Tema 3 - Middleware

Tecnologías de los Sistemas de Información en la Red

Índice

1. Introducción

- Los componentes de un sistema distribuido de gran escala son:
  - Especificados y desarrollados de manera independiente
    - Sin ningún comité que revise todos los casos de uso posibles.
  - Desplegados autónomamente
    - Cada uno diseñado como un agente
    - Pero estableciendo dependencias entre ellos
      - *Consumiendo* o *produciendo* funcionalidad de/para otros agentes
- Necesitan interactuar de manera sencilla y útil
  - P.ej., un servicio de planificación de rutas puede depender de un servicio GIS que facilite información básica sobre distancias.
  - P.ej., un sistema de autorización de entrada puede necesitar un servicio remoto de reconocimiento biométrico.

1. Introducción

- Incluso los sistemas fuertemente unidos (por tener pocos componentes estrechamente relacionados)...
  - Suelen estar desarrollados por más de un programador
  - Constan de múltiples componentes interdependientes
- En cualquier caso:
  - Necesitan resolver las complicaciones de un entorno distribuido
  - El resultado debe estar tan libre de errores como sea posible
    - La depuración es mucho más compleja en un sistema distribuido
  - El desarrollo suele estar sujeto a plazos estrictos
    - Debe invertirse el menor tiempo posible en la codificación

1. Introducción

- ¿Cómo asegurar que no se necesita ser un genio para...
  - ...escribir los "millones" de líneas de código necesarias?
  - ...desplegar y gestionar los sistemas resultantes?
- Un problema es la complejidad de los detalles
  - Resolver estas complejidades es una fuente de errores
  - Preocuparse por demasiados detalles a la vez es una receta para alcanzar el desastre
    - A lo que debe añadirse las largas sesiones de depuración
      - Frecuentemente tras el despliegue, en producción
- Otro problema: muchas tareas son repetitivas
  - Podríamos ahorrar recursos reutilizando soluciones previas

1. Introducción

- Fuentes de complejidad
- Principalmente para solicitar servicios
  - Por tanto, relacionadas con la comunicación
  - Encontrar servidores que proporcionen el servicio
    - ¿Cómo puede un cliente contactar con el servicio sin conocer el ordenador concreto de cada uno de sus servidores?
    - ¿Cómo se identifican y localizan los servidores?
    - ¿Cómo se identifican los clientes en una petición?
      - Para que el servidor pueda responder
  - Especificación de la funcionalidad de los servicios
    - ¿Cuál es el API de un servicio?
    - ¿Cómo averiguar si su versión ha cambiado?

1. Introducción

- Al dar formato a la información transmitida...
  - ¿Cómo debe el programador construir e interpretar las peticiones de servicio?
    - P.ej., ¿qué significa el octavo byte de la petición?
  - Compatibilidad entre los entornos de programación utilizados en cada extremo
    - Relacionado con la **representación interna de la información**
    - P.ej., cliente escrito en Java, servidor escrito en C

- Sincronización entre cliente y servidor
  - Ordenación de acciones: ¿Cómo sabrá un cliente que el servicio ha aceptado sus peticiones?
  - Recepción de resultados:¿Cómo devuelve un servidor los resultados? ¿Cómo los obtiene el cliente?

1. Introducción

- Seguridad
  - ¿Cómo puede el programador de servicios saber qué peticiones deben ser aceptadas?
  - ¿Cómo pueden garantizarse las propiedades relacionadas con la seguridad?

- Gestión de fallos
  - ¿Cómo sabrá un agente cuándo iniciar acciones de recuperación ante un fallo?
    - P.ej., ¿cómo averiguar cuándo un servidor ha caído?
    - P.ej., ¿cómo averiguar si la información transmitida se ha perdido?
    - …

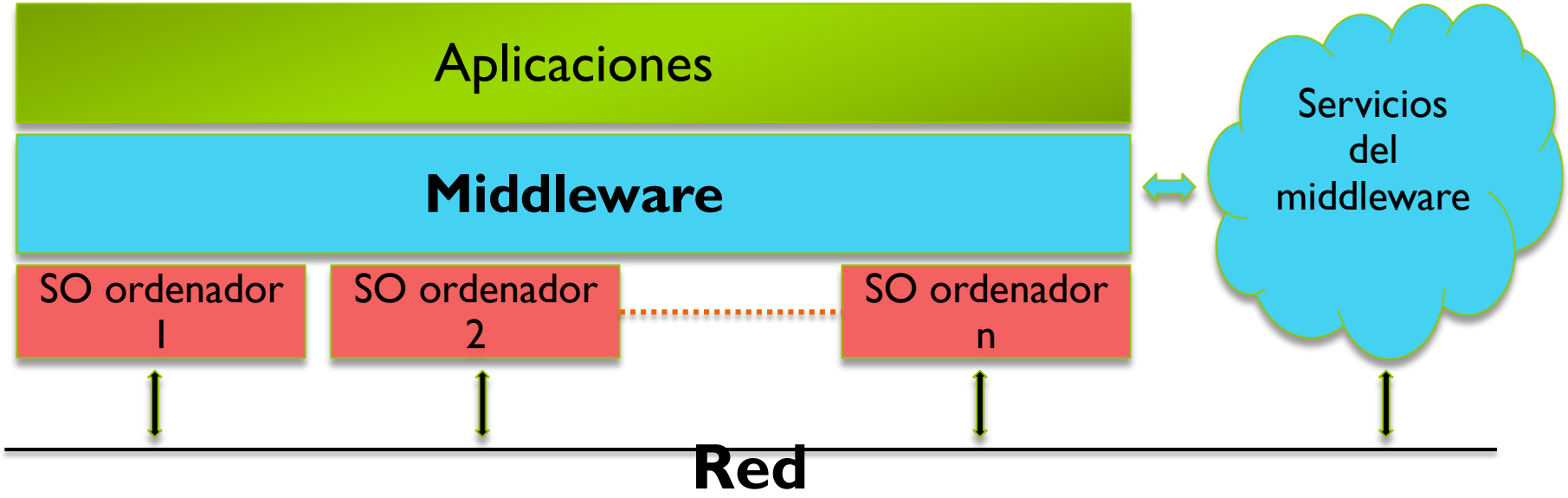
1. Introducción

- Técnicas para superar esta complejidad
  - Estándares (de facto y de iure)
    - Introducen formas racionales de hacer las cosas
      - La experiencia es un grado
    - Ayudan a familiarizar a los programadores con las técnicas a utilizar
      - No tienen por qué aprender cosas nuevas cada vez
    - Facilitan la interoperabilidad
      - Mejor elección para los integradores y administradores de sistemas
    - Proporcionan funcionalidad de alto nivel
      - Menos código que escribir
      - Menor complejidad para gestionarlo
  - Middleware

Índice

2. Middleware

- Nivel (o niveles) de software y servicios entre las aplicaciones y el nivel de comunicaciones (sistema operativo)
- Introduce múltiples "transparencias"
- **Transparencia**: reducción de la complejidad, ocultando y manejando los detalles de manera uniforme



2. Middleware: Características aconsejables

- Perspectiva del programador
  - Implantación sencilla
    - Conceptos claros y bien definidos
    - Poca complejidad en los elementos manejados
      - Evita errores al programar
  - Resultado fiable
    - Proporciona una manera de hacer las cosas estandarizada, extendida, comprensible y bien definida.
  - Mantenimiento sencillo
    - Los cambios en sus API deben tener un bajo impacto en la necesidad de revisar los programas en desarrollo
- Perspectiva del administrador
  - Fácil instalación, configuración y actualización
  - Interoperabilidad: Facilidad para interactuar con productos de terceras partes

Índice

3. Sistemas de mensajería

- Asincrónicos por naturaleza
  - Desacoplamiento entre emisor y receptor
- Se envían piezas concretas de información
  - Mensaje: transmisión atómica (todo o nada)
  - Tamaño arbitrario
  - Soporte para estructurar el mensaje
  - Gestión de colas
    - Con ciertas garantías de orden
- No se impone una visión de estado compartido
  - Mejor escalabilidad potencial
  - Mayor facilidad para evitar problemas de concurrencia
  - Encaja perfectamente en el modelo de sistema distribuido ya presentado

3. Sistemas de mensajería: ejemplos

- Estándar establecido: AMQP
  - Ejemplo: RabbitMQ
  - Apache ActiveMQ
- STOMP
- ØMQ

3. Sistemas de mensajería: técnicas

- Dos clases principales
  - Sistemas no persistentes (*transient / stateless*)
    - Exige que el receptor esté activo para transmitir el mensaje
  - Sistemas persistentes
    - Los mensajes se mantienen en buffers: El receptor no tiene por qué existir cuando se envíe el mensaje
- Entre los sistemas persistentes: con/sin gestor (siguiente diapositiva)

3. Sistemas de mensajería: técnicas

- Entre los sistemas persistentes
  - Basados en gestor (*Broker-based*)
    - Servidores concretos guardan los mensajes y proporcionan garantías fuertes
    - Sobrecarga derivada de la necesidad de mantener los mensajes en disco
    - Ejemplo: AMQP
  - Sin gestor (*Brokerless*)
    - Emisores y receptores mantienen los mensajes
      - Normalmente en memoria principal
    - Garantías de persistencia más débiles
      - Se asegura: desacople entre emisor y receptor, predisposición del receptor para intervenir...
    - Puede tomarse como base para construir sistemas basados en gestor
    - Ejemplo: ØMQ

Índice

4. ZeroMQ

1. Introducción
2. Middleware
3. Sistemas de mensajería
4. ZeroMQ
    1. Introducción
    2. Mensajes
    3. API de ØMQ
    4. Sockets avanzados
5. Otro middleware
6. Conclusiones
7. Referencias

4.1. Introducción: Objetivos de ØMQ

- 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 las URL.

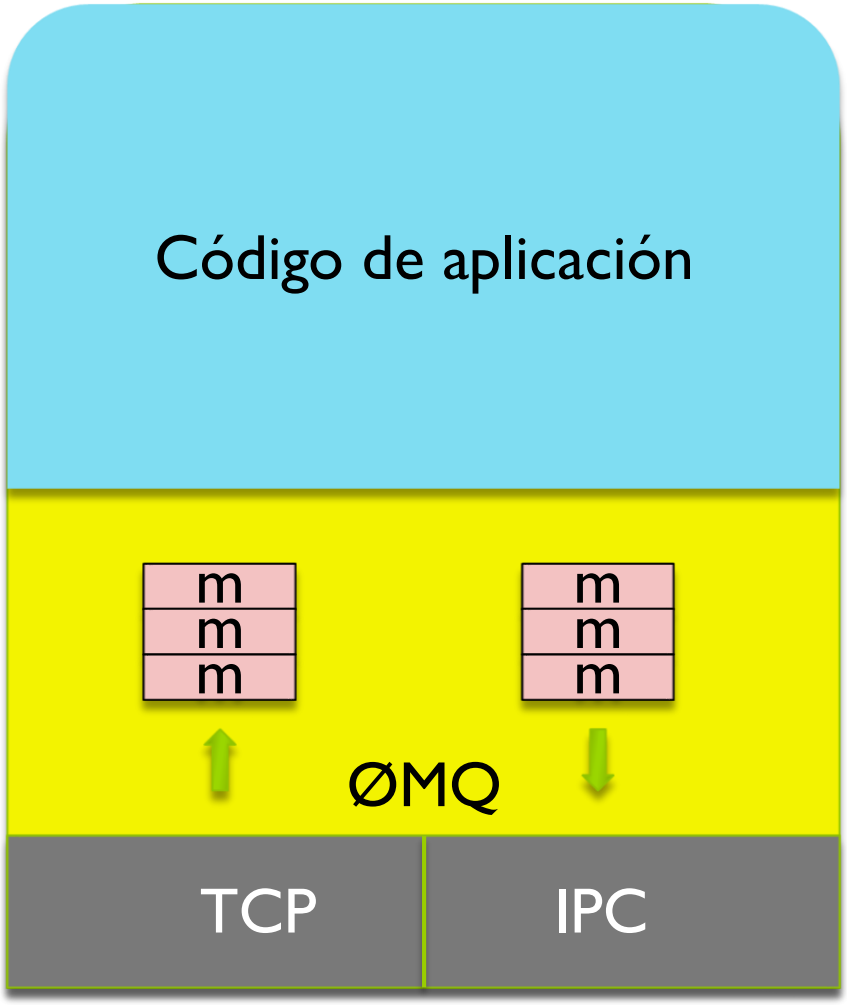
4.1. Introducción: Características principales

- Comunicación basada en mensajes
  - Persistencia débil: colas en memoria principal
- Es sólo una biblioteca
  - No se necesita arrancar ningún servidor específico (*broker*)
  - 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

4.1. 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
    - IPC (Sockets Unix)
- Transporte utilizado para instanciar un socket
  - Fácilmente modificable mediante un cambio en la configuración

4.1. Introducción: Vista de un proceso ØMQ



- La aplicación enlaza con la biblioteca ØMQ
- ØMQ mantiene colas en memoria
  - En el emisor
  - En el receptor
- ØMQ usa niveles de comunicación

4.1. Introducción: Instalación de ØMQ

- **ØMQ es una biblioteca**
  - Debe instalarse antes de utilizarla en los programas
  - Para utilizarla en NodeJS hay que usar un módulo: **zeromq**
    - El módulo debe importarse en cada programa
      - `const zmq = require('zeromq')`
    - Al instalar el módulo, este ya se encarga de instalar la biblioteca

- **La orden necesaria para instalar el módulo es:**
  - **npm install zeromq@5**

4. ZeroMQ

1. Introducción
2. Middleware
3. Sistemas de mensajería
4. ZeroMQ
   1. Introducción
   2. Mensajes
   3. API de ØMQ
   4. Sockets avanzados
5. Otro middleware
6. Conclusiones
7. Referencias

4.2. 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" (multi-segmento)
  - 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, ØMQ 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

4.2. Mensajes

- El contenido de los mensajes resulta transparente para ØMQ
- No se necesita soporte para "marshalling"
  - No hay que preocuparse por la codificación
  - Los mensajes son "blobs" para ØMQ
  - Pero el API de ØMQ 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

4.2. 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 …

4. ZeroMQ

1. Introducción
2. Middleware
3. Sistemas de mensajería
4. ZeroMQ
    1. Introducción
    2. Mensajes
    3. API de ØMQ
    4. Sockets avanzados
5. Otro middleware
6. Conclusiones
7. Referencias

4.3. API ØMQ

1. Sockets
   - Envío y recepción utilizan sockets
   - Varios tipos de sockets
   - Operaciones bind/connect
2. Patrones de comunicación
   - Soportados por tipos específicos de socket

4.3.1 Sockets ØMQ

- La creación de un socket es sencilla:

```
const zmq = require('zeromq')

const zsock = zmq.socket(<TIPO SOCKET>)
```

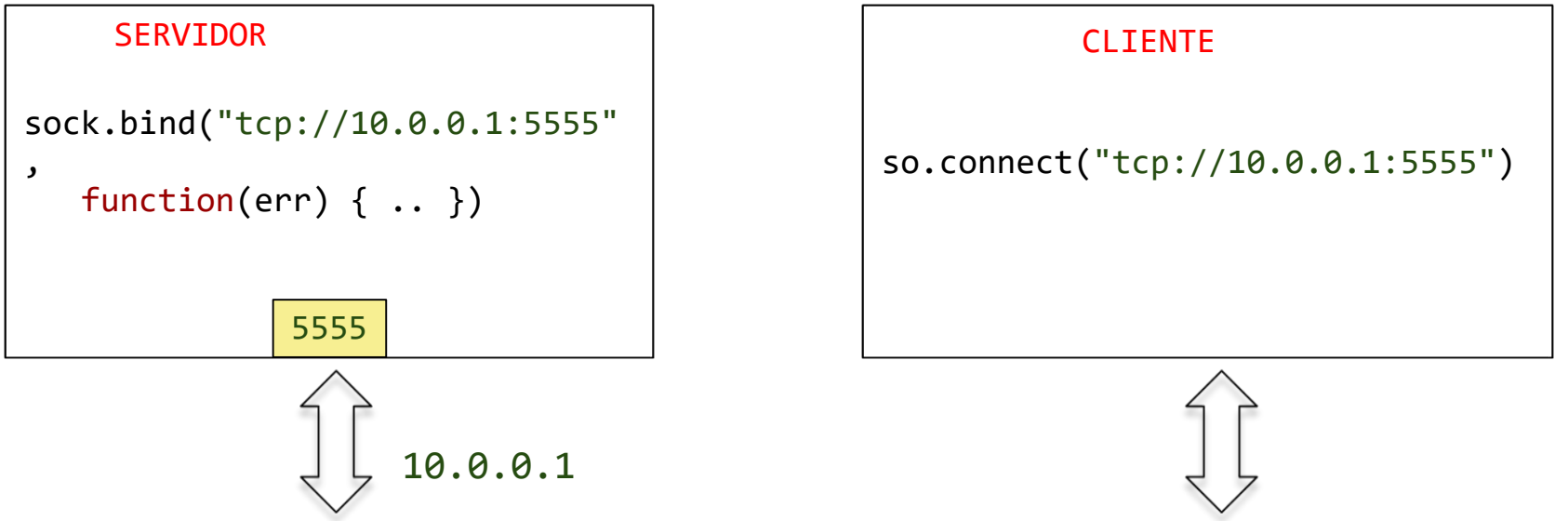
- Donde <TIPO SOCKET> será uno de los siguientes

req	push	pub
rep	pull	sub
dealer	*pair*	*xsub*
router		*xpub*

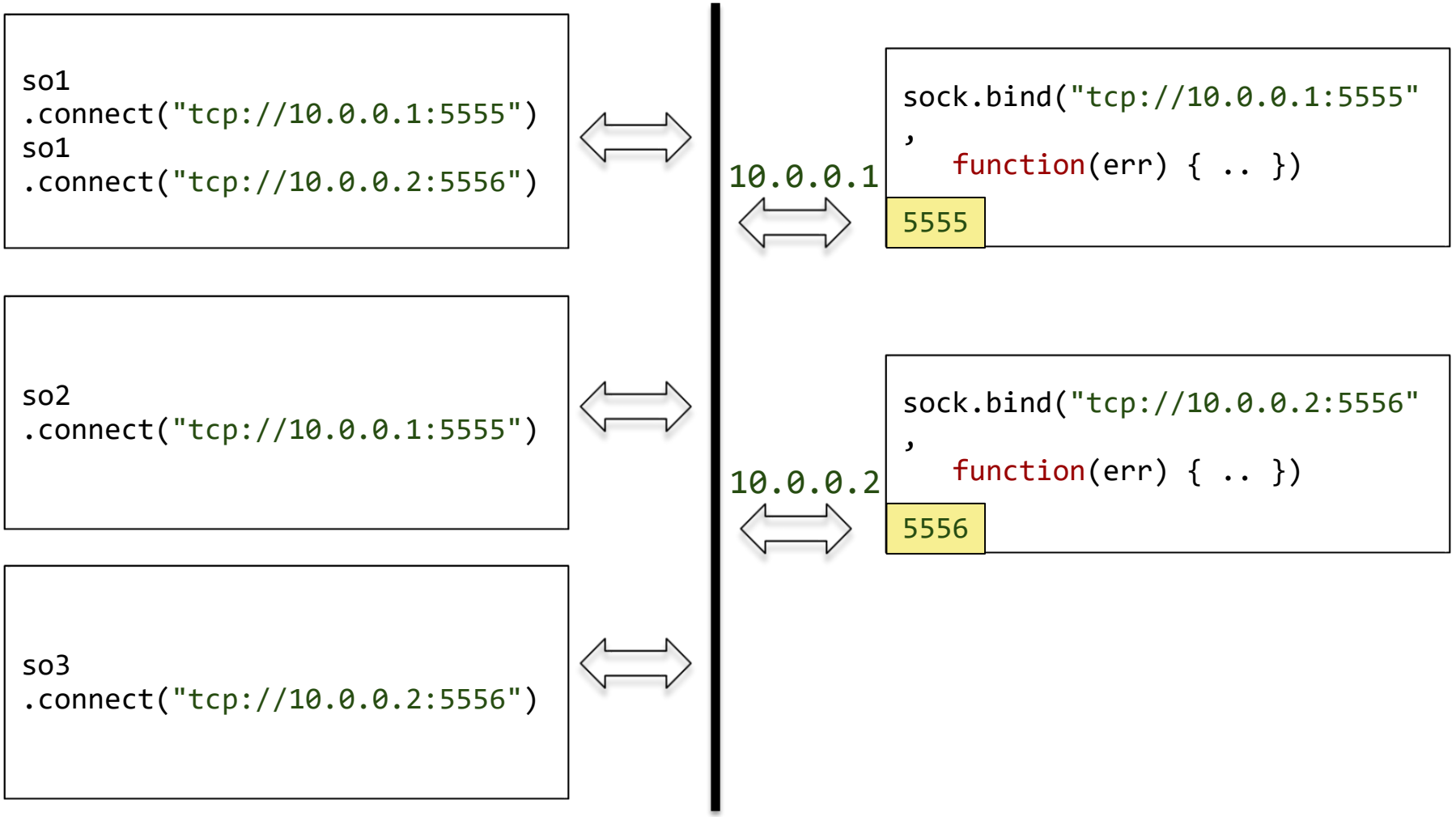
- Qué tipos utilizar dependerá de los patrones de conexión en los que intervenga

4.3.1 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



4.3.1 Sockets: Múltiples conexiones son posibles



4.3.1 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

4.3.1 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"

4.3.1 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"

4.3.1 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>

4.3.1 Sockets: Envío de mensajes

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

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

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

4.3.1 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 (let key in arguments) {
      console.log("Part" + key + ": " + arguments[key])
   }
})
```

- … o convertir antes en vector

```
let segments = Array.from(arguments)
segments.forEach(function(seg) { … })
```

4.3.1 Sockets: Opciones

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

```
sock.identity = 'frontend'
sock.subscribe('SOCCER')
```

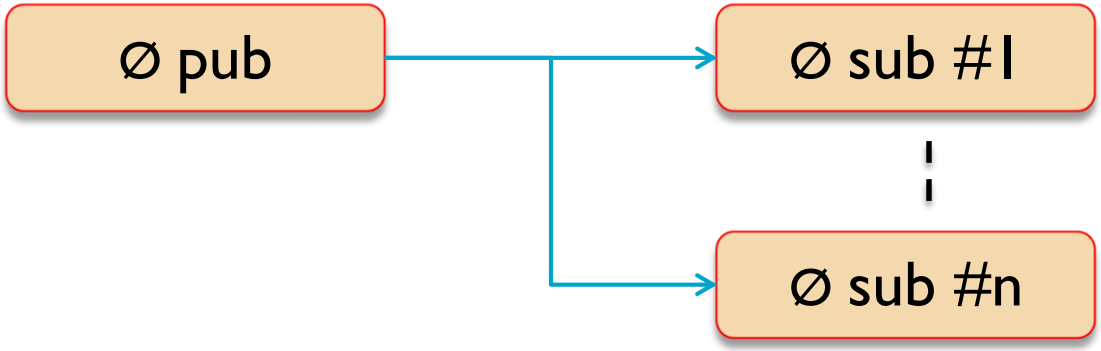
- **identity** es conveniente a la hora de conectar con sockets "router"
  - Fija el ID del agente que se conecte al "router"
  - Debe utilizarse antes de llamar al método connect().
- **subscribe**, utilizado por sockets "sub"
  - Fija el filtro de prefijos aplicado al socket "pub"

4.3.2 Patrones básicos

- **Req**uest/**Rep**ly (sincrónico)

- Push-pull

- Pub-Sub



4.3.2 Patrones básicos: **req**uest/**rep**ly

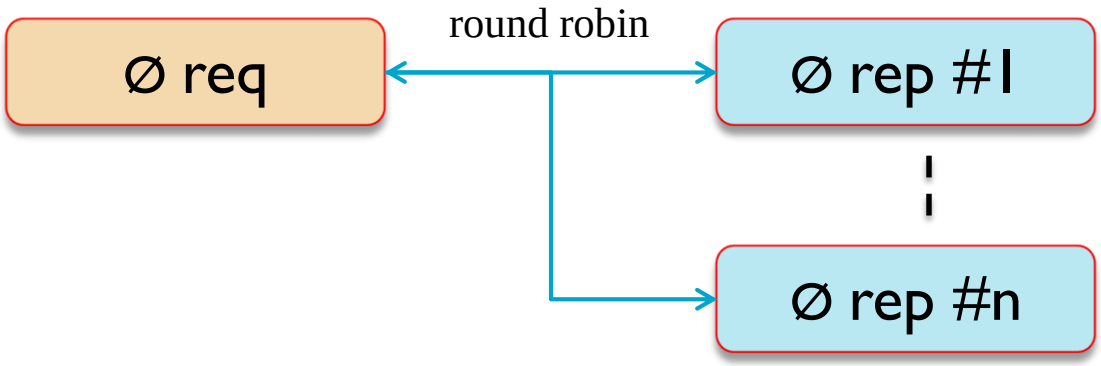
- Implantado mediante sockets …
  - **req** en el cliente
  - **rep** en el servidor
- Cada mensaje enviado vía **req** necesita asociarse a una contestación desde el socket **rep** del servidor

- Patrón de comunicación sincrónico
  - Todos los pares petición/respuesta están totalmente ordenados
  - Los "endpoints" pueden reaccionar asincrónicamente

4.3.2 Patrones básicos: **req**uest/**rep**ly

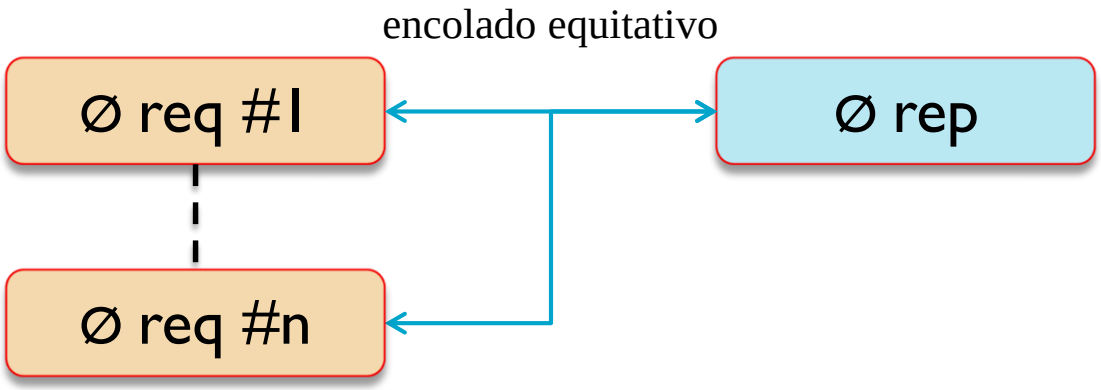
- Cuando se ha enviado un mensaje a través de un socket **req**, otro envío posterior por ese socket será encolado localmente
  - Hasta que se reciba el mensaje de respuesta
    - Entonces se enviará el mensaje encolado
- Cuando se ha recibido un mensaje a través de un socket **rep**, otra recepción posterior por ese socket será encolada localmente
  - Hasta que la respuesta a la solicitud anterior haya sido enviada
    - Entonces el mensaje encolado se entregará a la aplicación
- Cada envío con un socket **rep** queda bloqueado si no se ha recibido previamente su petición asociada por ese mismo socket
  - El bloqueo finaliza al recibir la petición correspondiente

4.3.2 Patrones básicos: **req**uest/**rep**ly con distribución



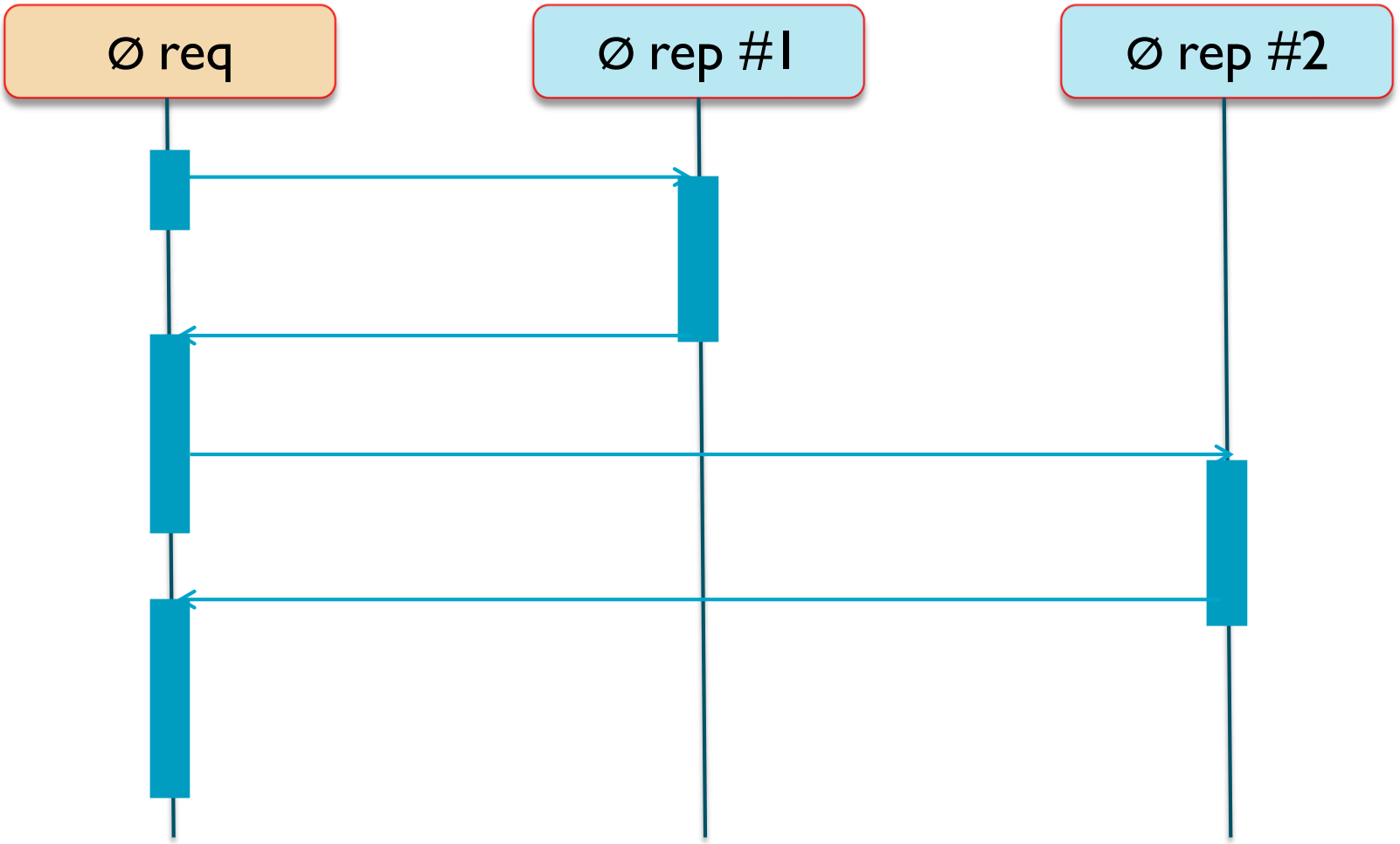
- Cuando un **req** conecta con más de un **rep**, cada mensaje de petición se envía a un **rep** diferente
  - Se sigue una política "round-robin"
- La operación continúa siendo sincrónica:
  - ØMQ no envía nuevas peticiones hasta que cada respuesta sea recibida
  - No hay paralelización de peticiones

4.3.2 Patrones básicos: múltiples peticionarios

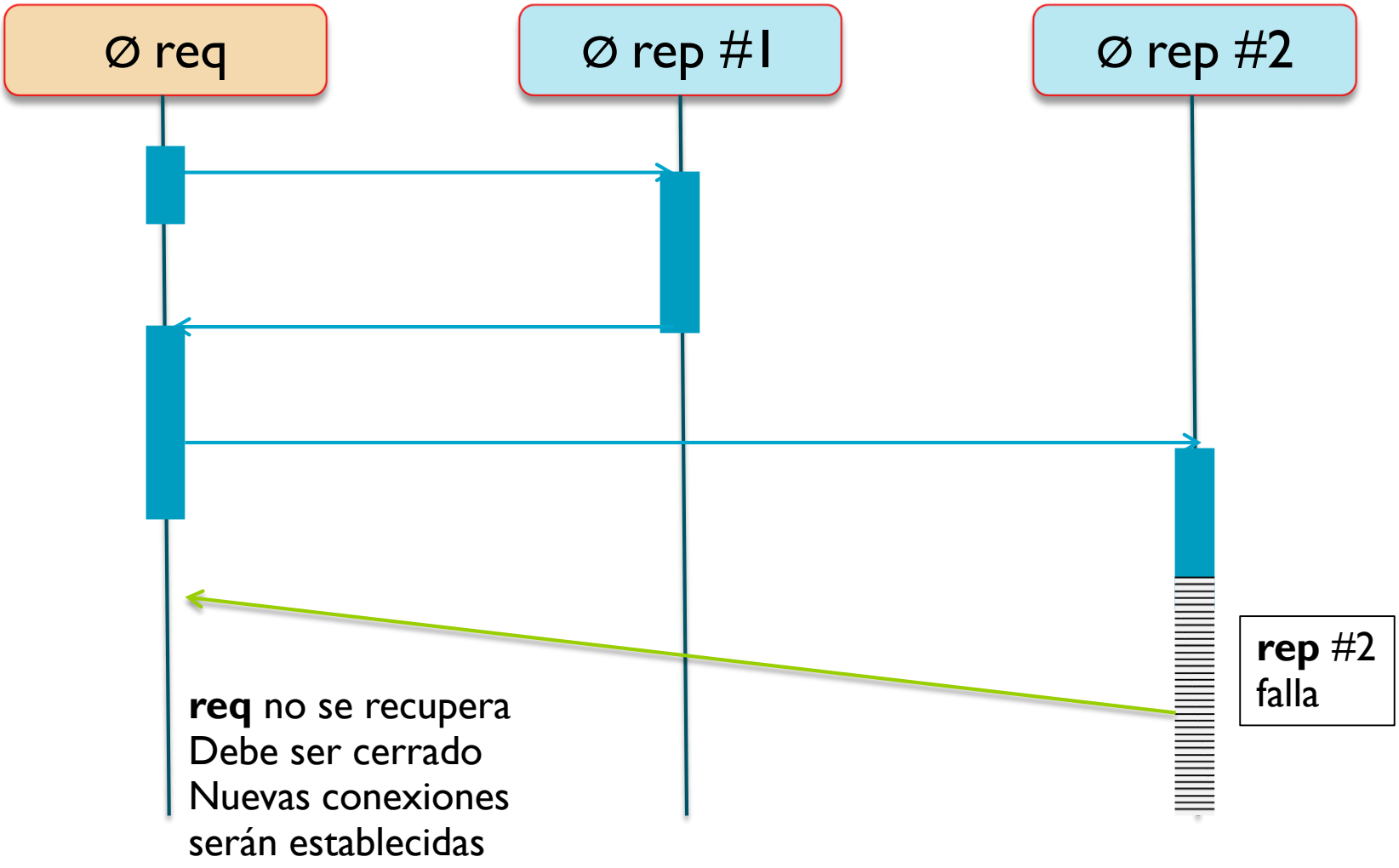


- Configuración típica para un servidor
- El socket **rep** gestiona los mensajes de entrada con una cola
  - Ningún socket **req** sufrirá inanición

4.3.2 Patrones básicos: Secuencia petición/respuesta **sin fallos**



4.3.2 Patrones básicos: petición/respuesta **con fallos**



4.3.2 Patrones básicos: req/rep mínimo

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

```
const zmq = require('zeromq')
const 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')
})
```

4.3.2 Patrones básicos: req/rep, dos servidores

```
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('Hello')
rq.send('Hello again')

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

```
const zmq = require('zeromq')
const 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')
})
```

```
const zmq = require('zeromq')
const 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')
})
```

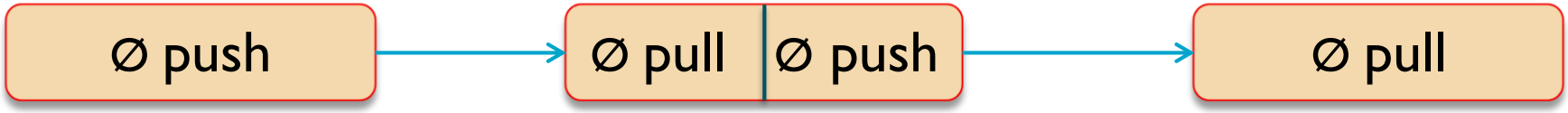
4.3.2 Patrones básicos: req/rep, estructura de los mensajes

- 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

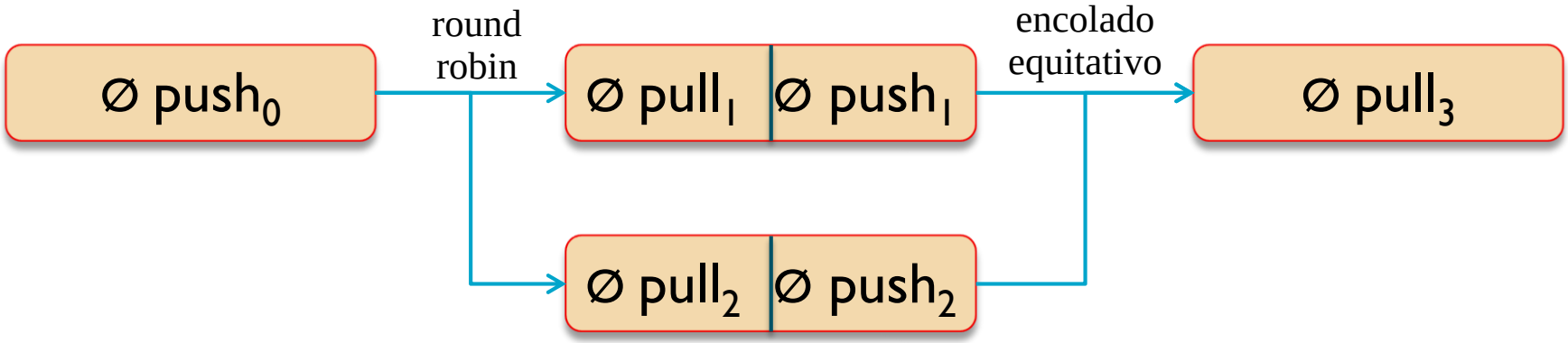
0	""
7	"esto es"
3	"una"
8	"petición"

4.3.2 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:



4.3.2: Patrones: ejemplo push/pull, productor/consumidores

```
const zmq = require("zeromq")
const producer = zmq.socket("push")
let 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)
})
```

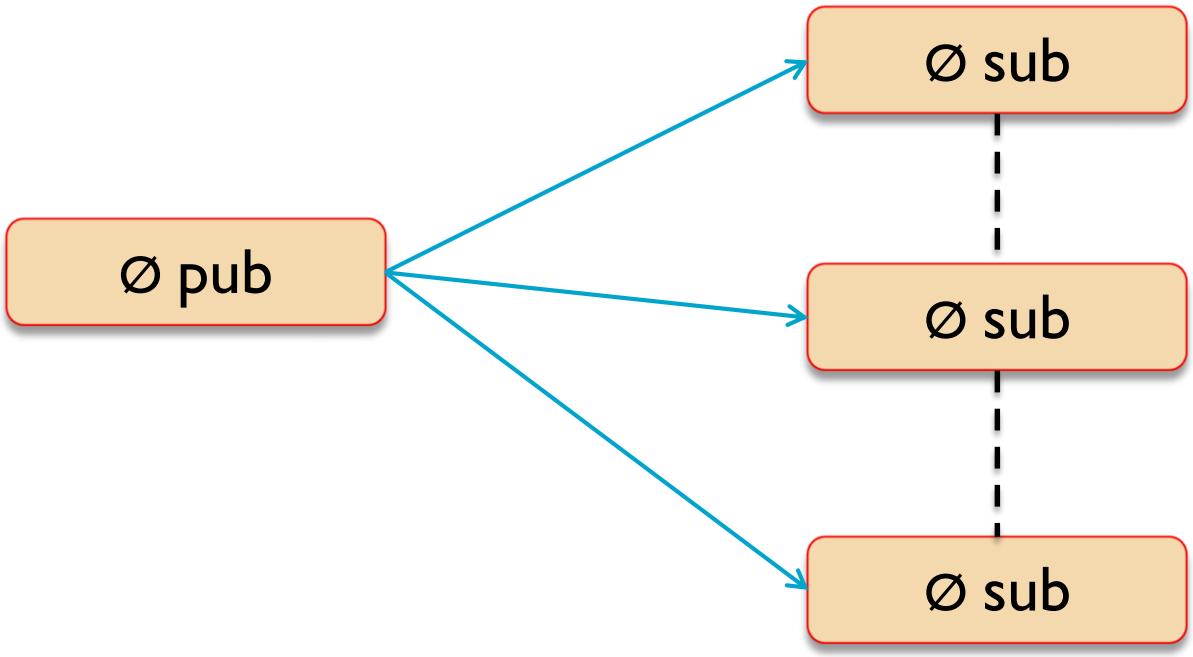
```
const zmq = require("zeromq")
const consumer = zmq.socket("pull")

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

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

4.3.2 Patrones básicos: Publish/Subscribe (pub/sub)

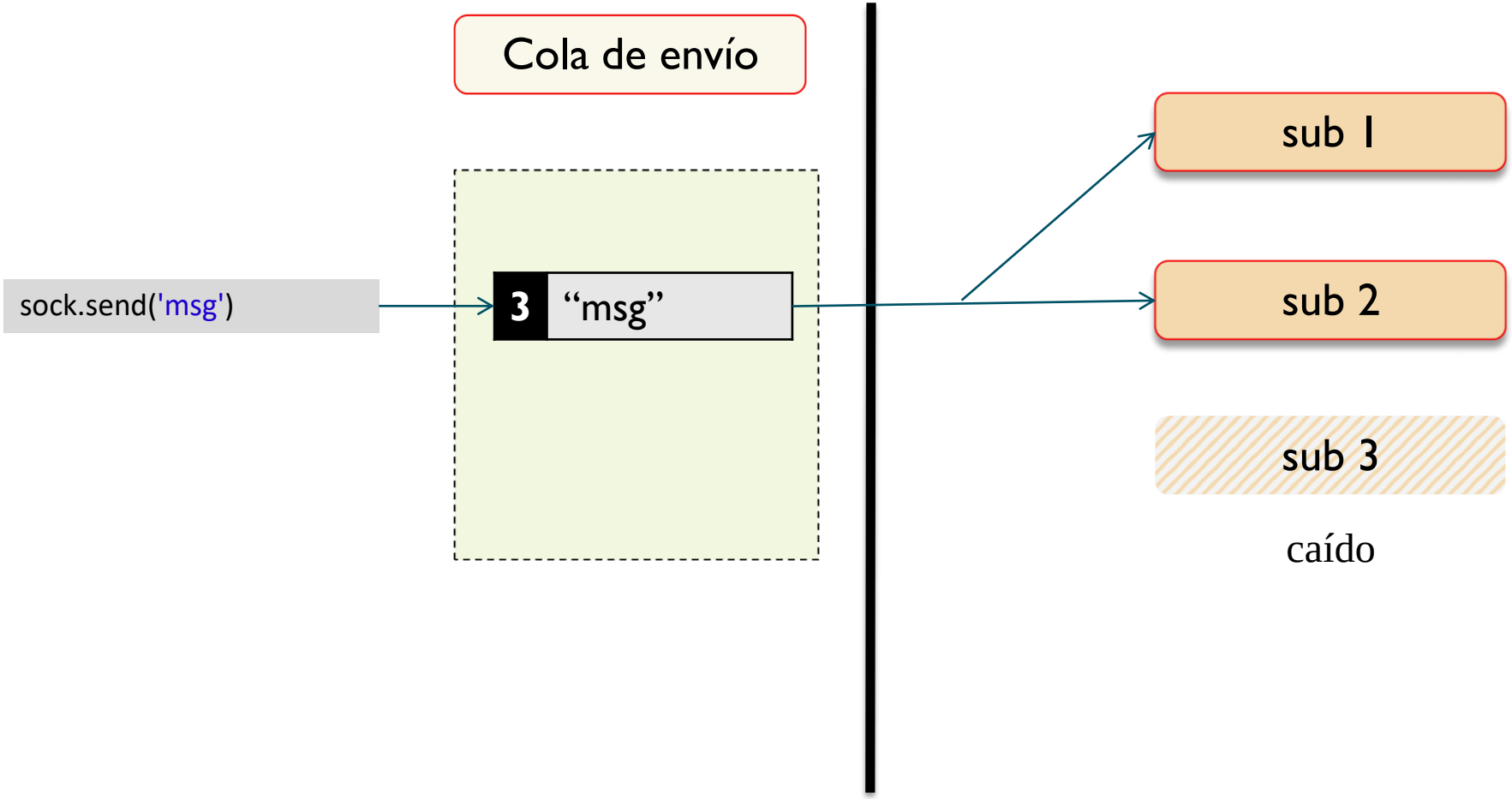
- Este patrón implanta la difusión de mensajes...



- ... con **una condición**: los receptores pueden decidir que se suscriben solo a ciertos mensajes
  - Se trata de un multienvío

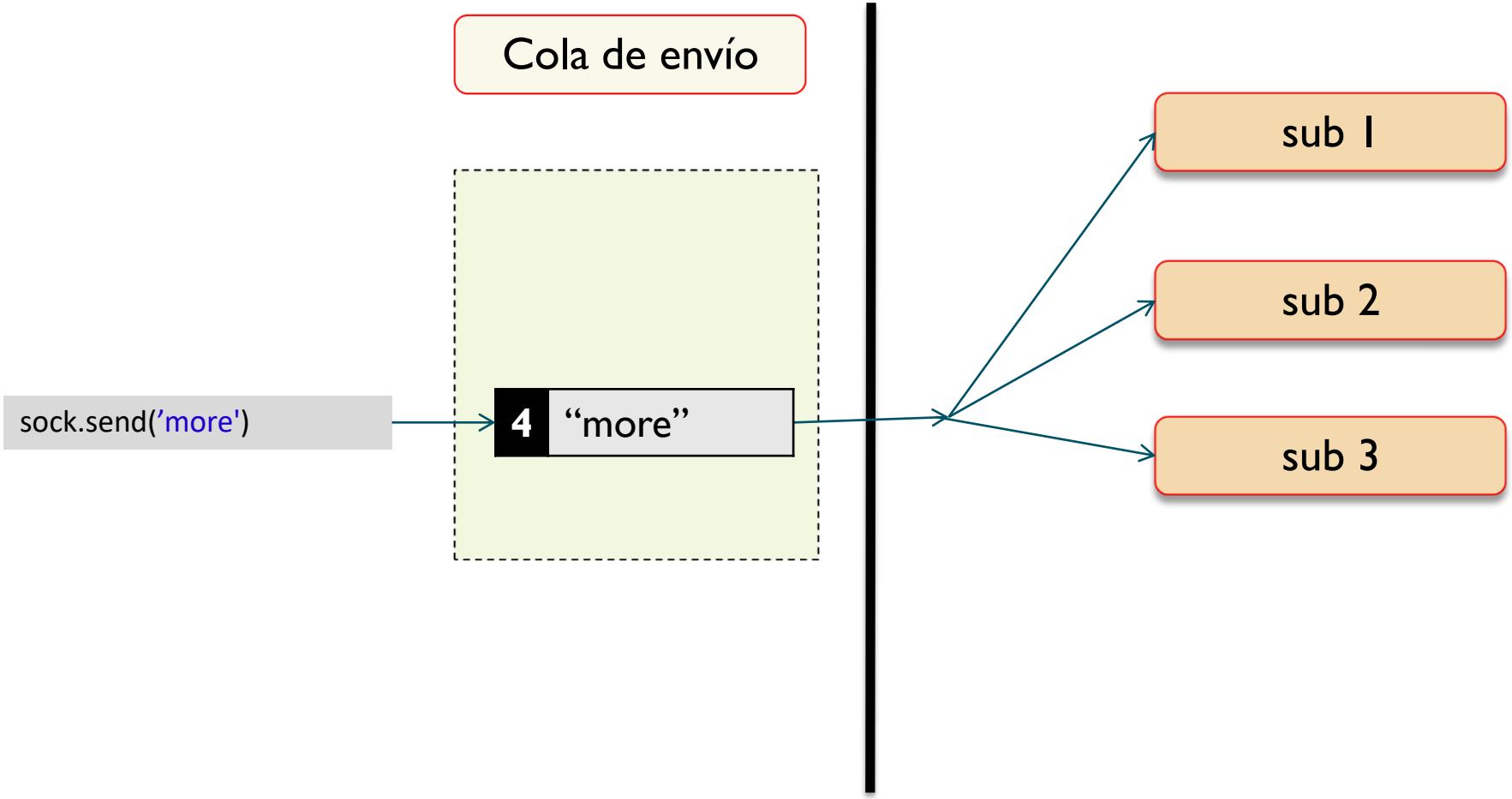
4.3.2 Patrones básicos: pub/sub

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



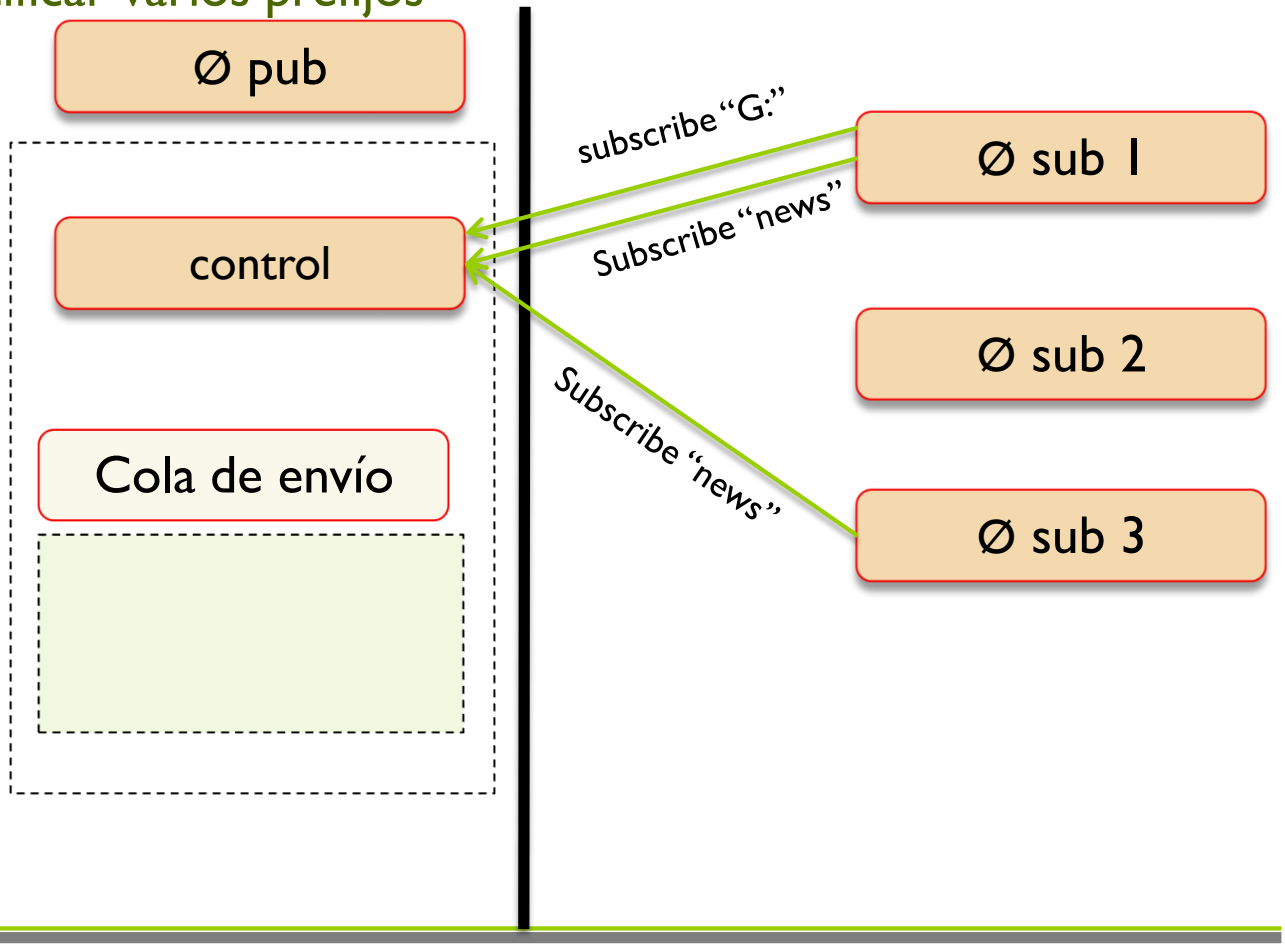
4.3.2 Patrones básicos: pub/sub

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



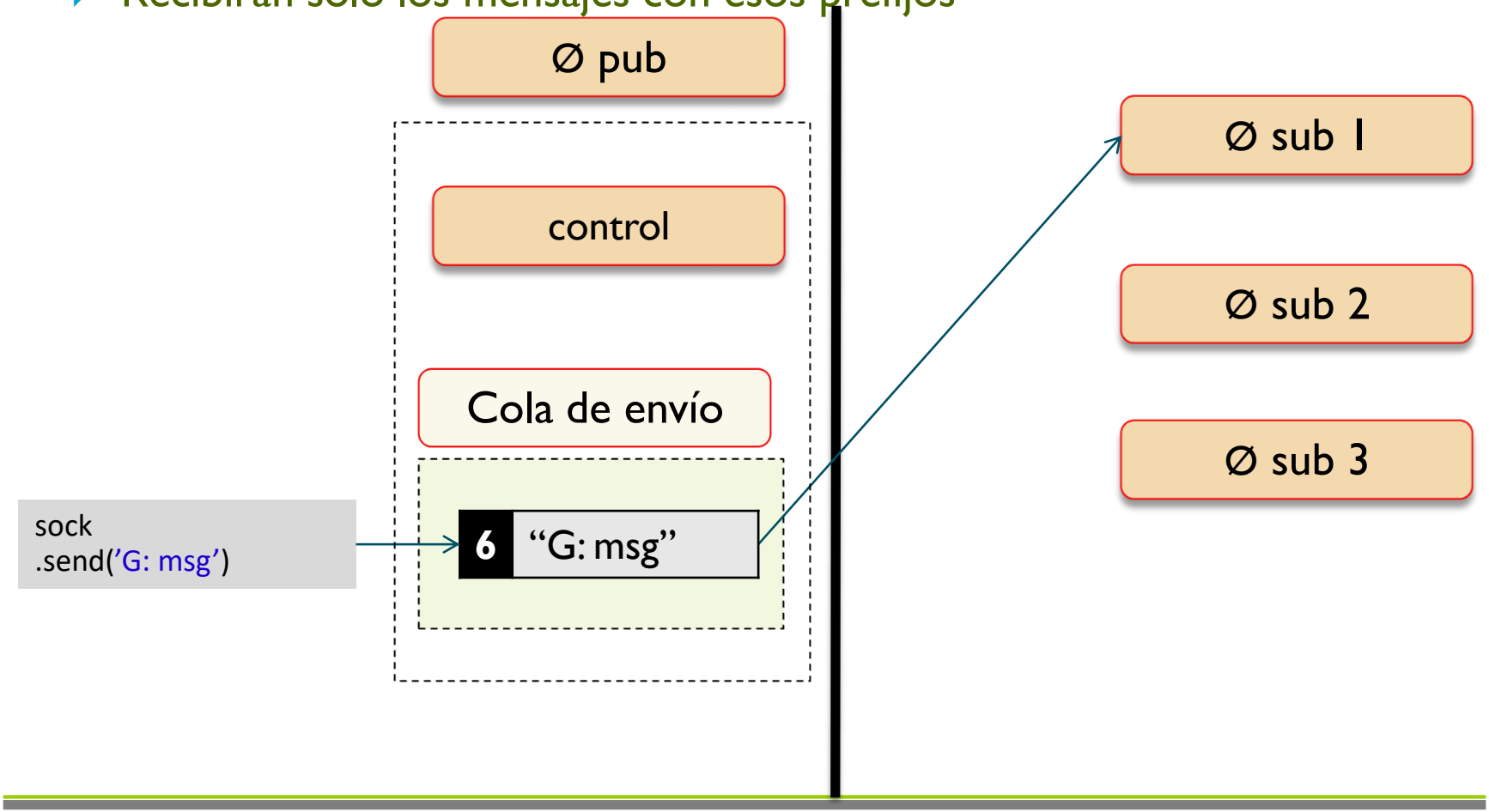
4.3.2 Patrones básicos: pub/sub: Suscripción/filtrado

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



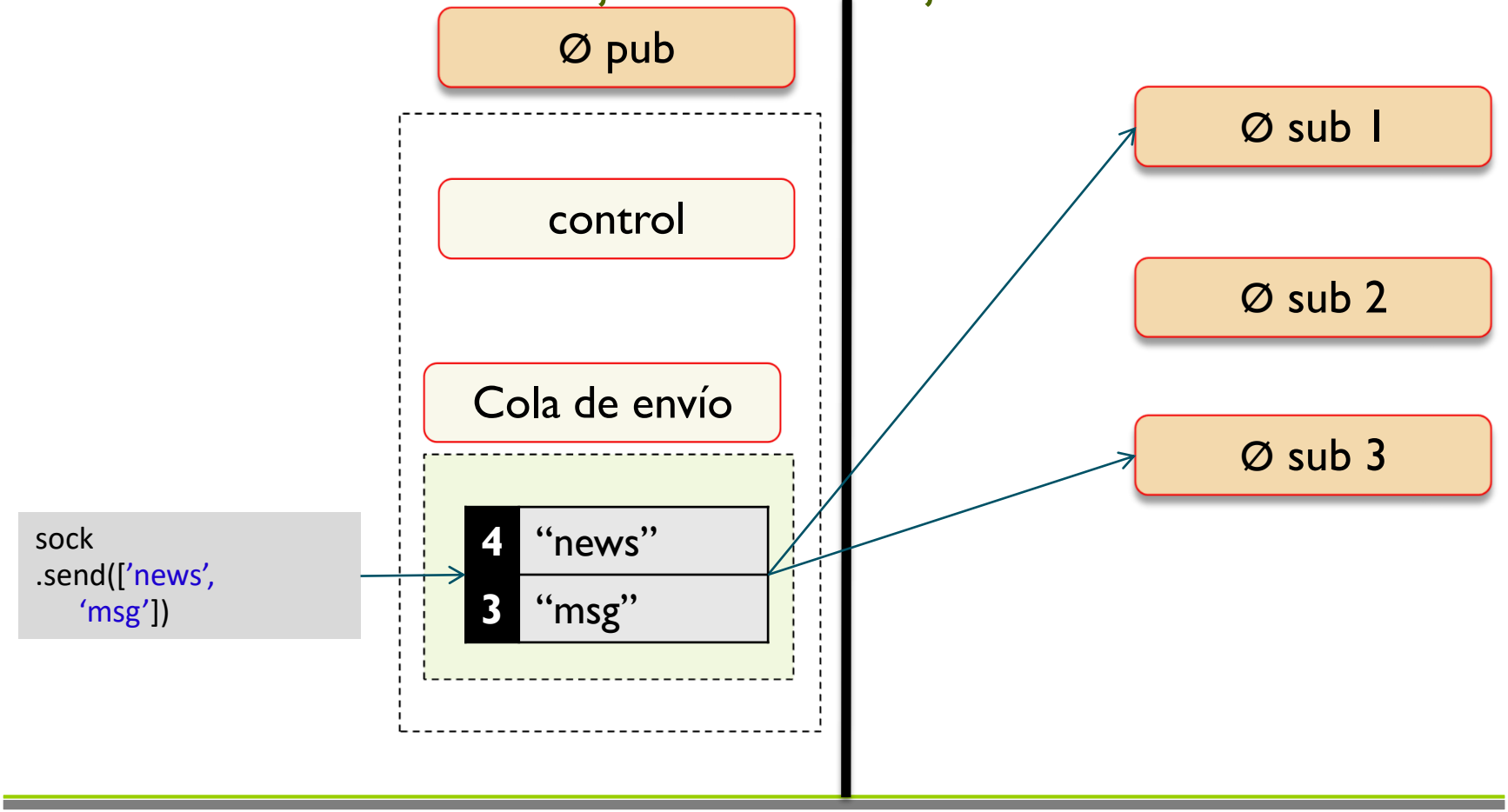
4.3.2 Patrones básicos: pub/sub: Suscripción/filtrado

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



4.3.2 Patrones básicos: pub/sub: Suscripción/filtrado

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



4.3.2 Patrones básicos. Ejemplo pub/sub

```
const zmq = require("zeromq")
const pub = zmq.socket('pub')
let count = 0

pub.bindSync("tcp://*:5555")

setInterval(function() {
  pub.send("TEST " + count++)
}, 1000)
```

```
const zmq = require("zeromq")
const sub = zmq.socket('sub')

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

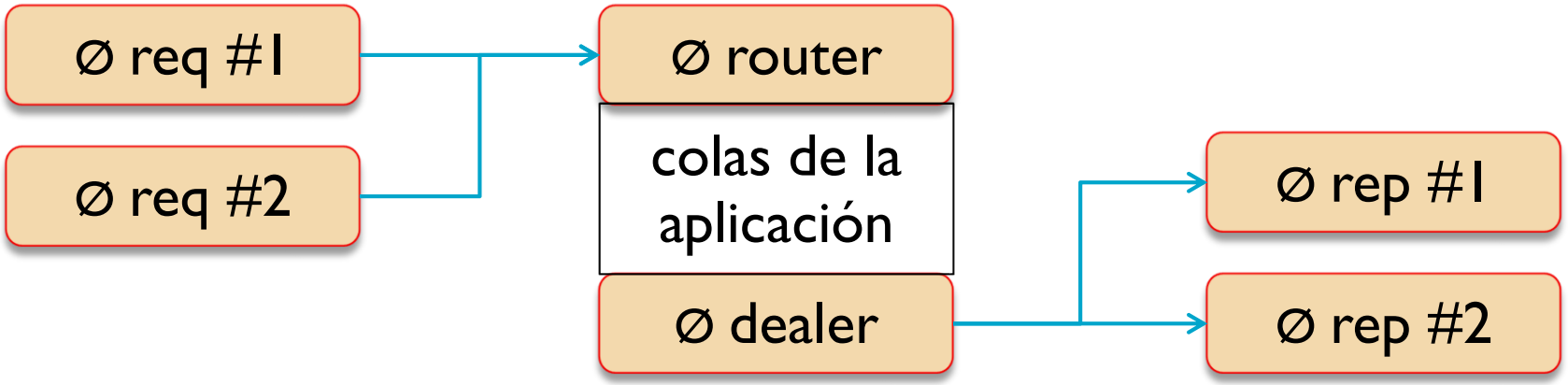
Los mensajes más antiguos podrían perderse si el suscriptor empieza tarde

4. ZeroMQ

1. Introducción
2. Middleware
3. Sistemas de mensajería
4. ZeroMQ
   1. Introducción
   2. Mensajes
   3. API de ØMQ
   4. Sockets avanzados
5. Otro middleware
6. Conclusiones
7. Referencias

4.4 Tipos de "sockets" avanzados

1. Dealer
   - Similar a **req**, pero asincrónico (cliente)
2. Router
   - Similar a **rep**, pero asincrónico y con capacidad para distinguir entre agentes (para encaminar las respuestas) (servidor)
3. Normalmente se implantan juntos en un mismo agente



4.4.1 Sockets dealer

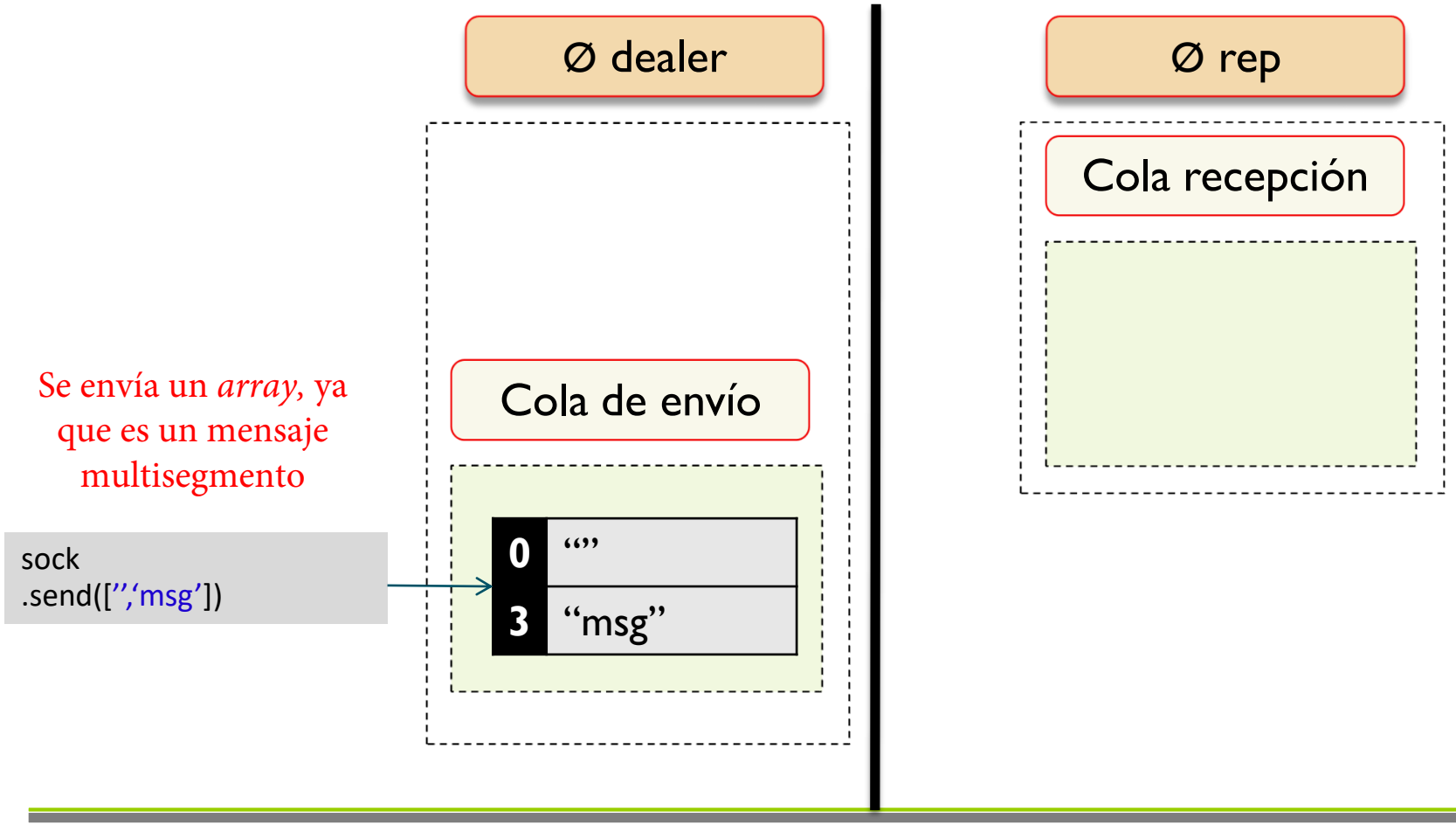
- Es un socket asincrónico de propósito general
- Usado frecuentemente como socket **req** asincrónico
  - No se bloquea por fallos en los agentes
  - PERO, debe construir un mensaje de petición adecuado
    - Con segmento vacío (delimitador) antes del cuerpo real del mensaje
    - Puede ubicar tras el delimitador cualquier número de segmentos

Si el *dealer* se comunica con un *router* (servidor asincrónico), estas precauciones **no serían necesarias.**

Únicamente hay que tenerlas en cuenta para **intreracciones con *rep***, ya que espera como delimitador una cadena vacía.

4.4.1 Sockets dealer: gestión de peticiones y respuestas

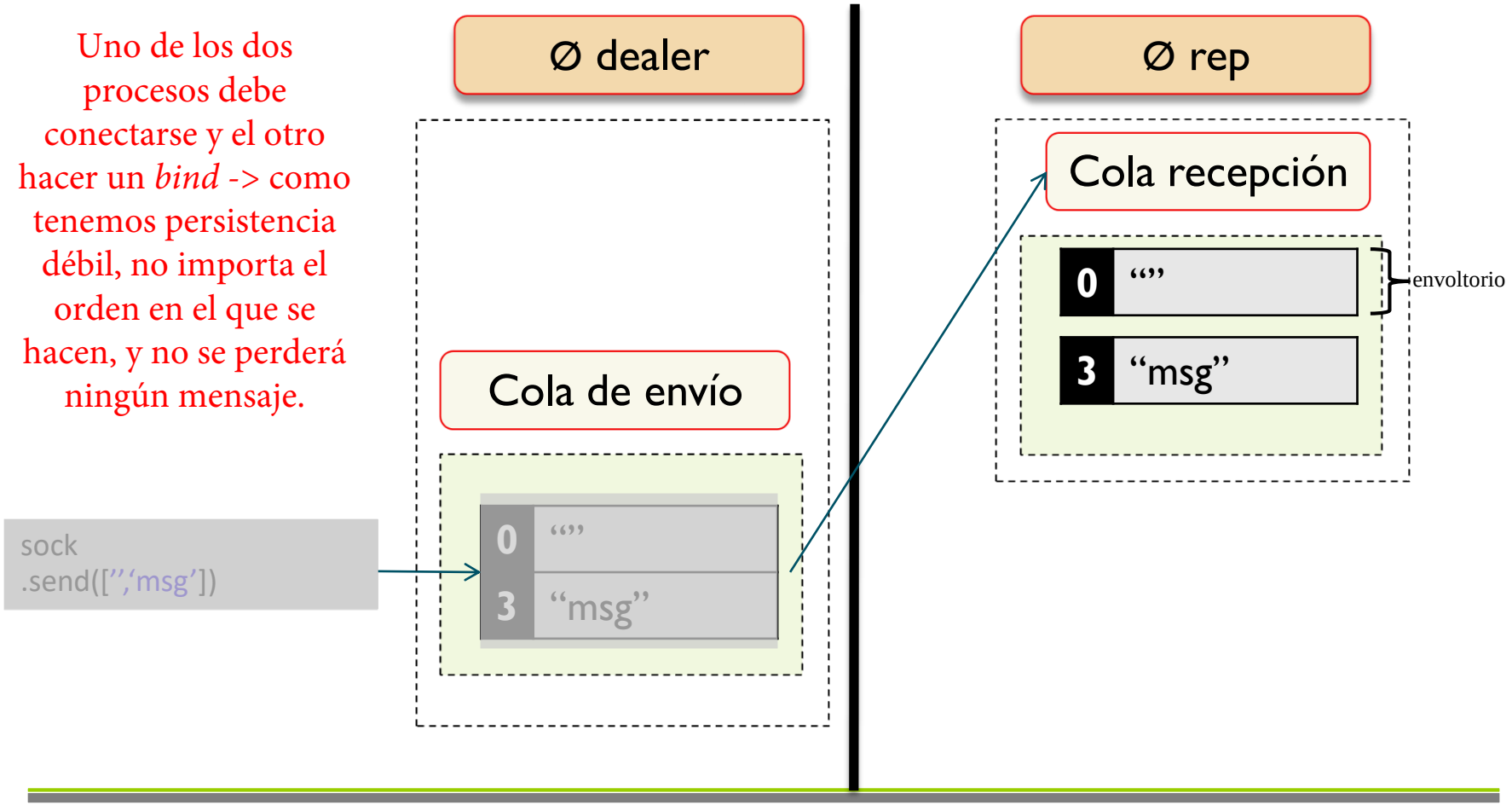
- El delimitador debe ser añadido (como cabecera) para comunicarse con un **rep**:



4.4.1 Sockets dealer: gestión de peticiones y respuestas

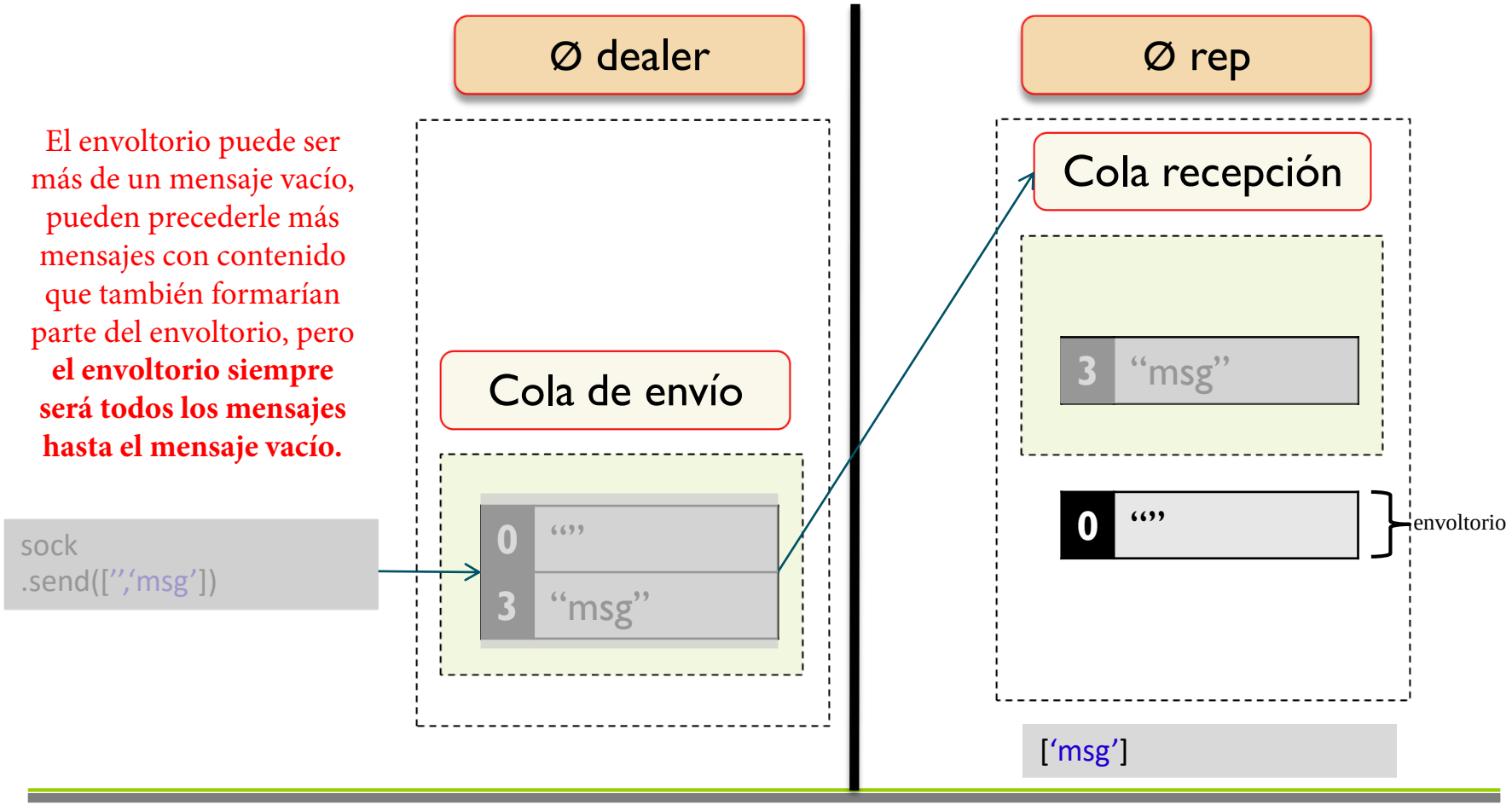
- Cuando se reciba, el socket **rep** quita el "envoltorio"

Uno de los dos procesos debe conectarse y el otro hacer un *bind* -> como tenemos persistencia débil, no importa el orden en el que se hacen, y no se perderá ningún mensaje.



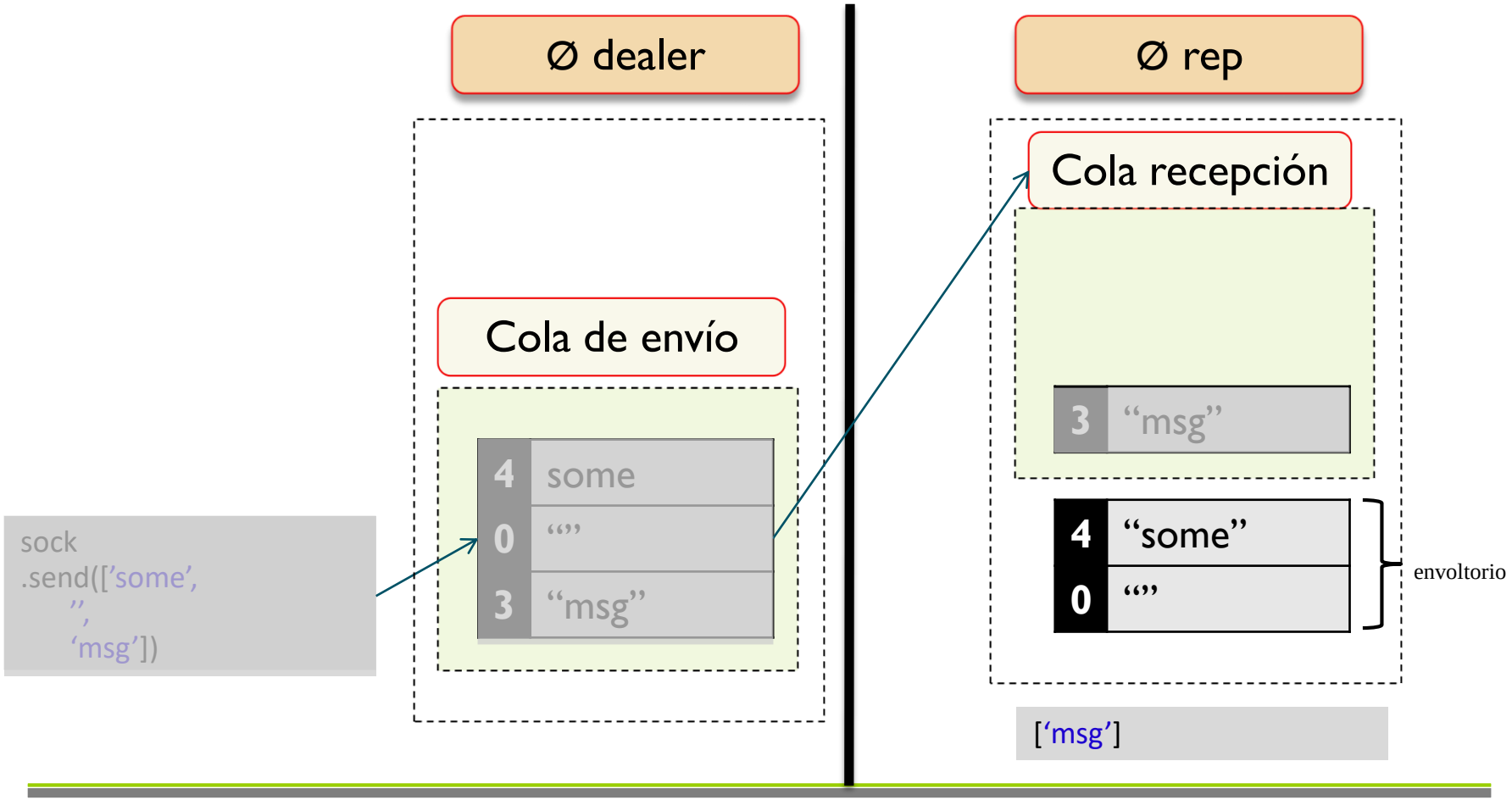
4.4.1 Sockets dealer: gestión de peticiones y respuestas

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



4.4.1 Sockets dealer: envoltorio

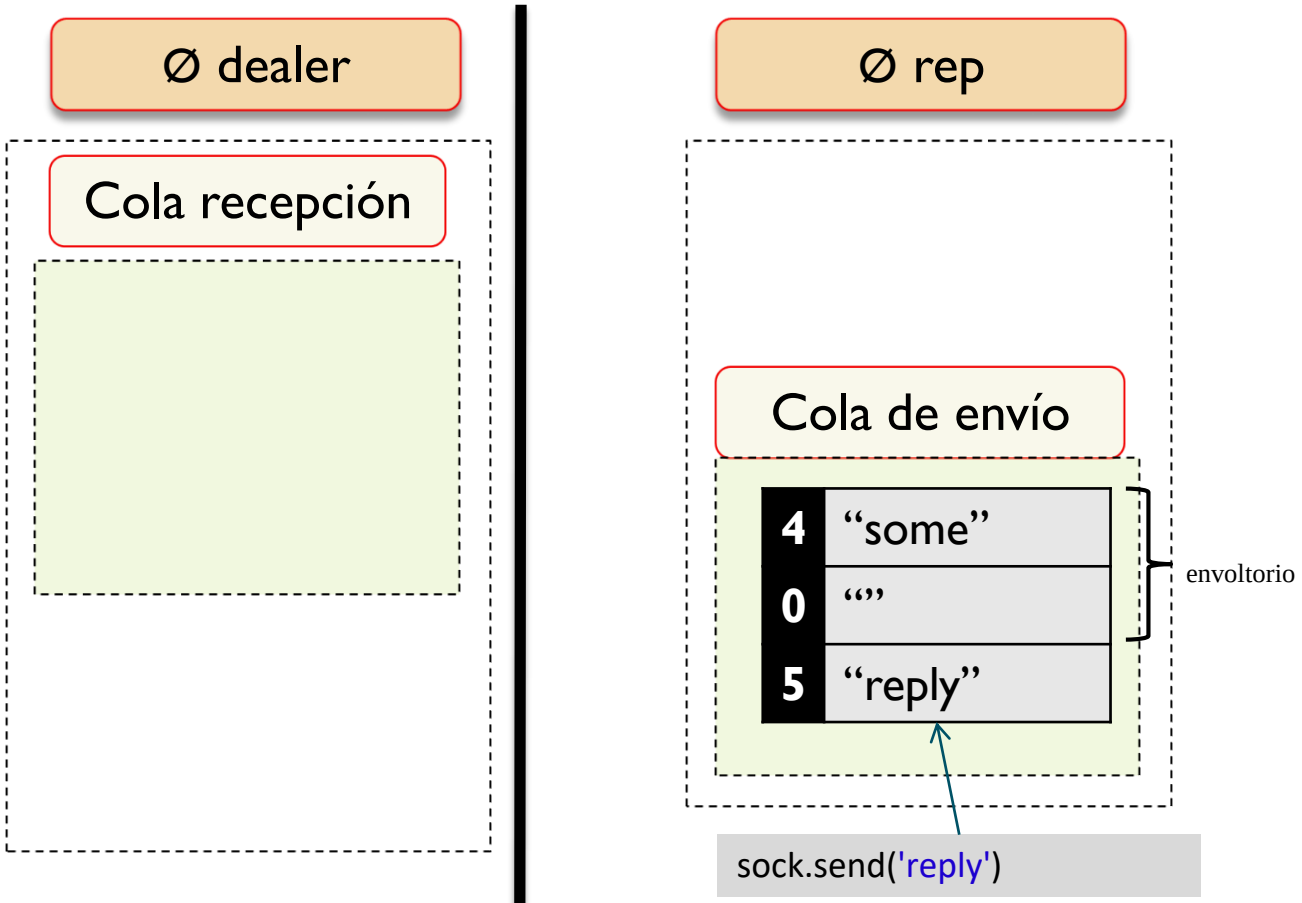
- El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - El envoltorio es guardado por el **rep**...



4.4.1 Sockets dealer: envoltorio

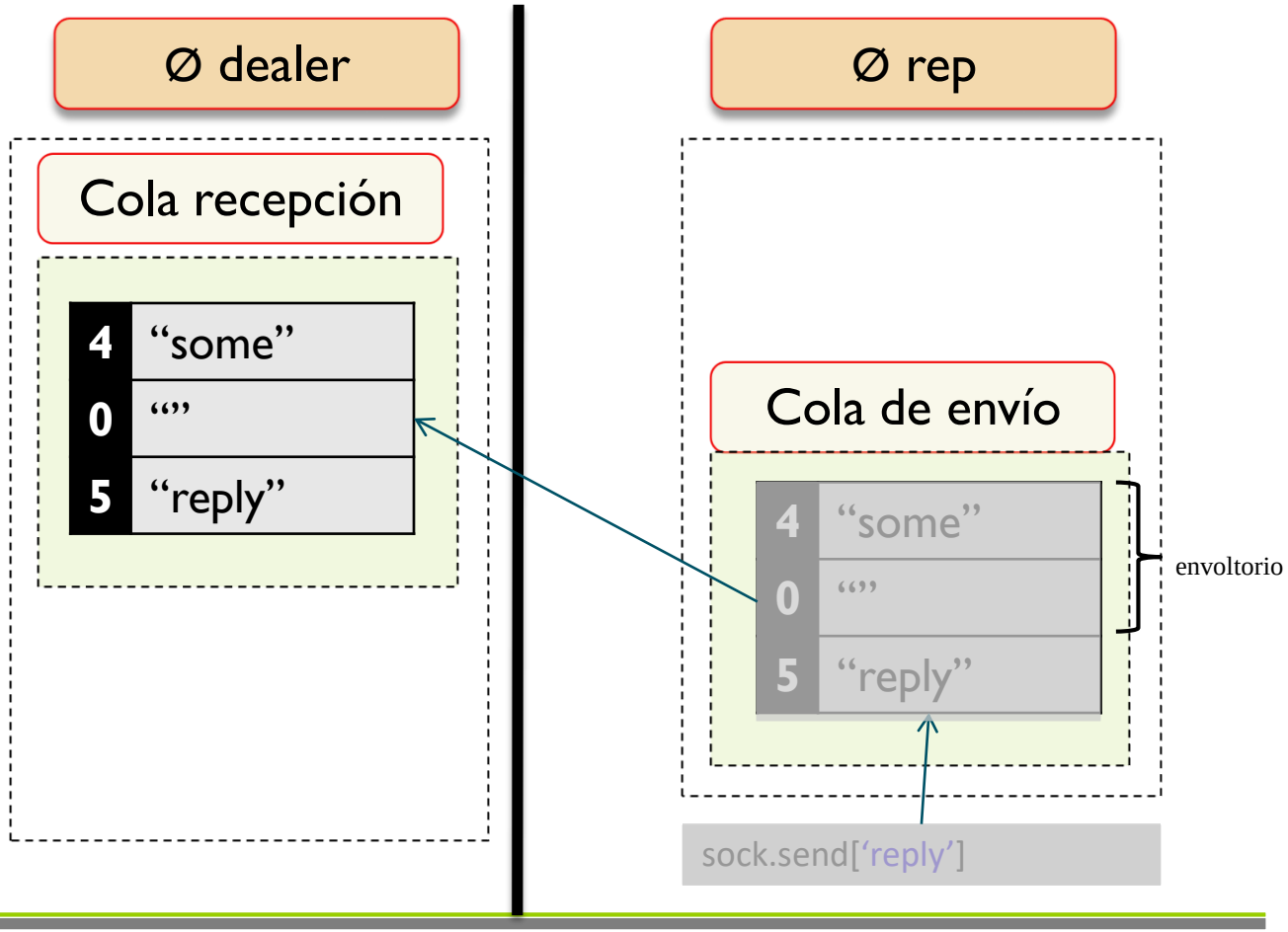
- El envoltorio es más general: Todos los segmentos hasta el primer delimitador
  - El envoltorio es guardado por el **rep... y reinsertado en la respuesta**

En la respuesta de *rep*, reinserta el delimitador en el envío



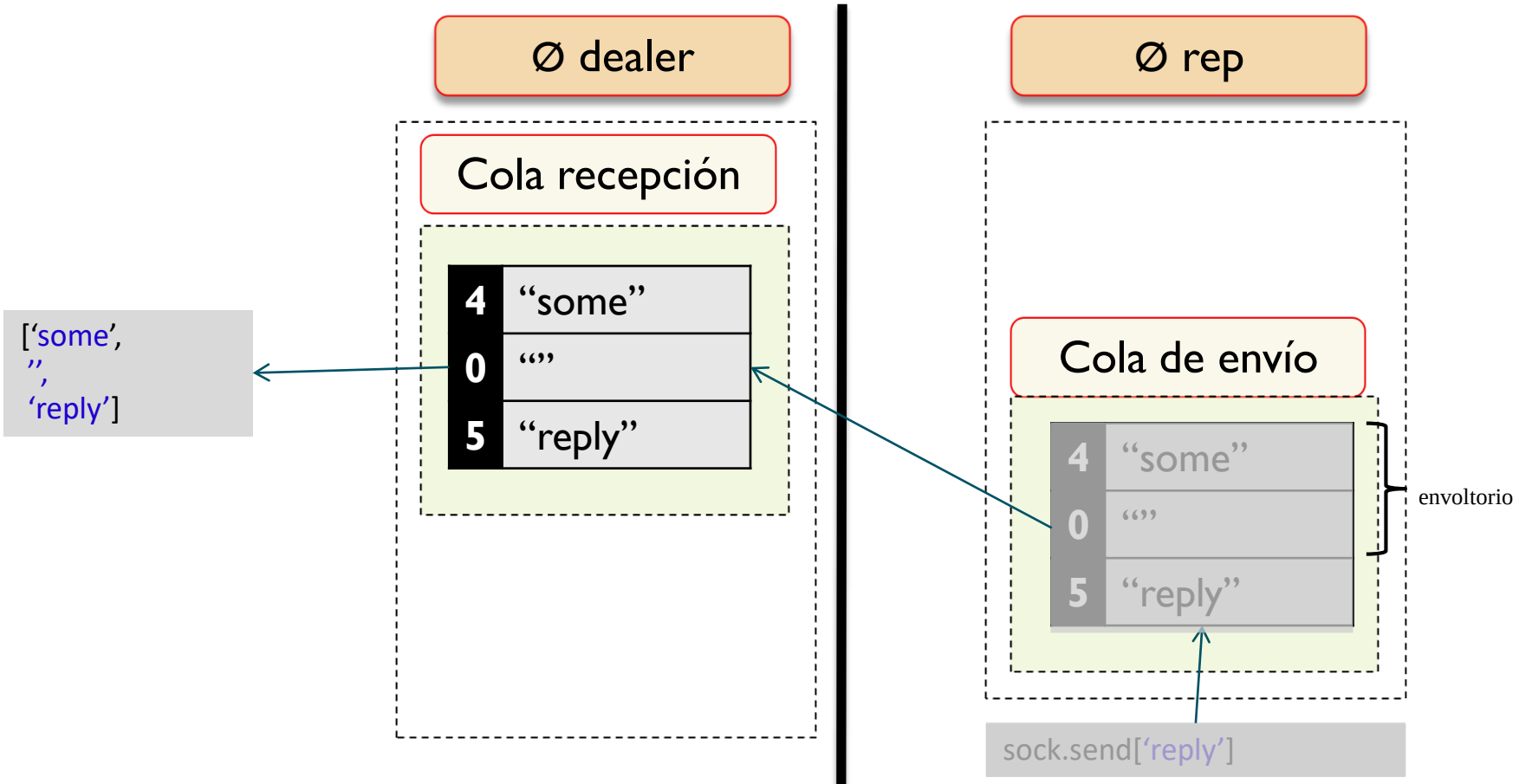
4.4.1 Sockets dealer: envoltorio

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



4.4.1 Sockets dealer: envoltorio

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



4.4.1 Sockets dealer: ejemplo de código

```
const zmq  = require('zeromq')
const dealer = zmq.socket('dealer')
let msg = ["", "Hello ", 0]
const host = ”tcp://localhost:888”

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

setInterval(function() {
  dealer.send(msg)
  msg[2]++
}, 1000)
                          // se utiliza el evento 'message' para recibir mensajes
dealer.on('message',
  function(h, seg1, seg2) {
    console.log('Response:’ + seg1 + seg2)
})
```

Como hay dos conexiones (dos *connect*), la mitad de mensajes va a cada conexión. La primera conexión gestionará los mensajes pares y la segunda, los impares.

```
const zmq = require('zeromq')
const rep = zmq.socket('rep')

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

```
const zmq = require('zeromq')
const rep = zmq.socket('rep')

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

4.4.2 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', se debe dar identidad al *socket* cliente antes de que se conecte al router.
    - 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

4.4.2 Sockets router

- Cuando el socket router envía un mensaje...:
  - Usa su primer segmento como la identidad de conexión, así...
    - Un socket router mantiene un par de colas de envío y recepción de mensajes **por conexión**.
    - Ese primer segmento se usa para localizar la conexión adecuada (el par de colas). Una vez encontrado...:
      - El primer segmento se elimina de manera implícita.
        - El programador no necesita hacer nada.
      - El resto del mensaje se deja en la cola de envío de esa conexión.
        - Eso completa el envío del mensaje.
  - Esto permite una gestión router-dealer muy sencilla en un *broker*:
    - El proceso *broker* usa un router "frontend" y un dealer "backend", es decir, router interactúa con clientes y dealer con servidores.
    - Cada mensaje recibido en el router se envía por el dealer.
    - Cada mensaje recibido en el dealer se envía por el router.
    - En ambos casos, no se necesita ninguna transformación de los mensajes.

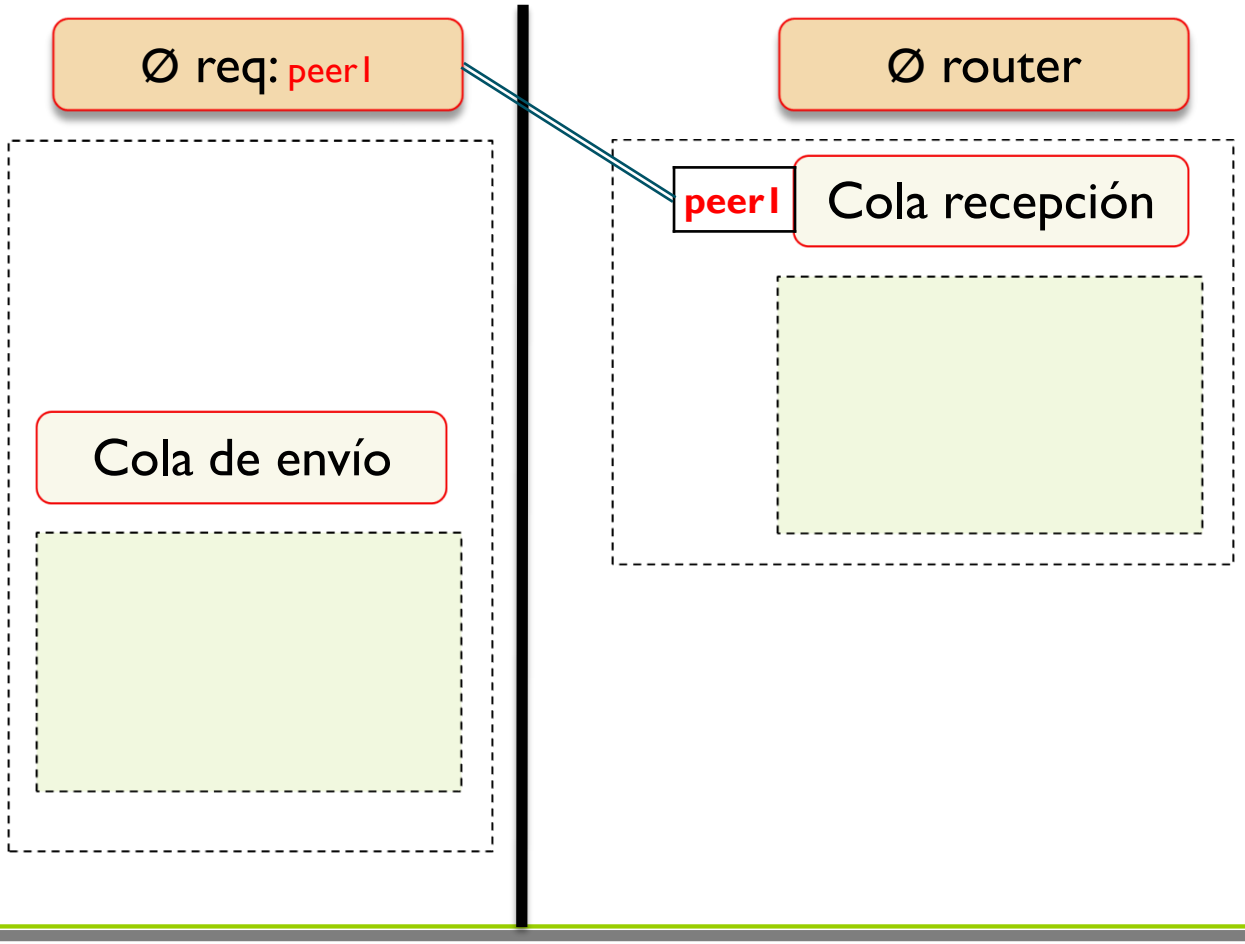
4.4.2 Sockets router: ejemplo con agente req

0. El agente req tiene identidad "peer1"

A partir de aquí la comunicación se desarrolla en 7 grandes pasos

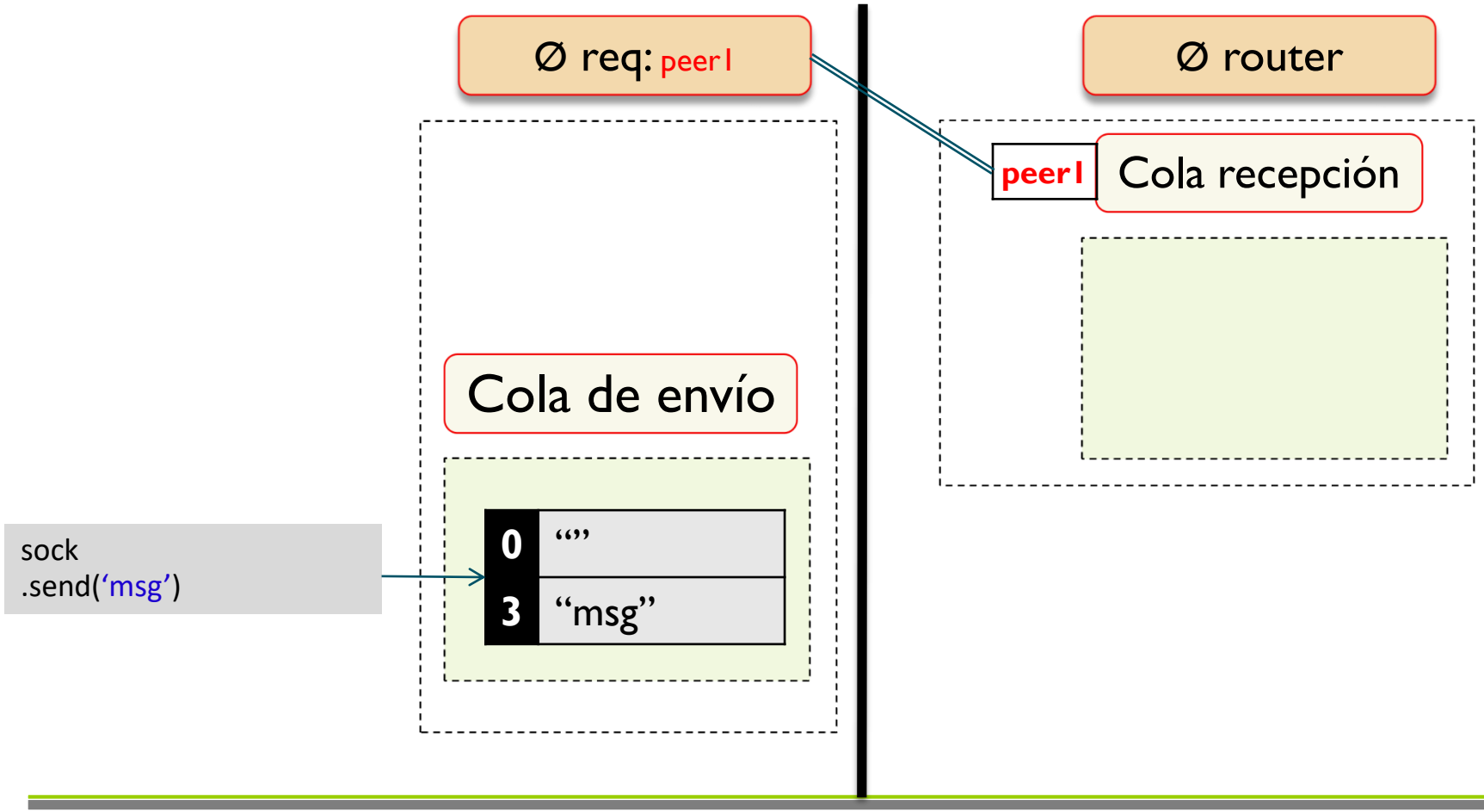
4.4.2 Sockets router: ejemplo con agente req

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



4.4.2 Sockets router: ejemplo con agente req

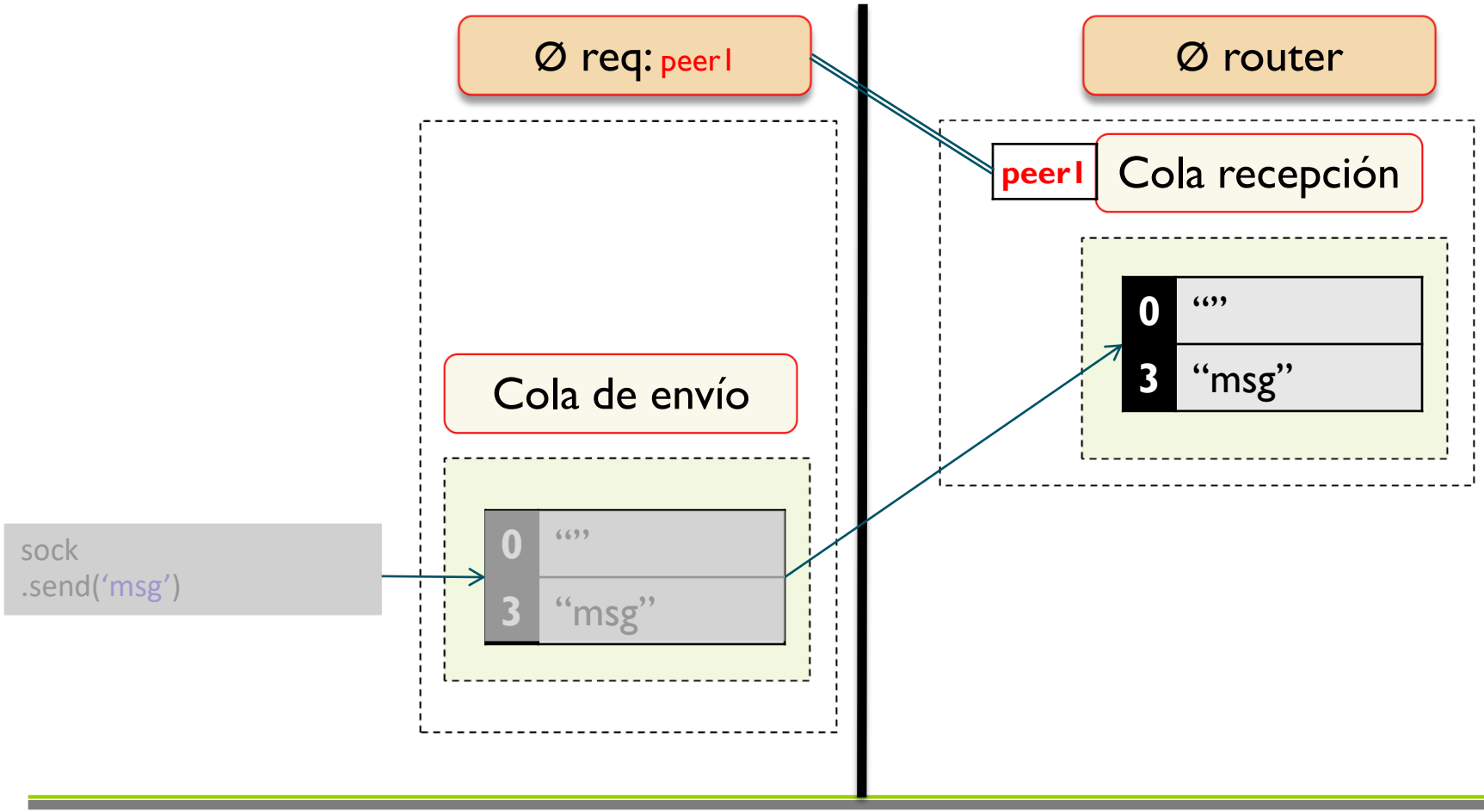
2. La aplicación req envía un mensaje
   2.1 El socket req **añade el delimitador**... automáticamente



4.4.2 Sockets router: ejemplo con agente req

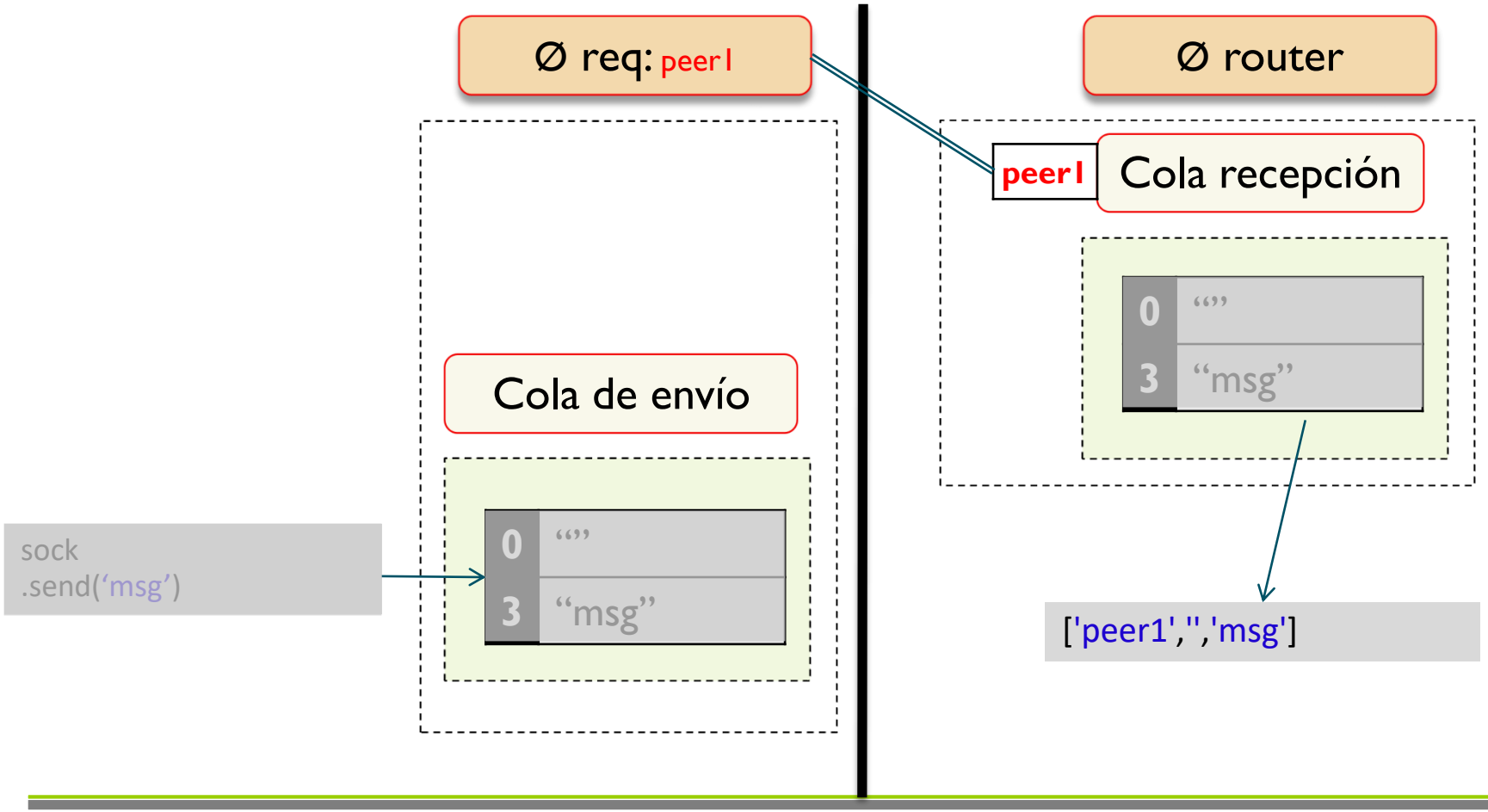
2. La aplicación req envía un mensaje

2.2 El socket req añade el delimitador... **y lo envía**



4.4.2 Sockets router: ejemplo con agente req

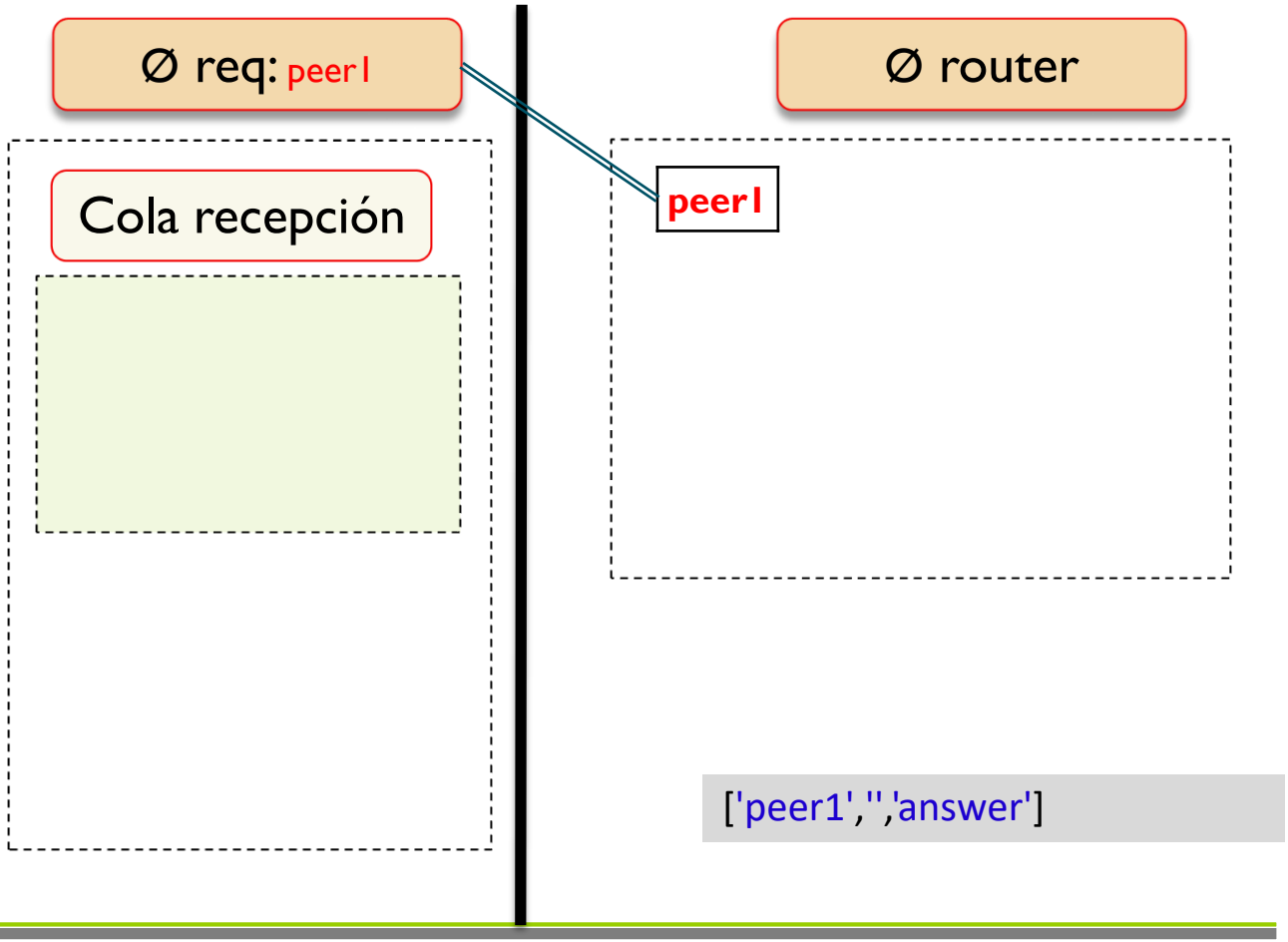
3. El socket router entrega el mensaje a su aplicación
   - Con la identidad del emisor en un nuevo segmento inicial en la cola de recepción



4.4.2 Sockets router: ejemplo con agente req

4. La aplicación del router crea el mensaje de respuesta

- El primer segmento contiene la identidad del agente que recibirá la contestación

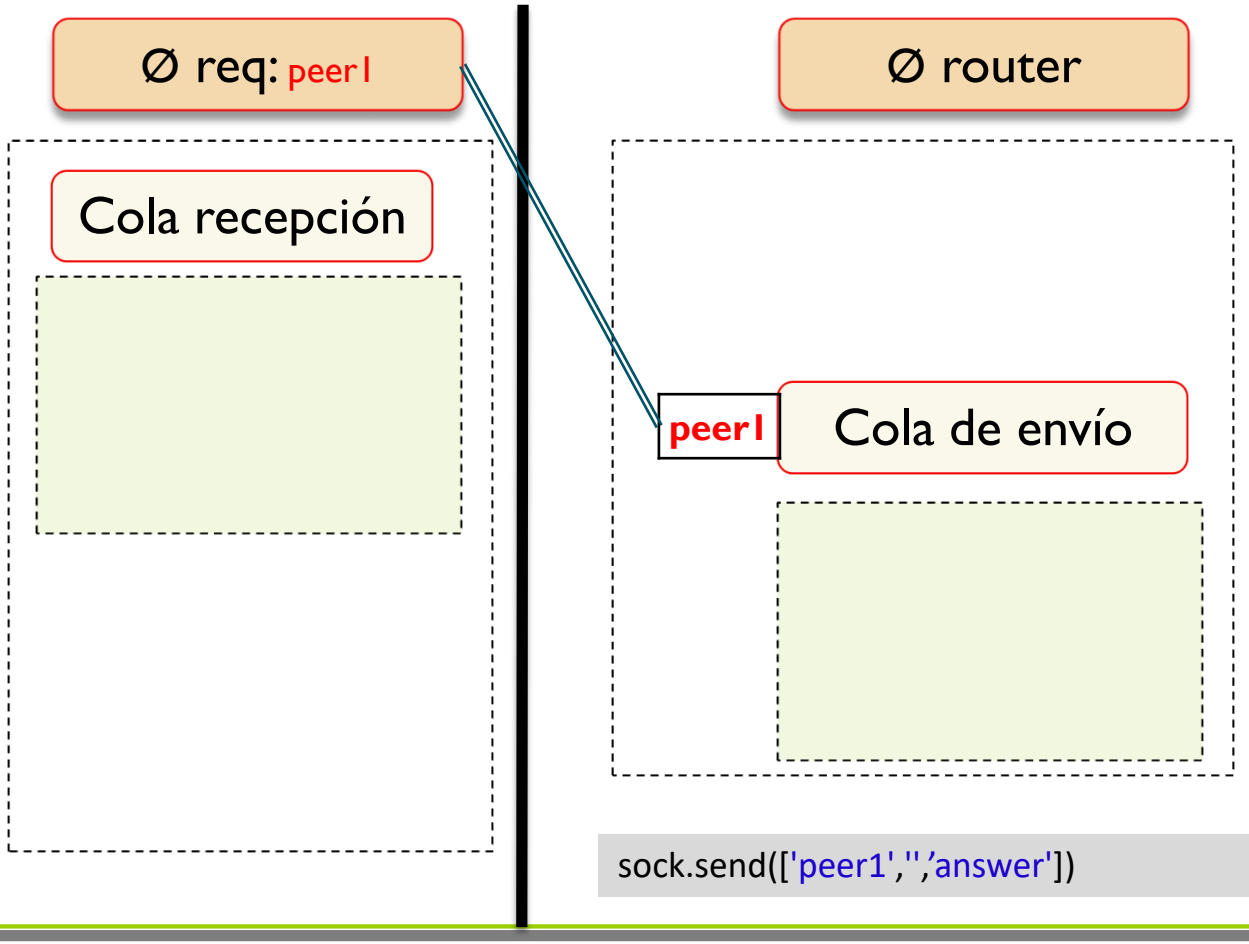


4.4.2 Sockets router: ejemplo con agente req

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

sock.send(['peer1','','answer'])

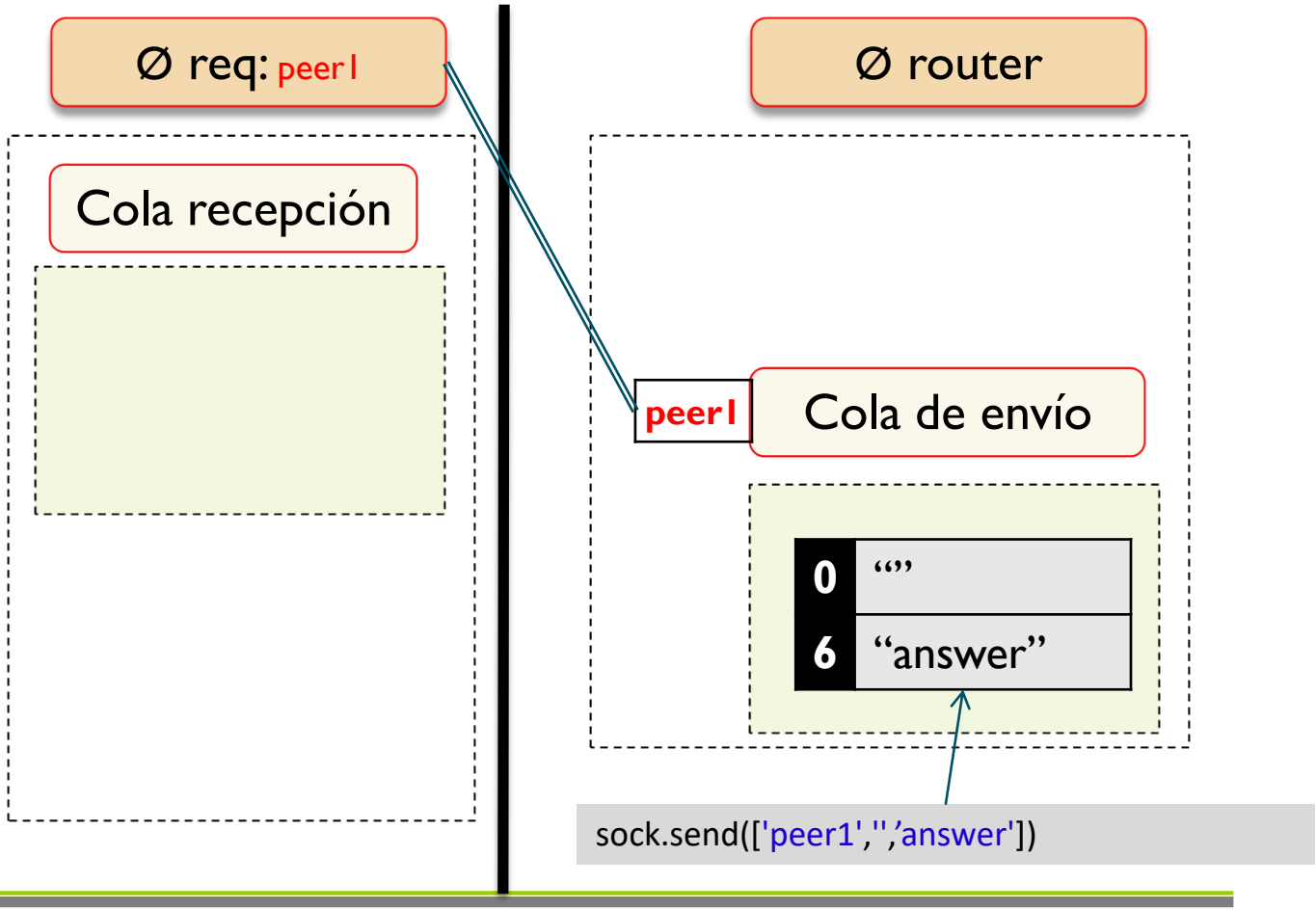
El primer segmento lo consume el *router* (ya que es el identificador) y el segundo (el segmento vacío), el *req*



4.4.2 Sockets router: ejemplo con agente req

6. El socket router quita el segmento con la identidad

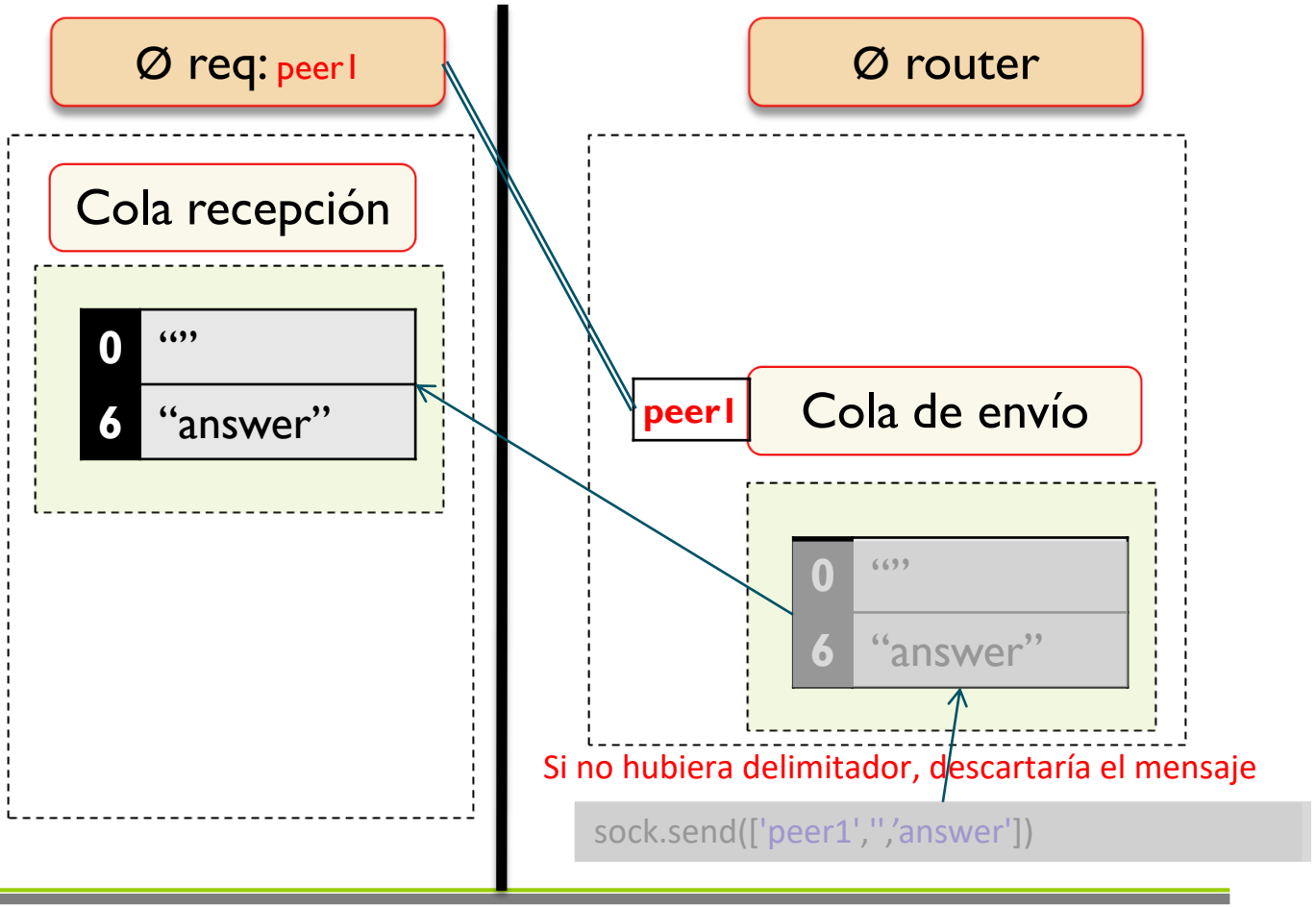
6.1 Deja el resto del mensaje **para enviar**...



4.4.2 Sockets router: ejemplo con agente req

6. El socket router quita el segmento con la identidad

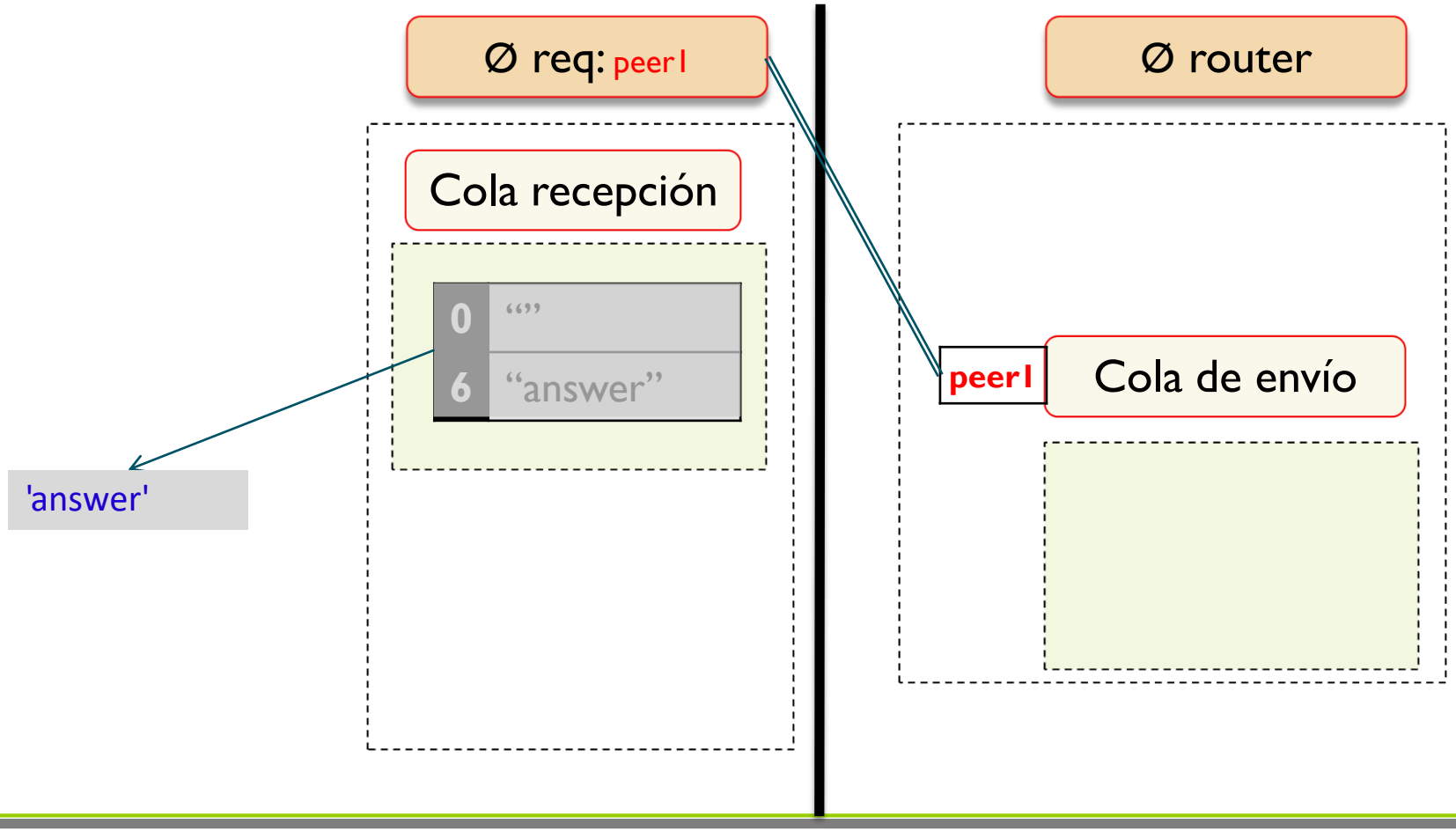
6.2 Deja el resto del mensaje para enviar... **y lo envía**



4.4.2 Sockets router: ejemplo con agente req

7 (y último). El socket req lo entrega a su aplicación

- Eliminando el segmento delimitador



Índice

5. Otro middleware

- Gestión de eventos
  - A menudo incluido en sistemas de mensajería
    - Patrón publicador-suscriptor
  - Ejemplo: JINI
- Seguridad
  - Autenticación
    - Una tercera parte garantiza la identidad de un agente
    - Ejemplo: OpenID
  - Autorización
    - Una tercera parte autoriza una petición
    - Ejemplo: OAuth
  - Integración con otros protocolos
    - Ejemplo: SSL/TLS y HTTPS
- Soporte transaccional
  - Coordinación de modificaciones de estado distribuidas atómicas
    - Soporta las situaciones de fallo

Índice

Conclusiones

- La complejidad de sistemas distribuidos debe ser resuelta con una adecuada gestión del código y de los servicios
- Los estándares simplifican este escenario familiarizándonos con las técnicas a utilizar
- Los "middleware" (nivel de la arquitectura entre la aplicación y las comunicaciones) implantan soluciones comunes para estos problemas
- Principales objetivos de los middleware
  - Tareas de comunicaciones
  - Petición de servicios
- Principales variantes
  - Mensajes
    - Gestión persistente/transitoria
    - Basados en gestor / Sin gestor (0MQ no necesita gestor)
- Otros middleware
  - Seguridad
  - Transacciones

Índice

7. Bibliografía

- http://zguide.zeromq.org/page:all
  - Puede leerse on-line.
  - Hay una versión en PDF.
  - En ese URL hay información adicional.