de los mensajes
  - ▶ Recibirán solo los mensajes con esos prefijos





## 3.2. Patrones básicos. Ejemplo pub/sub

```
var pub = zmq.socket('pub');  
var count = 0  
  
pub.bindSync("tcp://*:5555");  
  
setInterval(function() {  
  pub.send("TEST " + count++);  
}, 1000);
```

```
var sub = zmq.socket('sub');  
  
sub.connect("tcp://localhost:5555")  
sub.subscribe("TEST");  
sub.on("message", function(msg) {  
  console.log("Received: " + msg);  
});
```

Los mensajes más antiguos podrían perderse si el suscriptor empieza tarde



# Índice

---

1. Introducción
2. Mensajes
3. OMQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía

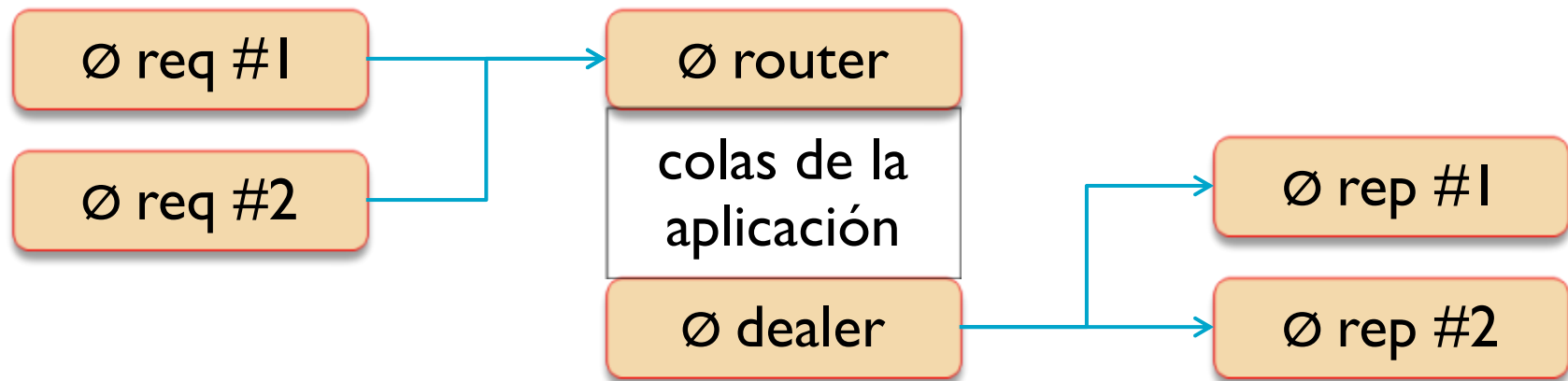
## 4. Tipos de “sockets” avanzados

### 1. Dealer

- ▶ Similar a **req**, pero asincrónico

### 2. Router

- ▶ Similar a **rep**, pero asincrónico y con capacidad para distinguir entre agentes (para encaminar las respuestas)
- ▶ Normalmente se implantan juntos en un mismo agente





## 4.1. Sockets dealer

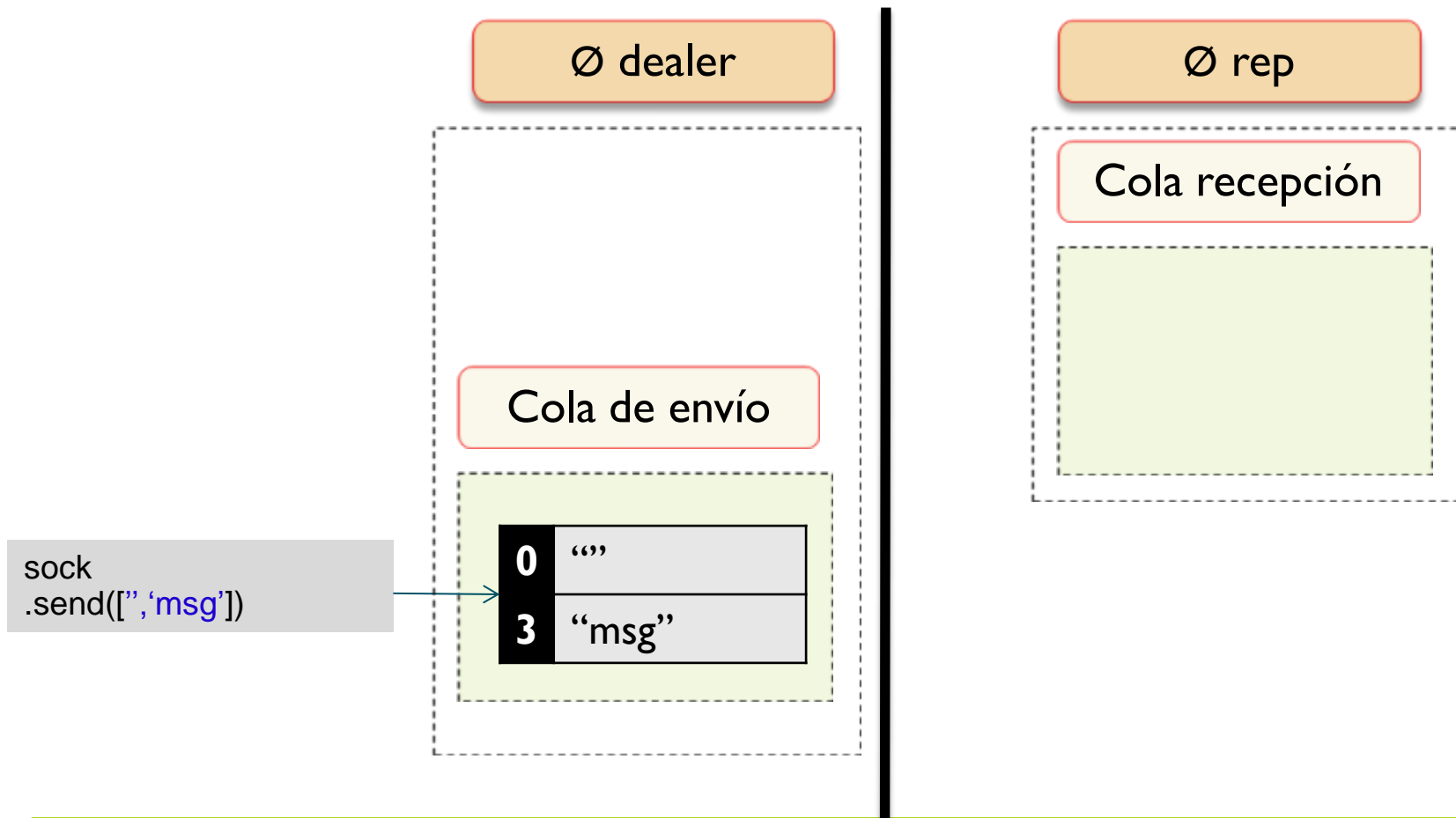
---

- ▶ Es un socket asincrónico de propósito general
- ▶ Puede conectar con req/rep/router/dealer/pull/push/pub/sub
  - ▶ Pub/sub no tiene excesivo sentido, pues es un subprotocolo con filtrado
- ▶ Usado frecuentemente como socket **req** asincrónico
  - ▶ No se bloquea por fallos en los agentes
  - ▶ PERO, debe construir un mensaje de petición adecuado
    - ▶ Con segmento vacío (delimitador) antes del cuerpo real del mensaje
    - ▶ Puede ubicar tras el delimitador cualquier número de segmentos
- ▶ Puede utilizarse también como socket **rep** asincrónico



## 4.1. Sockets dealer: gestión de peticiones y respuestas

- El delimitador debe ser añadido (como cabecera) para comunicarse con un **rep**:

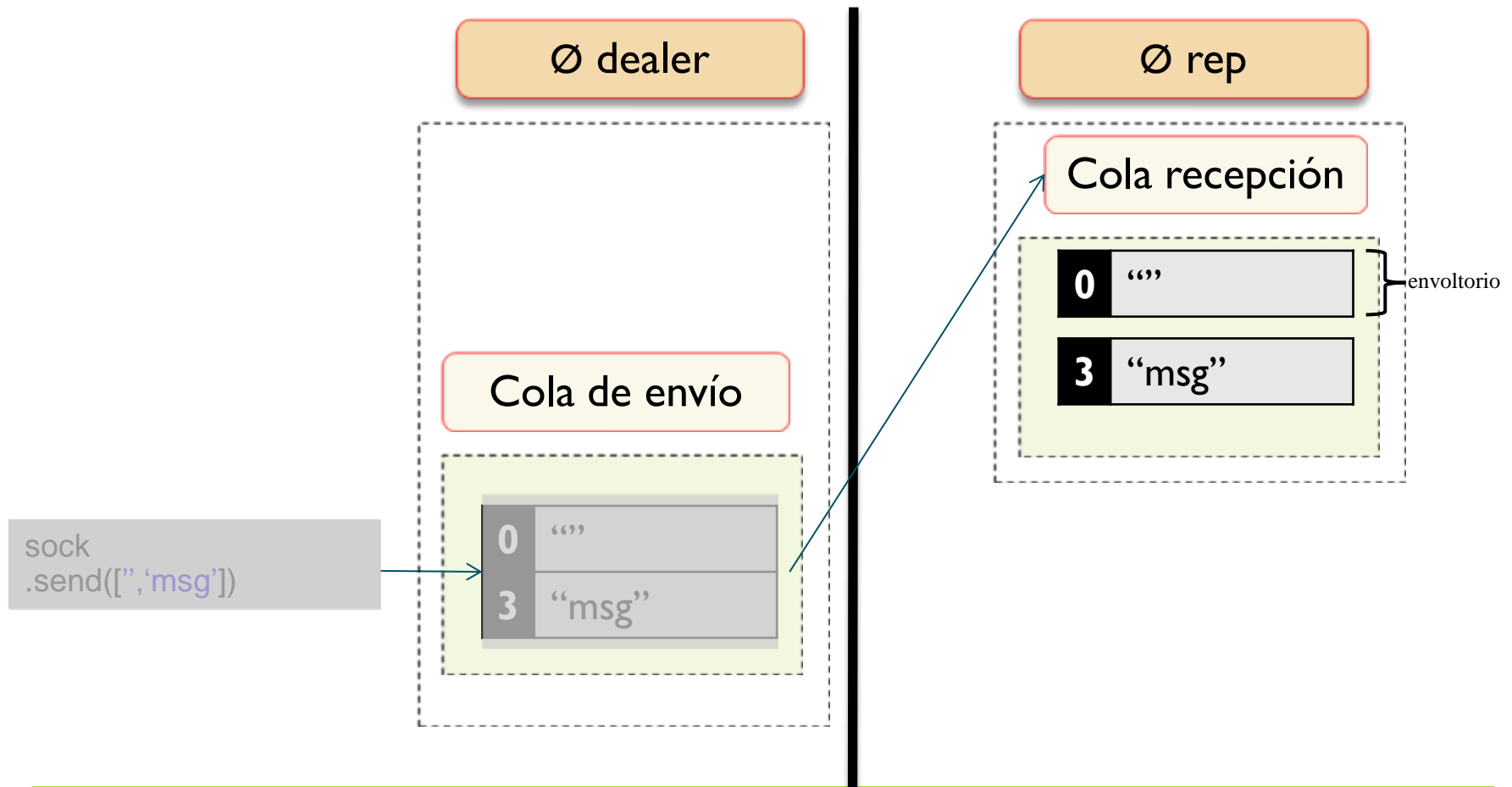






## 4.1. Sockets dealer: gestión de peticiones y respuestas

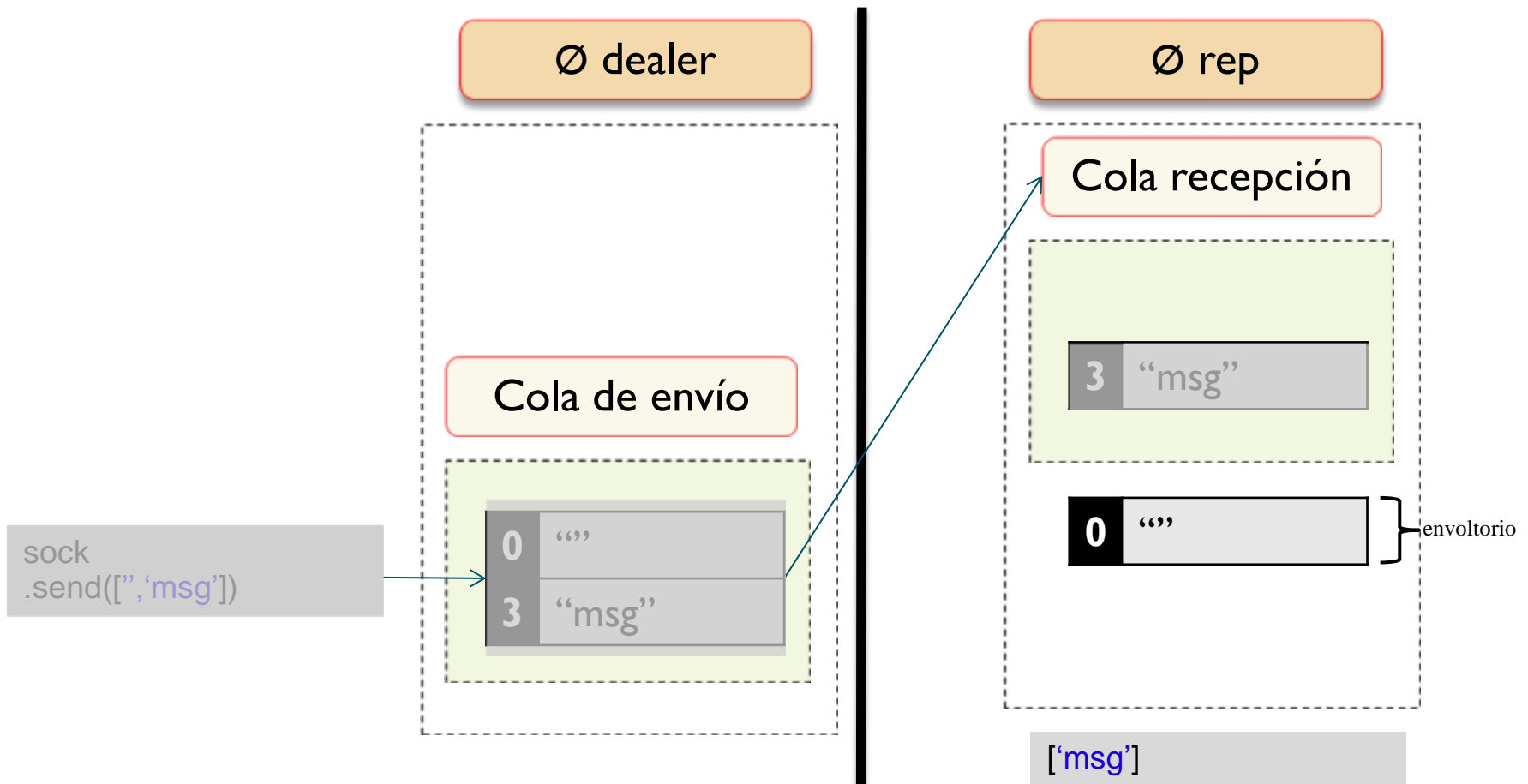
- ▶ Cuando se recibe, el socket **rep** quita el “envoltorio”





## 4.1. Sockets dealer: gestión de peticiones y respuestas

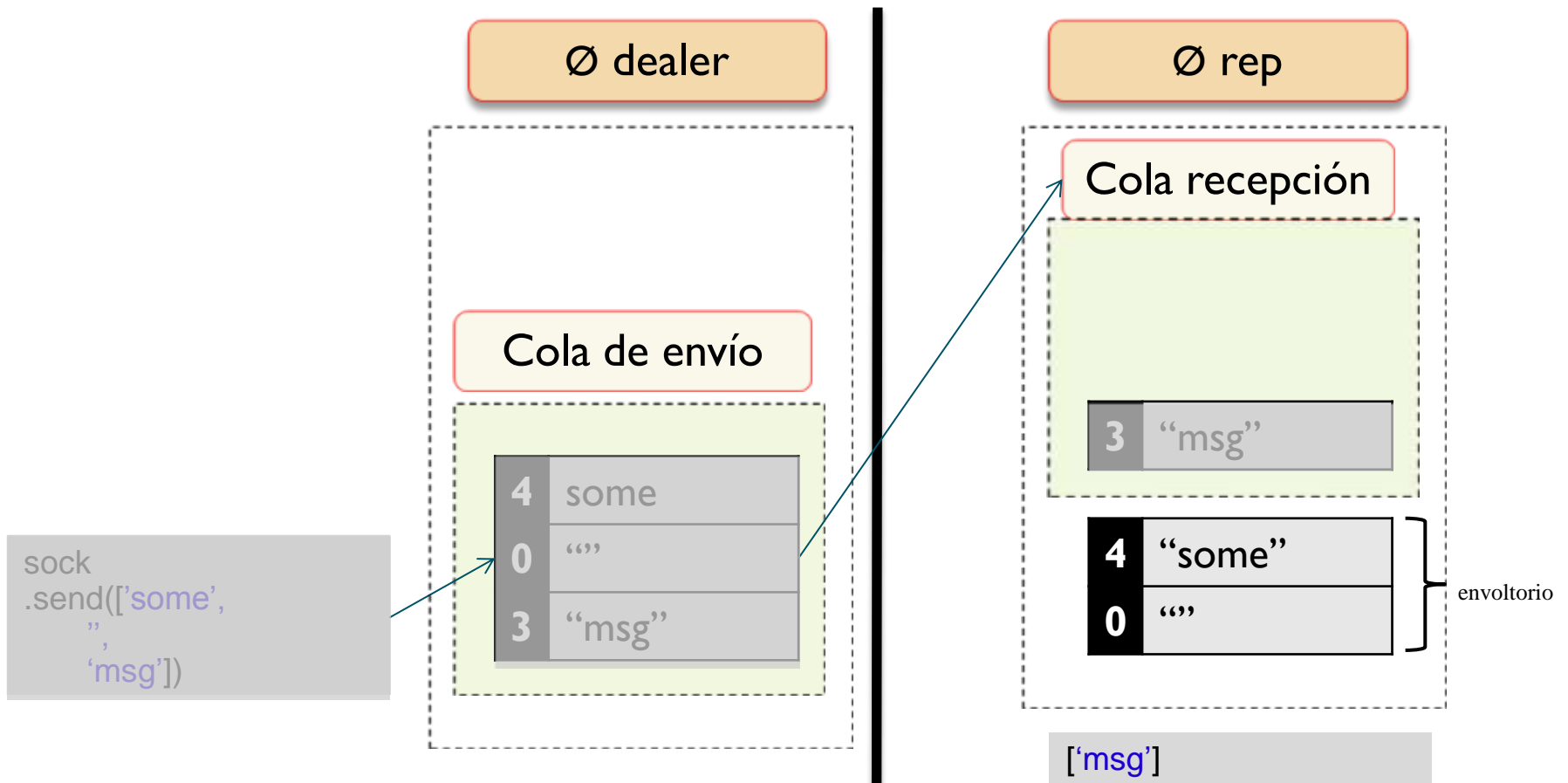
- ▶ Cuando se reciba, el socket **rep** quita el “envoltorio”
  - ▶ La aplicación solo recibe el resto del mensaje





## 4.1. Sockets dealer: envoltorio

- ▶ El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - ▶ El envoltorio es guardado por el **rep...**





## 4.1. Sockets dealer: envoltorio

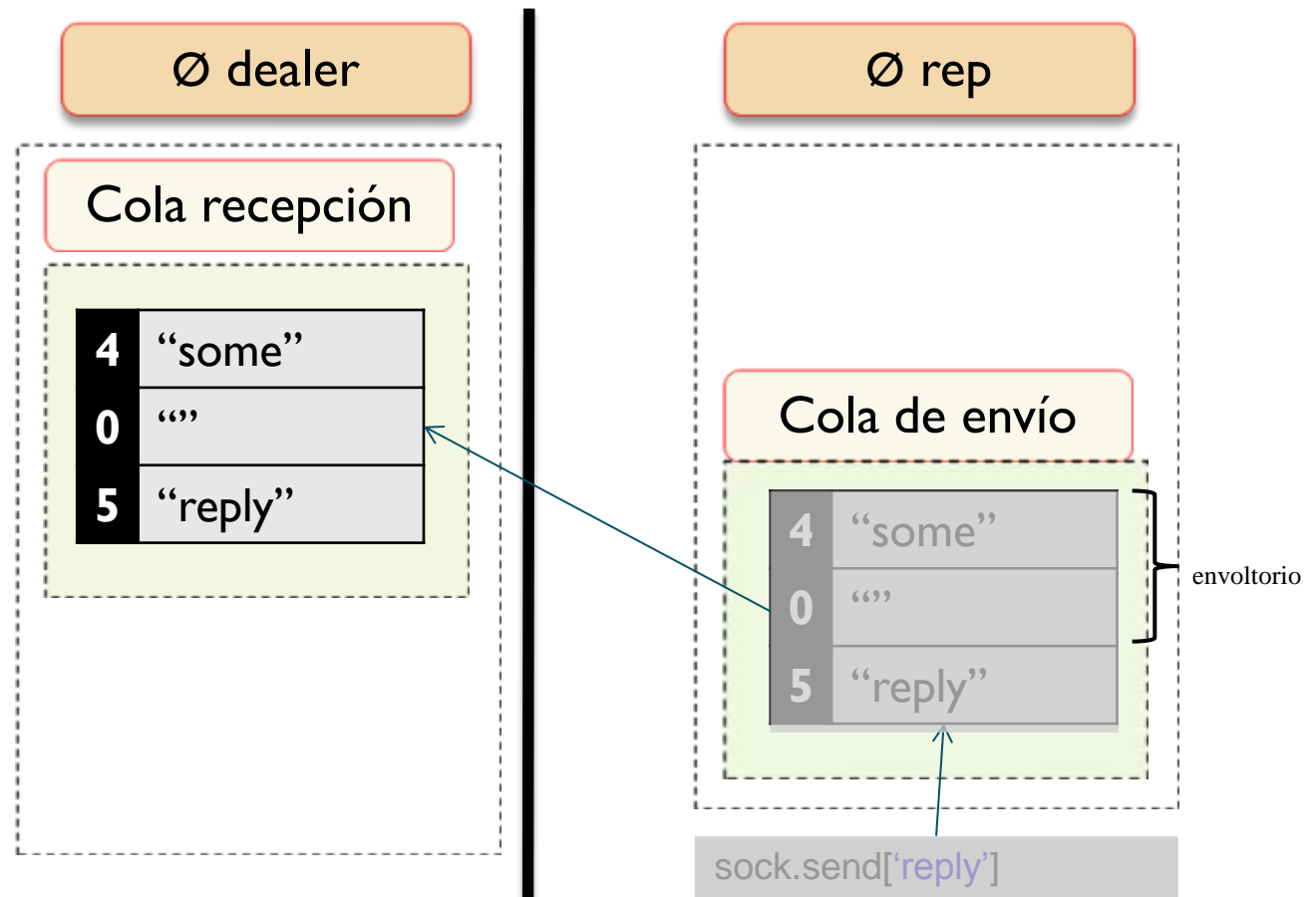
- ▶ El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - ▶ El envoltorio es guardado por el **rep...** y reinsertados en la respuesta





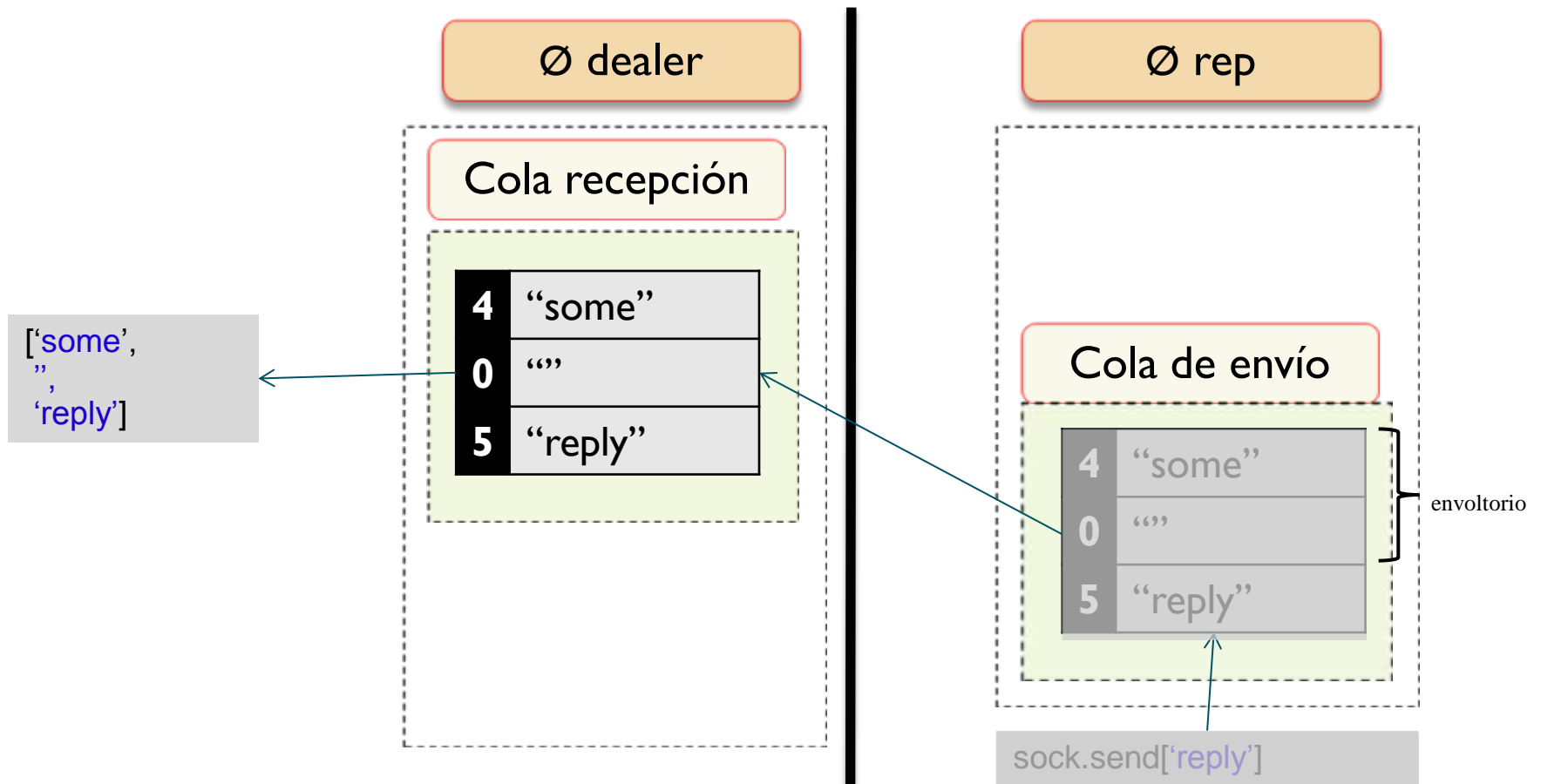
## 4.1. Sockets dealer: envoltorio

- ▶ El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - ▶ El mensaje generado se envía como respuesta



## 4.1. Sockets dealer: envoltorio

- ▶ El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - ▶ Y la aplicación dealer obtiene todo esto





## 4.1. Sockets dealer: ejemplo de código

```
var zmq = require('zmq');
var dealer = zmq.socket('dealer');
var msg = ["", "Hello ", 0];
var host = "tcp://localhost:888";

dealer.connect(host + 8);
dealer.connect(host + 9);

setInterval(function() {
  dealer.send(msg);
  msg[2]++;
}, 1000);

dealer.on('message',
  function(h, seg1, seg2) {
    console.log('Response:' + seg1 + seg2);
  });
```

```
var zmq = require('zmq');
var rep = zmq.socket('rep');

rep.bindSync('tcp://*:8888');
rep.on('message',
  function (intro, count) {
    rep.send(['World ',
              count]);
  });
```

```
var zmq = require('zmq');
var rep = zmq.socket('rep');

rep.bindSync('tcp://*:8889');
rep.on('message',
  function (intro, count) {
    rep.send(['World ',
              count]);
  });
```



## 4.2. Sockets router

- ▶ Sockets bidireccionales asincrónicos
- ▶ Permite enviar mensajes a agentes específicos
  - ▶ Asigna una identidad a cada agente con el que se conecte
  - ▶ La identidad es aquella dada al agente en su programa
    - ▶ `sock.identity = 'my name';`
    - ▶ Cuando el agente no tenga una identidad asociada
      - El socket router crea una identidad aleatoria para ese agente conectado
      - La identidad creada se mantiene mientras la conexión dure
      - Al cerrar la conexión y restablecerla, la identidad cambia
    - ▶ Los identificadores son cadenas arbitrarias de hasta 256 octetos
- ▶ Cuando el socket router pase un mensaje a la aplicación
  - ▶ Añade un segmento inicial con el identificador del agente emisor





## 4.2. Sockets router: ejemplo con agente req

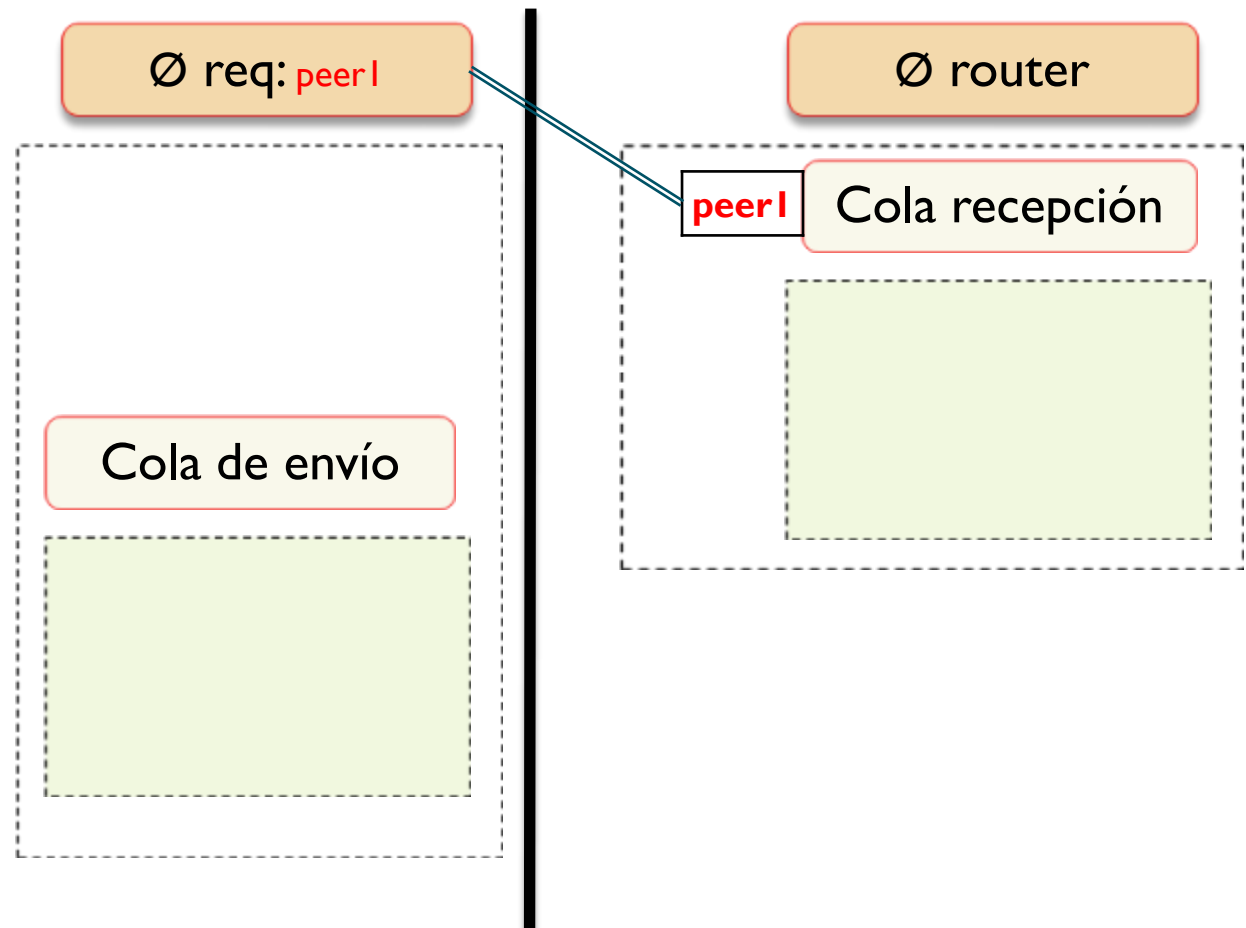
- ▶ El agente req tiene identidad “peer I”





## 4.2. Sockets router: ejemplo con agente req

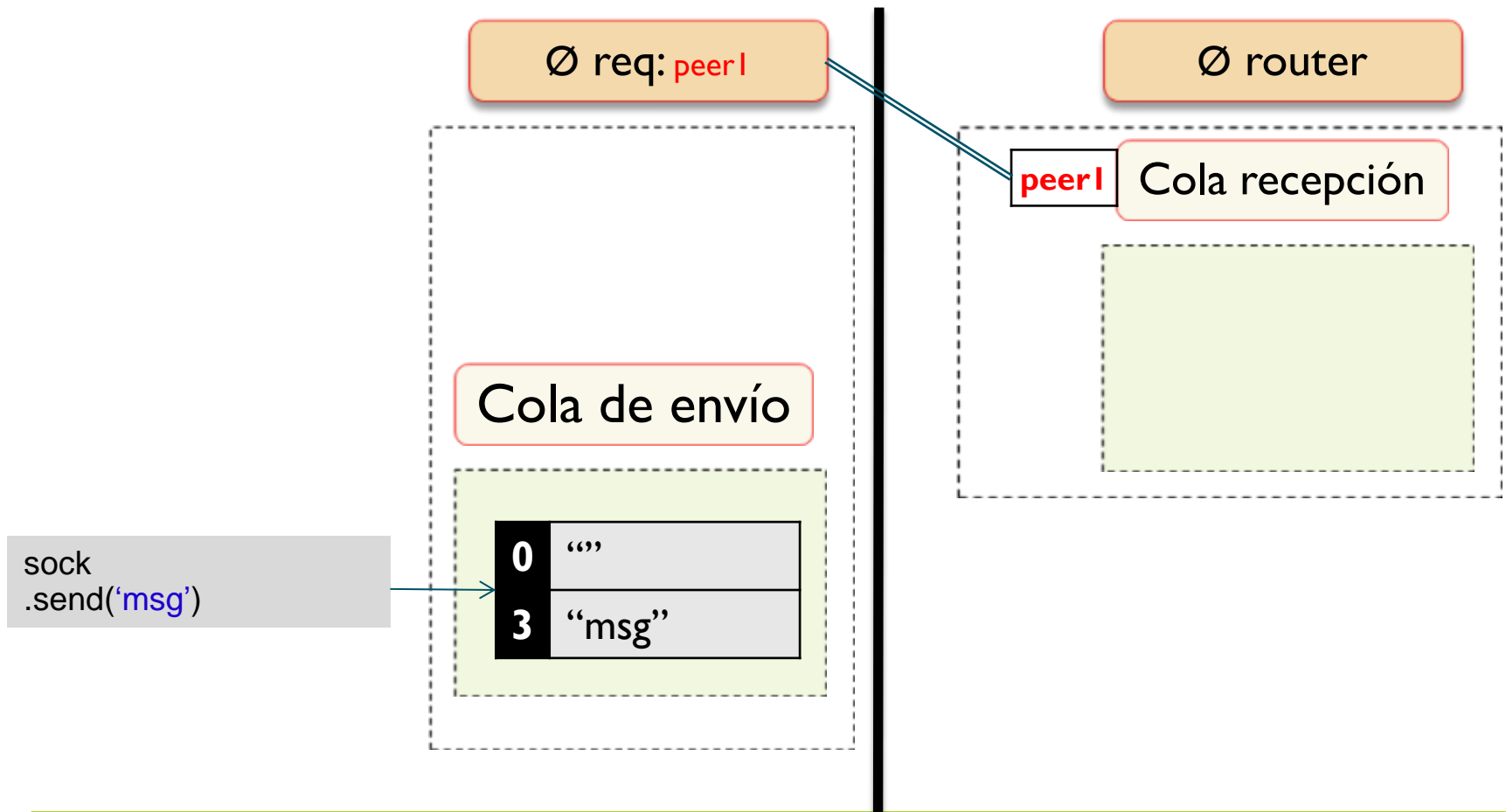
- ▶ El agente req conecta con el router
  - ▶ El router obtiene su identidad, lo almacena y le asocia colas de envío y recepción





## 4.2. Sockets router: ejemplo con agente req

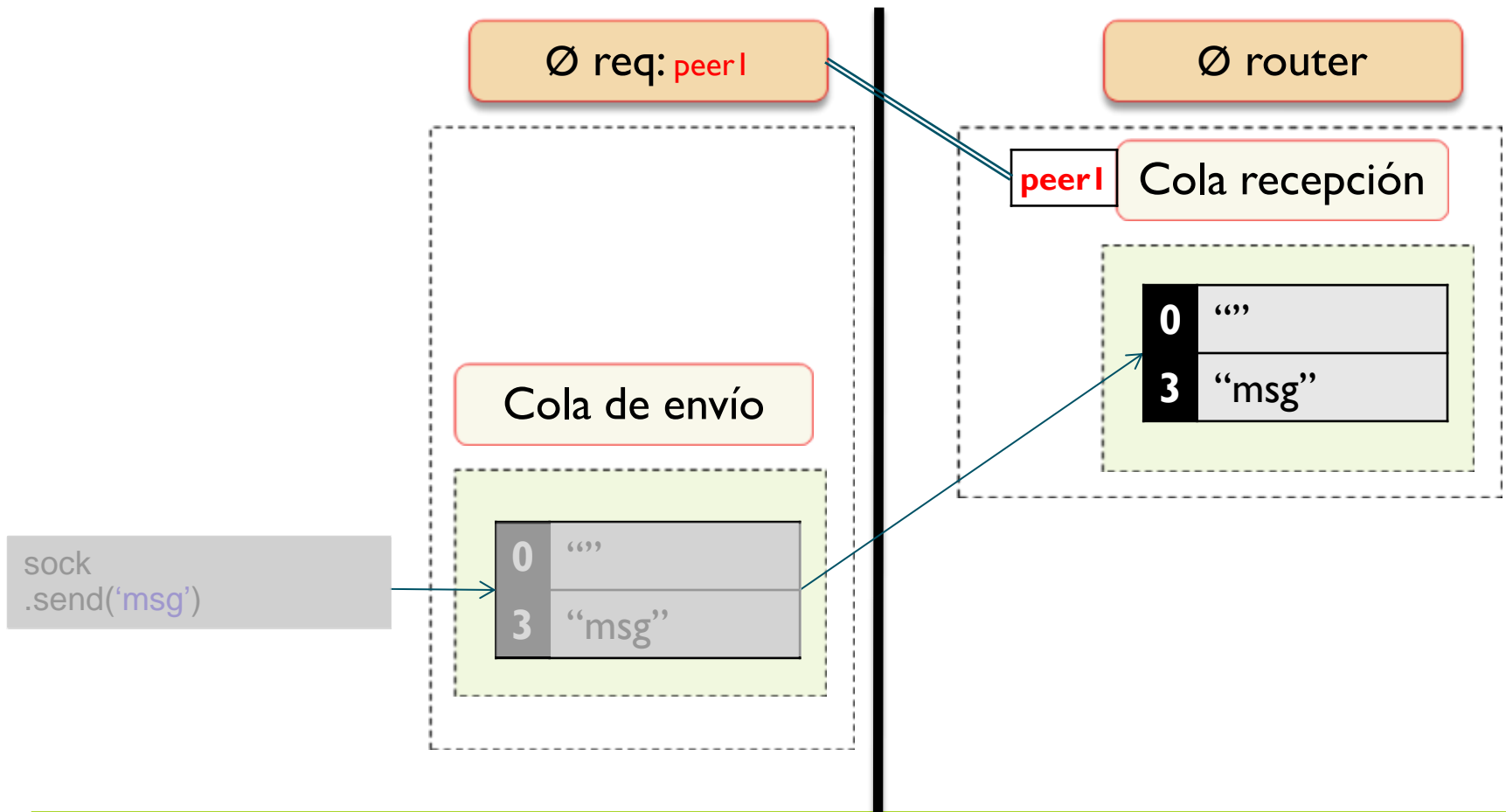
- ▶ La aplicación req envía un mensaje
  - ▶ El socket req añade el delimitador...





## 4.2. Sockets router: ejemplo con agente req

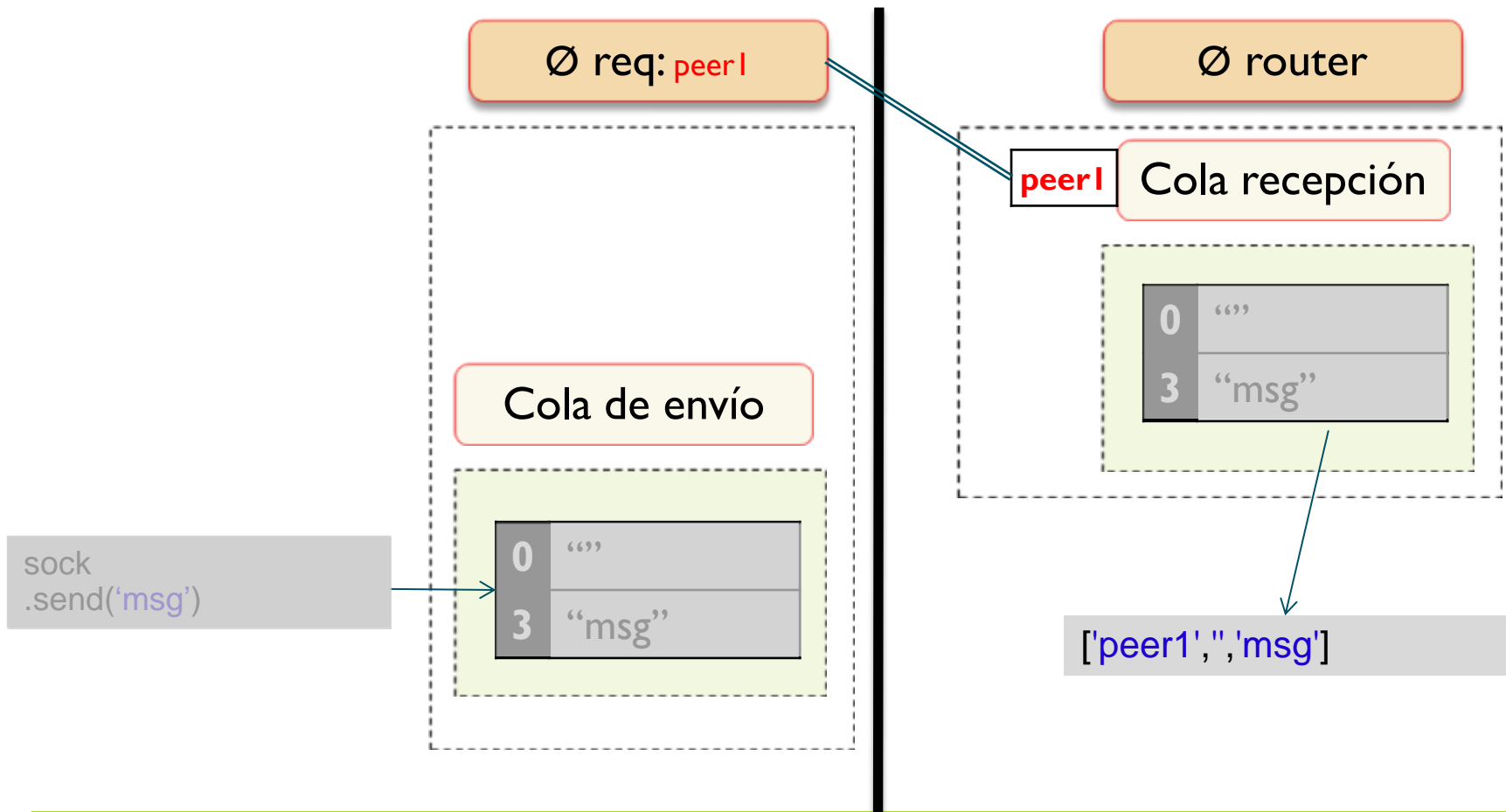
- ▶ La aplicación req envía un mensaje
  - ▶ El socket req añade el delimitador... y lo envía





## 4.2. Sockets router: ejemplo con agente req

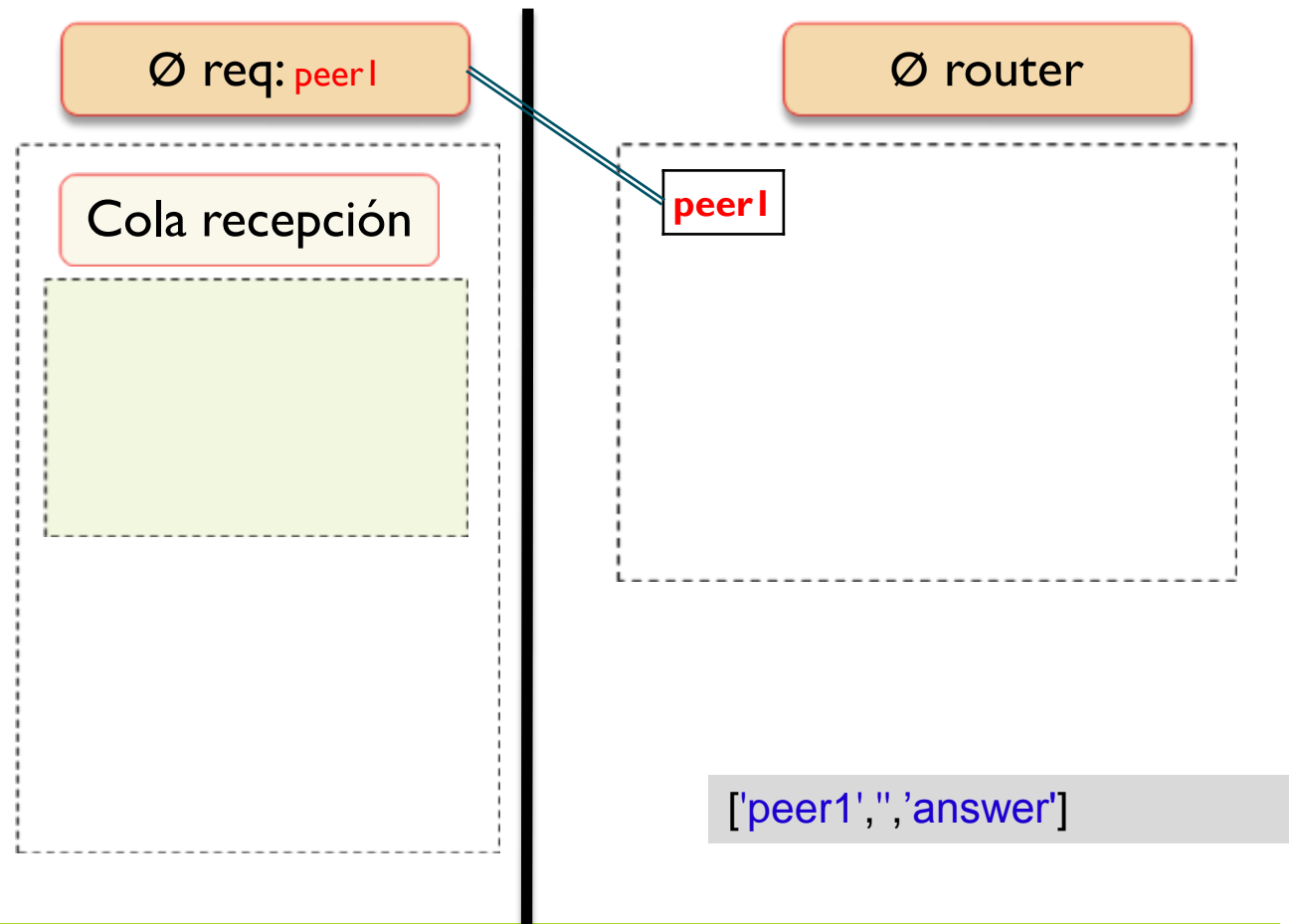
- ▶ El socket router entrega el mensaje a su aplicación
  - ▶ Con la identidad del emisor encabezando un segmento





## 4.2. Sockets router: ejemplo con agente req

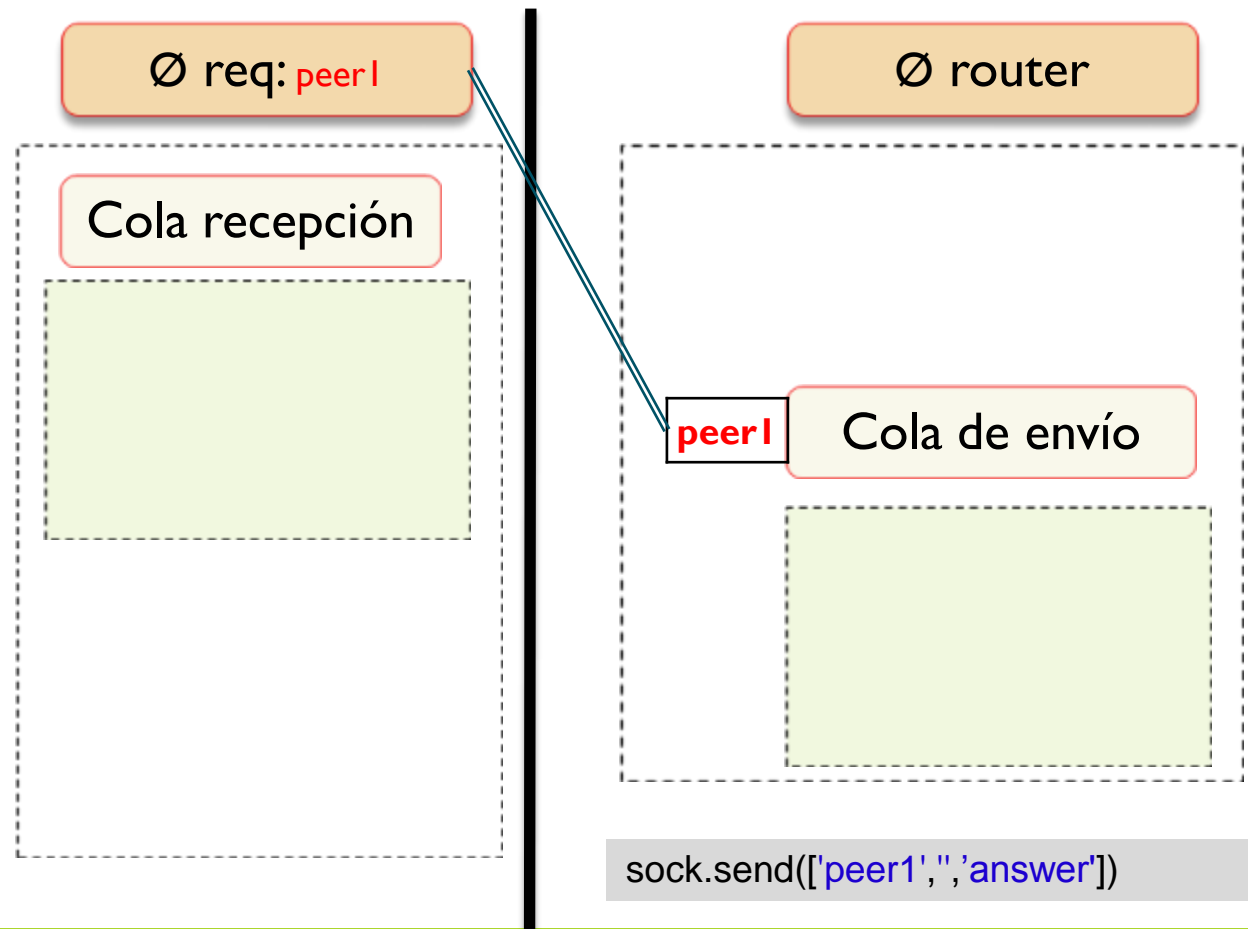
- ▶ La aplicación del router crea una respuesta, construyendo el mensaje de respuesta
  - ▶ El primer segmento contiene la identidad del agente que recibirá la contestación





## 4.2. Sockets router: ejemplo con agente req

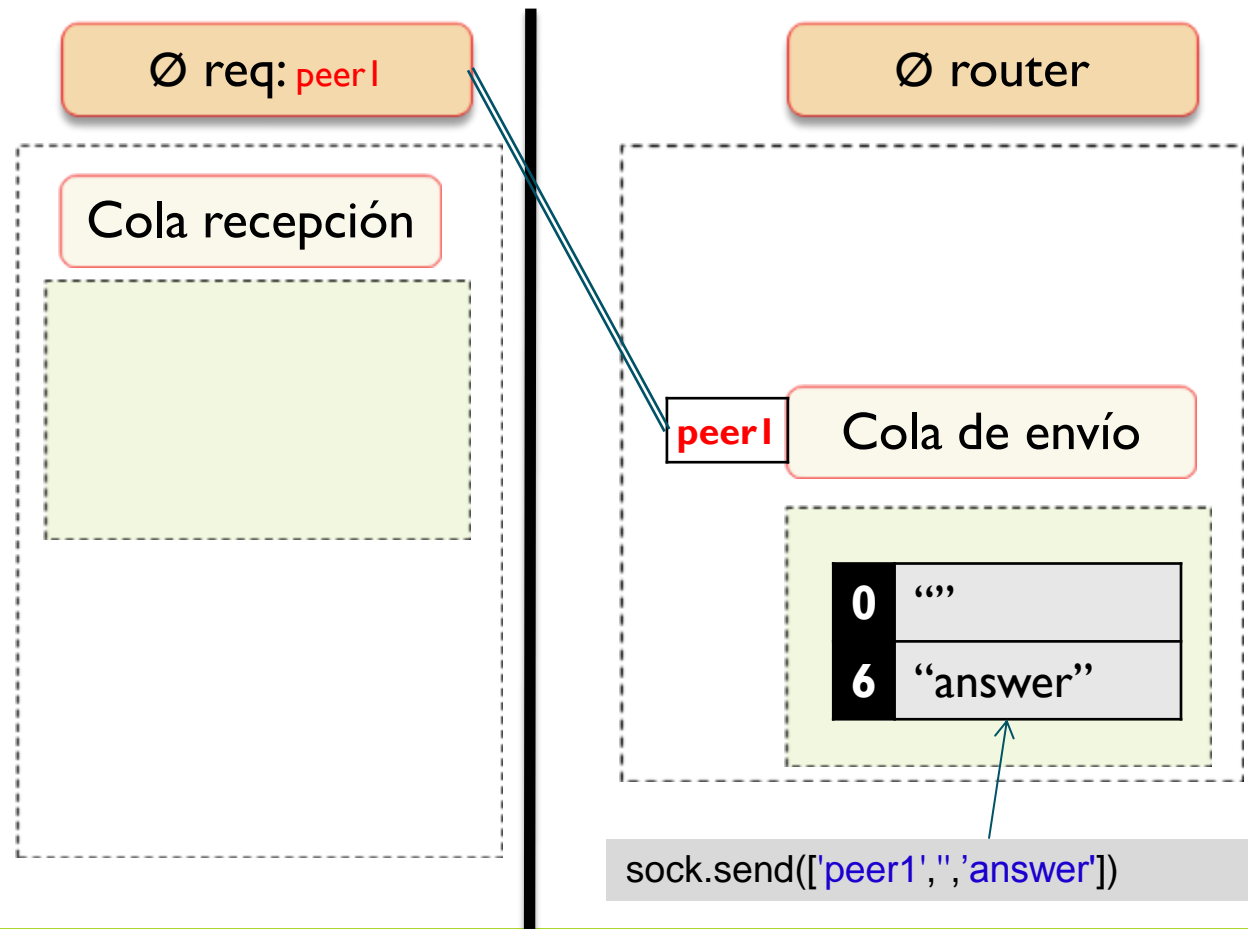
- ▶ La aplicación del router envía el mensaje
  - ▶ El socket router selecciona la cola de envío basándose en la identidad





## 4.2. Sockets router: ejemplo con agente req

- ▶ El socket router quita el segmento con la identidad
  - ▶ Deja el resto del mensaje para enviar...

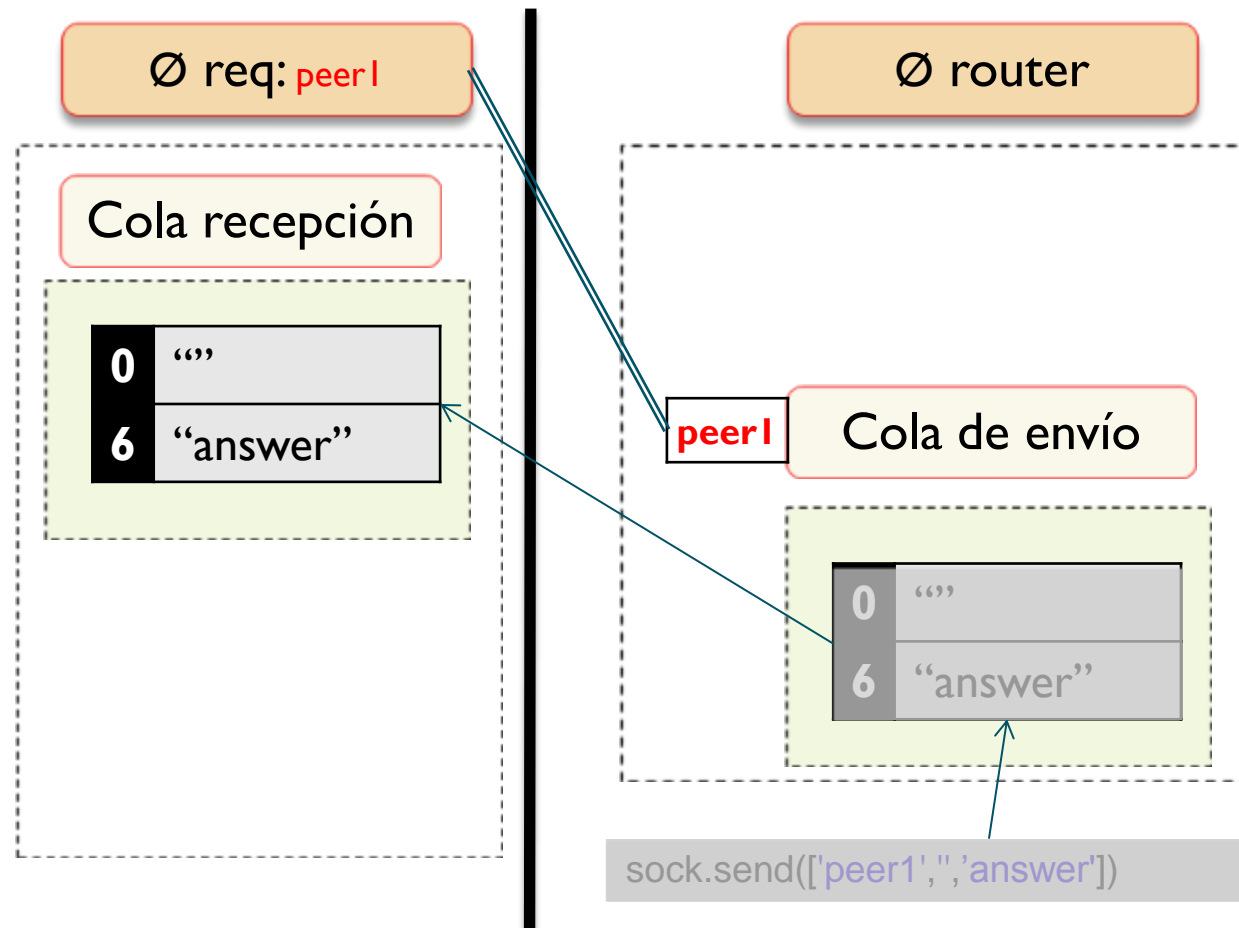






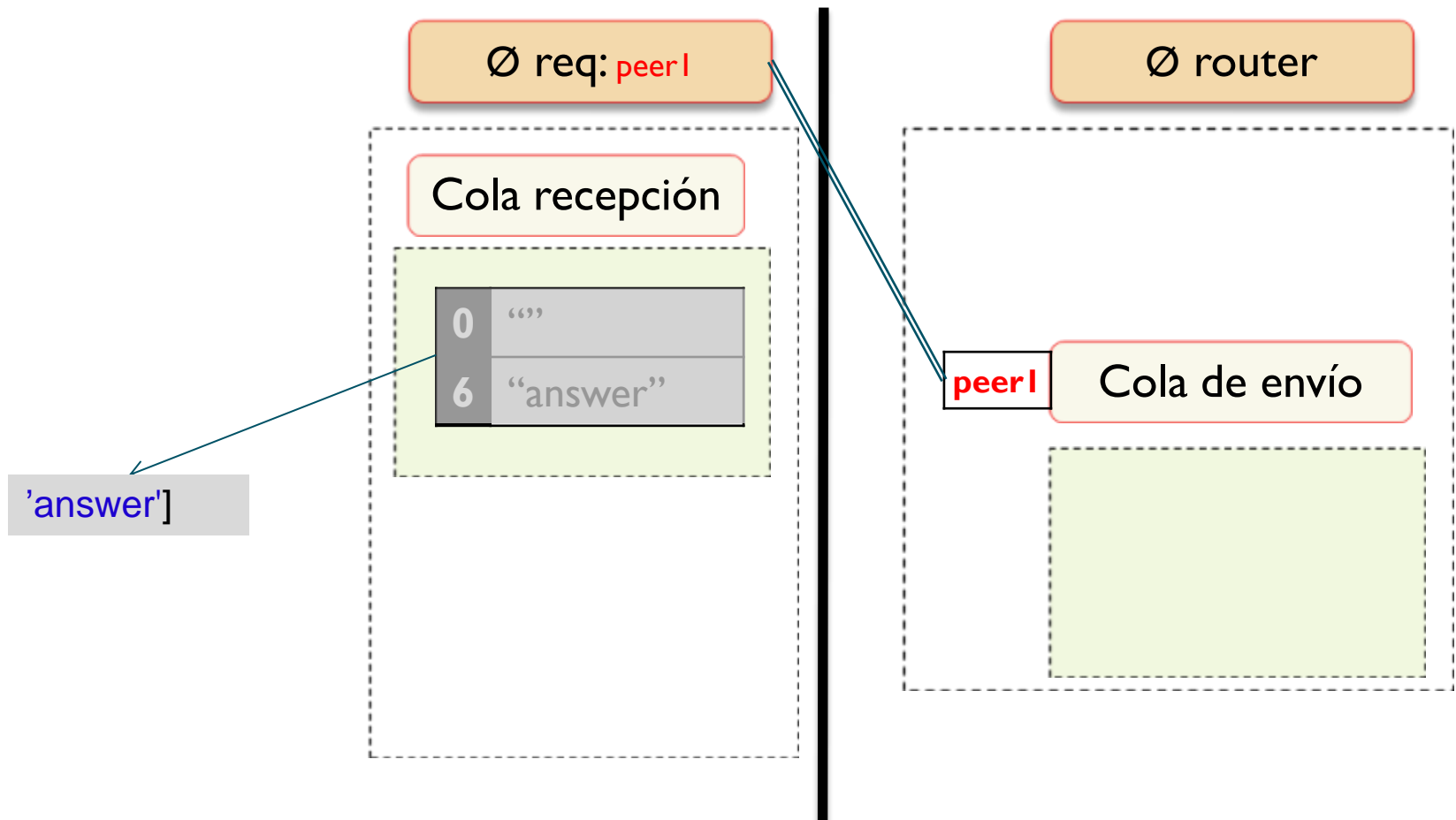
## 4.2. Sockets router: ejemplo con agente req

- ▶ El socket router quita el segmento con la identidad
  - ▶ Deja el resto del mensaje para enviar... y lo envía



## 4.2. Sockets router: ejemplo con agente req

- ▶ El socket req lo entrega a su aplicación
  - ▶ Eliminando el segmento delimitador





# Índice

---

1. Introducción
2. Mensajes
3. 0MQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía



## 5. Resultados de aprendizaje

---

- ▶ Al finalizar este seminario, el alumno debería ser capaz de:
  - ▶ Entender cómo 0MQ opera internamente
  - ▶ Escribir aplicaciones usando el mapeo de 0MQ para Node.js con tipos básicos de socket
  - ▶ Ser capaz de usar los tipos avanzados de socket para implantar patrones adicionales de conexión



# Índice

---

1. Introducción
2. Mensajes
3. 0MQ API
4. Tipos de “socket” avanzados
5. Resultados de aprendizaje
6. Bibliografía



## 6. Bibliografía

---

- ▶ <http://zguide.zeromq.org/page:all>
  - ▶ Permite lectura on-line
  - ▶ Existe una versión en PDF
  - ▶ El sitio web mantiene información adicional