Tema 3 – Middleware. ZeroMQ

Tecnologies dels Sistemes d'Informació en la Xarxa

Índex

1. Introducció
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

1. Introducció

- Els components d'un sistema distribuït de gran escala són:
  - Especificats i desenvolupats de manera independent
    - Sense cap comitè que revise tots els casos d'ús possibles
  - Desplegats autònomament
    - Cadascun dissenyat com un agent
    - Però establint dependències entre ells
      - *Consumint* o *produint* funcionalitat de/per a altres agents
- Necessiten interactuar de manera senzilla i útil
  - P. ex., un servei de planificació de rutes pot dependre d'un servei GIS que facilite informació bàsica sobre distàncies.
  - P. ex., un sistema d'autorització d'entrada pot necessitar un servei remot de reconeixement biomètric.

1. Introducció

- Fins i tot els sistemes fortament units (per tenir pocs components estretament relacionats)...
  - Solen estar desenvolupats per més d'un programador
  - Consten de múltiples components interdependents
- En qualsevol cas:
  - Necessiten resoldre les complicacions d'un entorn distribuït
  - El resultat ha d'estar tan lliure d'errors com siga possible
    - La depuració és molt més complexa en un sistema distribuït
  - El desenvolupament sol estar subjecte a terminis estrictes
    - Ha d'invertir-se el menor temps possible en la codificació

1. Introducció

- Com assegurar que no es necessita ser un geni per a...
  - ...escriure els "milions" de línies de codi necessàries?
  - ...desplegar i gestionar els sistemes resultants?
- Un problema és la complexitat dels detalls
  - Resoldre aquestes complexitats és una font d'errors
  - Preocupar-se per massa detalls alhora és una recepta per a aconseguir el desastre
    - Al que ha d'afegir-se les llargues sessions de depuració
      - Freqüentment després del desplegament, en producció
- Un altre problema: moltes tasques són repetitives
  - Podríem estalviar recursos reutilitzant solucions prèvies

1. Introducció

- Fonts de complexitat
- Principalment per a sol·licitar serveis
  - Per tant, relacionades amb la comunicació
  - Trobar servidors que proporcionen el servei
    - Com pot un client contactar amb el servei sense conèixer l'ordinador concret de cadascun dels seus servidors?
    - Com s'identifiquen i localitzen els servidors?
    - Com s'identifiquen els clients en una petició?
      - Perquè el servidor puga respondre
  - Especificació de la funcionalitat dels serveis
    - Quina és l'API d'un servei?
    - Com esbrinar si la seua versió ha canviat?

1. Introducció

- En donar format a la informació transmesa...
  - Com deu el programador construir i interpretar les peticions de servei?
    - P. ex., què significa el vuitè byte de la petició?
  - Compatibilitat entre els entorns de programació utilitzats en cada extrem
    - P. ex., client escrit en Java, servidor escrit en C
- Sincronització entre client i servidor
  - Ordenació d'accions: Com sabrà un client que el servei ha acceptat les seues peticions?
  - Recepció de resultats: Com retorna un servidor els resultats? Com els obté el client?
- Seguretat
  - Com pot el programador de serveis saber quines peticions han de ser acceptades?
  - Com poden ser garantides les propietats relacionades amb la seguretat?
- Gestió de fallades
  - Com sabrà un agent quan iniciar accions de recuperació davant una fallada?
  - P. ex., com esbrinar quan un servidor ha caigut?
  - P. ex., com esbrinar si la informació transmesa s'ha perdut?
  - …

1. Introducció

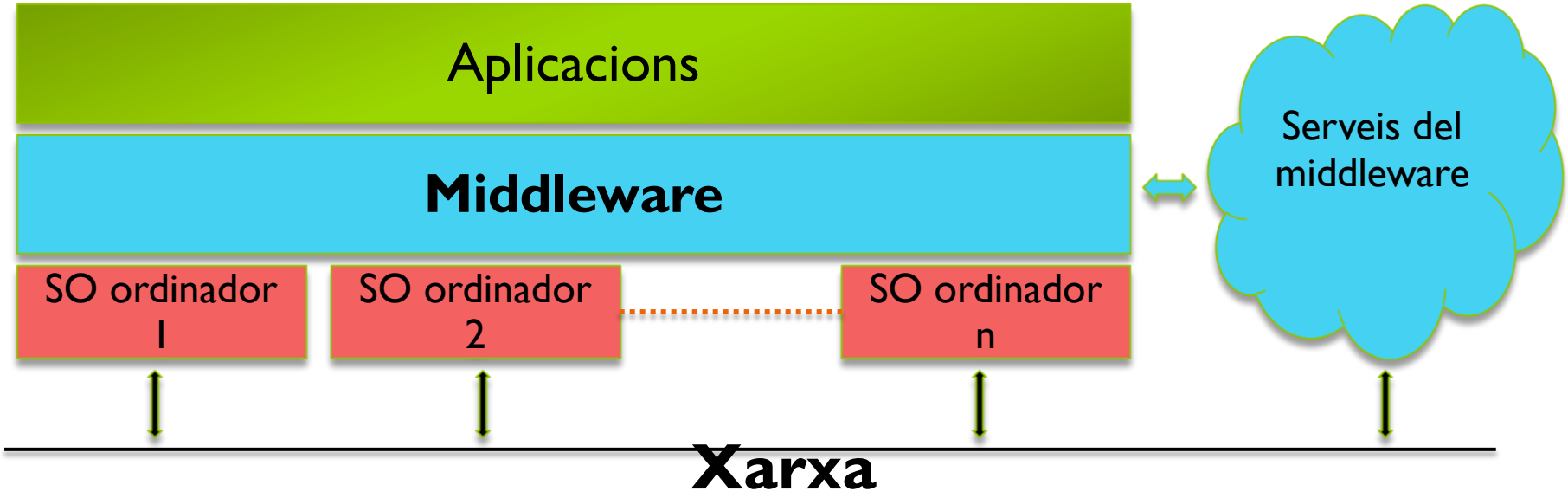
- Tècniques per a superar aquesta complexitat
  - Estàndards (de facto i de iure)
    - Introdueixen formes racionals de fer les coses
      - La pràctica trau mestre
    - Ajuden a familiaritzar els programadors amb les tècniques a utilitzar
      - No cal aprendre coses noves cada vegada
    - Faciliten la interoperabilitat
      - Millor elecció per als integradors i administradors de sistemes
    - Proporcionen funcionalitat d'alt nivell
      - Menys codi que escriure
      - Menor complexitat per a gestionar-ho
  - Middleware

Índex

1. Programació distribuïda fiable
2. Middleware
3. Comunicacions
4. Invocació de mètodes remots
5. Sistemes d'objectes distribuïts
6. Sistemes de missatgeria
7. Altres middleware
8. Conclusions

2. Middleware

- Nivell (o nivells) de programari i serveis entre les aplicacions i el nivell de comunicacions (sistema operatiu)
- Introdueix múltiples "transparències"
- Transparència
  - Reducció de la complexitat, ocultant i manejant els detalls de manera uniforme



2. Middleware: Característiques aconsellables

- Perspectiva del programador
  - Implantació senzilla
    - Conceptes clars i ben definits
    - Poca complexitat en els elements manejats
      - Evita errors en programar
  - Resultat fiable
    - Proporciona una manera de fer les coses estandarditzada, estesa, comprensible i ben definida.
  - Manteniment senzill
    - Els canvis en les seues API han de tenir un baix impacte en la necessitat de revisar els programes en desenvolupament
- Perspectiva de l'administrador
  - Fàcil instal·lació, configuració i actualització
  - Interoperabilitat: Facilitat per a interactuar amb productes de terceres parts

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

3. Sistemes de missatgeria

- Implícitament asincrònics
  - Desacoblament entre emissor i receptor
- S'envien peces concretes d'informació
  - Missatge: transmissió atòmica (tot o res)
  - Grandària arbitrària
  - Suport per a estructurar el missatge
  - Gestió de cues
    - Amb certes garanties d'ordre
- No s'imposa una visió d'estat compartit
  - Millor escalabilitat potencial
  - Major facilitat per a evitar problemes de concurrència
  - Encaixa perfectament en el model de sistema distribuït ja presentat
- Exemples
  - Estàndard establit: AMQP
    - Exemple: RabbitMQ
    - Apache ActiveMQ
  - STOMP
  - ØMQ

3. Sistemes de missatgeria: Tècniques

- Dues classes principals
  - Sistemes no persistents (*transient* / *stateless*)
    - Exigeix que el receptor estiga actiu per a transmetre el missatge
  - Sistemes persistents
    - Els missatges es mantenen en buffers: El receptor no necessita existir quan s'envie el missatge
- Entre els sistemes persistents
  - Basats en gestor (*Broker-based*)
    - Servidors concrets guarden els missatges i proporcionen garanties fortes
    - Sobrecàrrega derivada de la necessitat de mantenir els missatges en disc
    - Exemple: AMQP
  - Sense gestor (*Brokerless*)
    - Emissors i receptors mantenen els missatges
      - Normalment en memòria principal
    - Garanties de persistència més febles
      - S'assegura: desacoblament entre emissor i receptor, predisposició del receptor per a intervenir...
    - Pot prendre's com a base per a construir sistemes basats en gestor
    - Exemple: ØMQ

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
    1. Introducció
    2. Missatges
    3. API de ØMQ
    4. Sockets avançats
5. Altres middleware
6. Conclusions
7. Bibliografia

4.1. Introducció: Objectius de ØMQ

- Middleware de comunicacions simple
  - Configuració senzilla: URL per a nomenar "endpoints"
  - Ús còmode i familiar: API similar als sockets BSD
- Àmpliament disponible
  - Implementació migrable
- Suporta patrons bàsics d'interacció
  - Elimina la necessitat que cada desenvolupador "reinvente la roda"
  - Fàcil d'usar (de manera immediata)
- Rendiment
  - Sense sobrecàrregues innecessàries
  - Compromís entre fiabilitat i eficiència
- El mateix codi pot utilitzar-se per a comunicar
  - Fils en un procés
  - Processos en una màquina
  - Ordinadors en una xarxa IP
  - Només es necessiten canvis en les URL

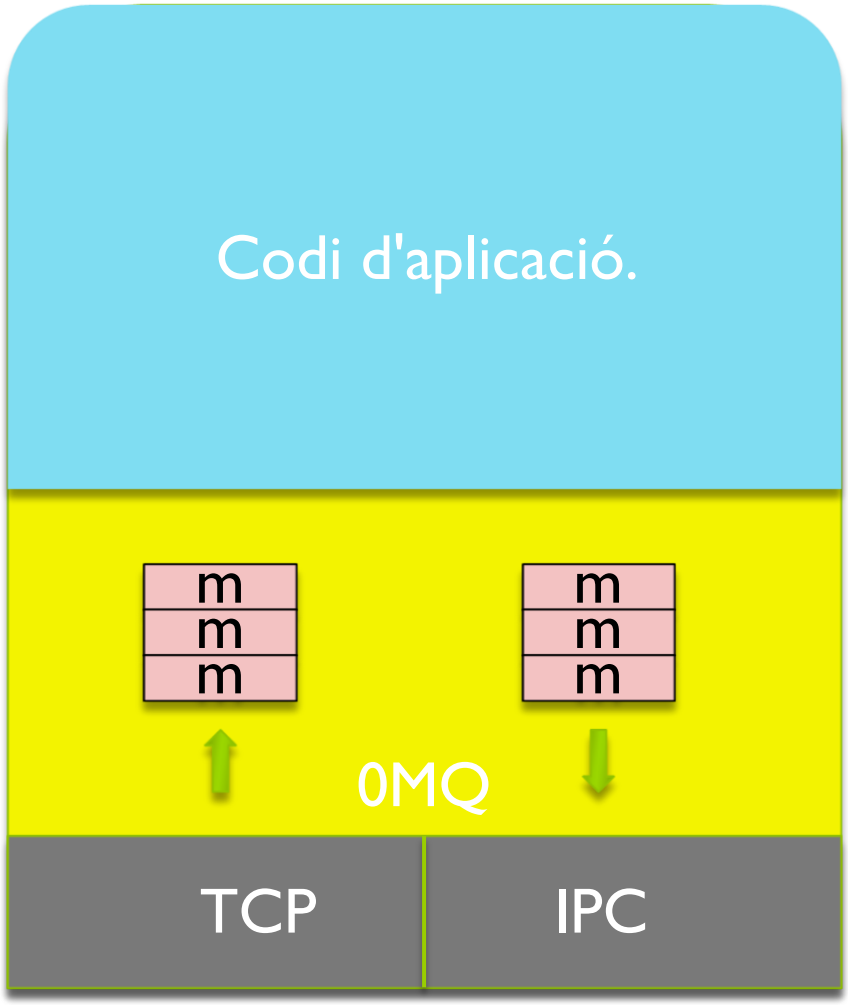
4.1. Introducció: Característiques principals

- Comunicació basada en missatges
  - Persistència feble: cues en memòria principal
- És només una biblioteca
  - No es necessita arrancar cap servidor específic (broker)
  - Implantada en C++
  - Disponible en la majoria dels sistemes operatius
    - Linux_XYZ
    - Windows
    - BSD
    - MacOS X
  - *Bindings* disponibles per a molts llenguatges i entorns de programació

4.1. Introducció: Tecnologia

- Proporciona sockets per a enviar i rebre missatges
  - send/receive, bind/connect interfície per als sockets
- Pot utilitzar aquests transports:
  - Entre processos
    - TCP/IP
    - Multicast fiable (pgm)
    - IPC (Sockets Unix)
- Transport utilitzat per a instanciar un socket
  - Fàcilment modificable mitjançant un canvi en la configuració

4.1. Introducció: Vista d'un procés ØMQ



- L'aplicació enllaça amb la biblioteca ØMQ
- ØMQ manté cues en memòria
  - En l'emissor
  - En el receptor
- ØMQ usa nivells de comunicació

4.1. Introducció: Instal·lació

- **ØMQ és una biblioteca**
  - Ha d'instal·lars-se abans d'usar-la
  - Per a utilitzar-la en NodeJS, cal usar un mòdul: **zeromq**
    - El mòdul ha d'importar-se en cada programa:
      - `const zmq = require('zeromq')`
    - En instal·lar el mòdul, aquest ja s'encarrega d'instal·lar la biblioteca

- **L'ordre necessària per a instal·lar el mòdul és:**
  - **npm install zeromq**

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
   1. Introducció
   2. Missatges
   3. API de ØMQ
   4. Sockets avançats
5. Altres middleware
6. Conclusions
7. Bibliografia

4.2. Missatges: Middleware orientat a missatges

- Els missatges és el que s'envia
  - No hi ha problemes d'empaquetat ("framing") per a l'aplicació
  - La gestió de "buffers" també està resolta
- Els missatges poden ser "multi-part" (multi-segment)
  - El suport per a estructurar els missatges resulta senzill
- Els missatges es lliuren atòmicament
  - Es lliuren totes les parts o no s'entrega res
- Tant l'enviament com la recepció són asincrònics
  - Internament, ØMQ gestiona el flux de missatges entre les cues (dels processos) i els transports
- La gestió de la connexió i reconnexió entre agents és automàtica

4.2. Missatges

- El contingut dels missatges resulta transparent per a ØMQ
- No es necessita suport per a "marshalling"
  - No cal preocupar-se per la codificació
  - Els missatges són "blobs" per a ØMQ
    - Però l'API de ØMQ suporta una serialització senzilla de cadenes en els missatges

```
zsock.send(["Açò és", "un", "missatge"])
```

6	Açò és
2	un
8	missatge

- NOTA
  - Alguns tipus de socket utilitzen el primer segment

4.2. Missatges: Conseqüències

- El programador ha de decidir com estructurar el contingut del missatge
- En molts casos, pot ser tan senzill com una cadena
- Es pot utilitzar QUALSEVOL codificació
  - Binària, per exemple
- Aproximació senzilla: missatges XML
  - S'utilitzaran parsers XML
- Aproximació una mica més senzilla: missatges JSON
- L'aproximació més senzilla
  - Utilitzar cada segment per a una peça d'informació diferent, amb la seua pròpia codificació. Exemple:
    - Segment 1: nom de la interfície invocada, en format cadena
    - Segment 2: versió de l'API de la interfície, com una cadena
    - Segment 3: nom de l'operació
    - Segment 4: primer argument (un enter)
    - Segment …

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
   1. Introducció
   2. Missatges
   3. API de ØMQ
   4. Sockets avançats
5. Altres middleware
6. Conclusions
7. Bibliografia

4.3. API ØMQ

1. Sockets
   - Enviament i recepció utilitzen sockets
   - Diversos tipus de sockets
   - Operacions bind/connect
2. Patrons de comunicació
   - Suportats per tipus específics de socket

4.3.1. Sockets ØMQ

- La creació d'un socket és senzilla:

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

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

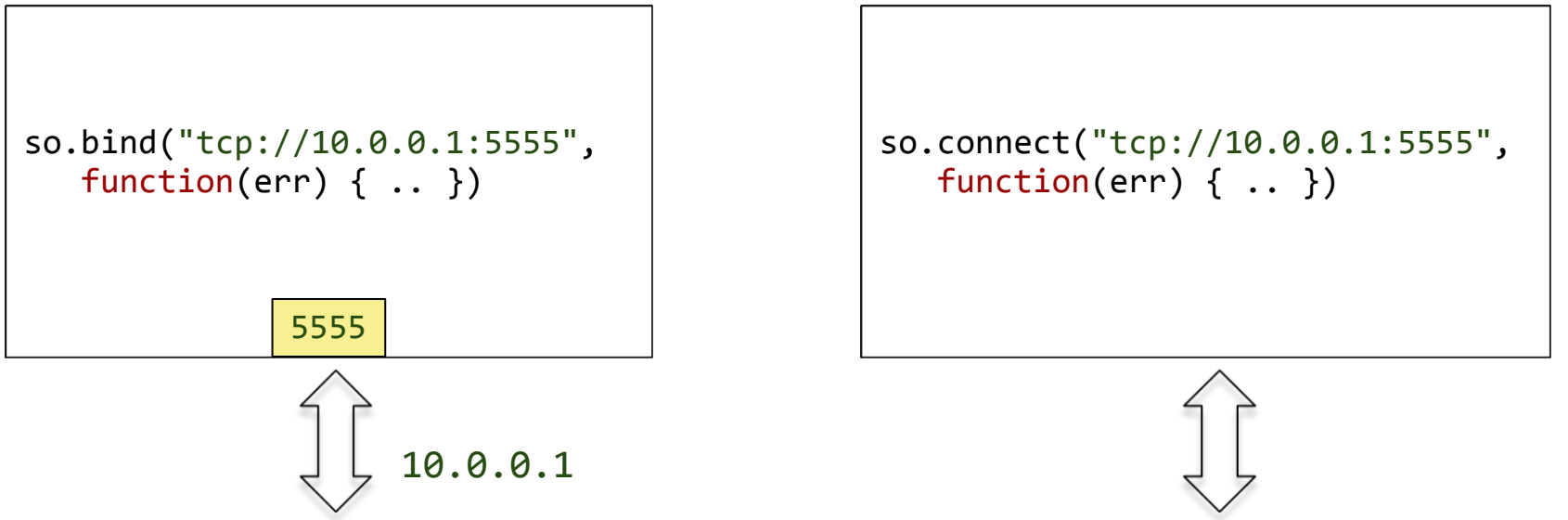
- On <TIPUS SOCKET> serà un dels següents

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

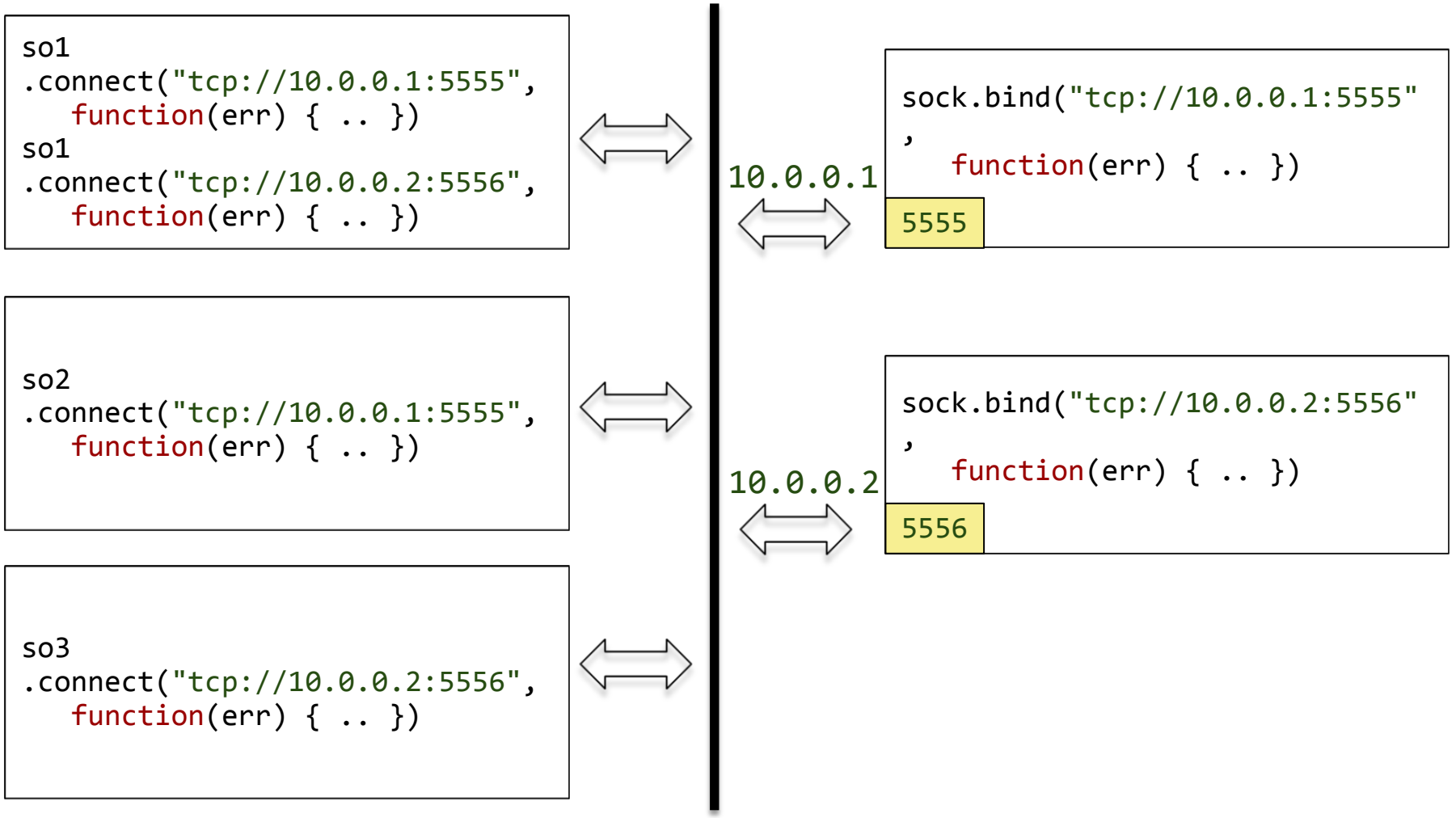
- Quins tipus utilitzar dependrà dels patrons de connexió en els quals intervinga

4.3.1. Sockets: Establint vies de comunicació

- Un procés realitza un **bind**
- Altres processos fan un **connect**
- Quan acaben, **close**
- **bind/connect** estan desacoblats: no hi ha requisits sobre la seua ordenació



4.3.1. Sockets: Múltiples connexions són possibles



4.3.1. Sockets: Connexions i cues

- Els sockets tenen cues de missatges associades
  - D'entrada (recepció), per a mantenir els missatges que hagen arribat
    - Generen l'esdeveniment "message" quan mantenen algun missatge
  - D'eixida (enviament), mantenint els missatges a enviar a altres agents
    - On es guarden els missatges enviats per l'aplicació
- Els sockets "router" mantenen un parell de cues (entrada/eixida) per agent connectat
  - La resta dels sockets no distingeixen entre agents
  - Els sockets "pub" queden fora d'aquesta discussió
- Els sockets "pull" i "sub" solament mantenen una cua d'entrada
- Els sockets "push" i "pub" solament mantenen una cua d'eixida

4.3.1. Sockets: bind / connect

- Quan realitzar un **bind** i quan un **connect**?
  - En la majoria dels casos no importa: gestionat en la configuració
- Observacions
  - Tots els agents coincidiran en algun "endpoint"
  - Els "endpoints" es referencien mitjançant les seues URL
  - En el transport TCP
    - L'adreça IP ha de pertànyer a una de les interfícies del socket (bind)
      - bind: El socket solament necessita una configuració IP local (o cap)
        - No necessita conèixer on estan els altres agents
    - El socket que realitze un "connect" necessita conèixer l'adreça IP del socket que realitze un "bind"

4.3.1. Sockets: Transports: TCP

- URL: `tcp://<adreça>:<port>`
- Tres maneres d'especificar l'adreça

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

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

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

- *: **bind** sobre totes les interfícies
- "eth0": **bind** sobre totes les adreces associades a la interfície "eth0"

4.3.1. Sockets: Transports: IPC

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

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

- Es necessita permís rw (lectura i escriptura) sobre el socket en <ruta-del-socket>

4.3.1. Sockets: Enviament de missatges

- Els segments poden extraure's d'un vector en una mateixa crida

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

- Els segments han de ser **buffers** o **cadenes**.
  - Les cadenes es converteixen en buffers, utilitzant codificació UTF8
    - El que no siga cadena es converteix primer a cadena

4.3.1. Sockets: Recepció

- Basat en esdeveniments "message" del socket
  - Els **arguments** del manejador contenen els segments del missatge
  - NOTA: Els segments són buffers binaris

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

- Per a un nombre variable de segments, usar "**arguments**" directament…

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

- … o convertir abans en vector

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

4.3.1 Sockets: Opcions

- N'hi ha moltes.
- Dos importants: **identity**, i **subscribe**

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

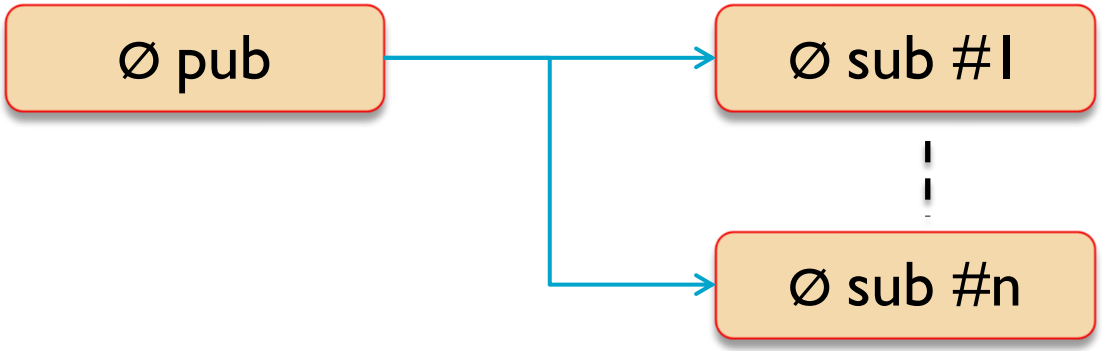
- **identity** és convenient a l'hora de connectar amb sockets "router"
  - Fixa l'identificador de l'agent que es connecte al "router"
  - Cal donar-li valor abans de cridar el mètode connect()
- **subscribe**, utilitzat per sockets "sub"
  - Fixa el filtre de prefixos aplicat al socket "pub"

4.3.2. Patrons bàsics

- **Req**uest/**Rep**ly (sincrònic)

- Push-pull

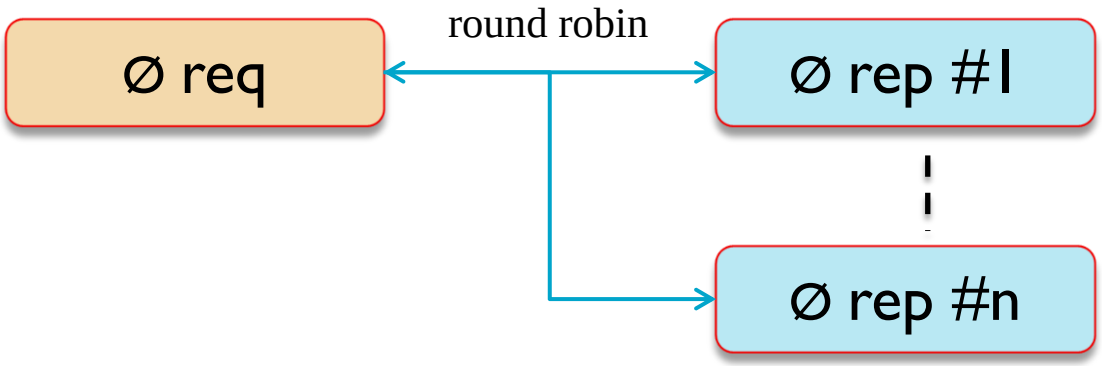
- Pub-Sub



4.3.2. Patrons bàsics: **req**uest/**rep**ly

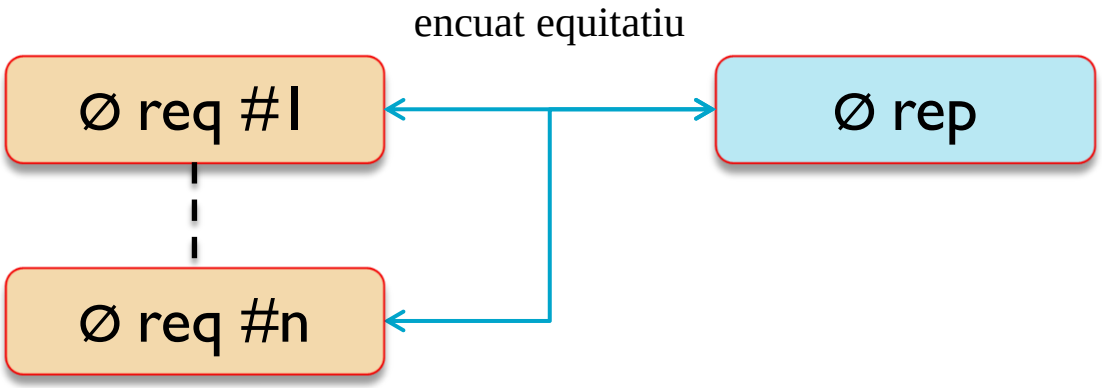
- Implantat mitjançant sockets **req** en el client
  - sockets **rep** en el servidor
- Cada missatge enviat via **req** necessita associar-se a una contestació des del socket **rep** del servidor
- Patró de comunicació sincrònic
  - Tots els parells petició/resposta estan totalment ordenats
  - Els "endpoints" poden reaccionar asincrònicament
- Quan s'ha enviat un missatge a través d'un socket **req**, un altre enviament posterior per aquest socket serà encuat localment
  - Fins que el missatge de resposta siga rebut
    - Llavors el missatge encuat s'enviarà
- Quan s'ha rebut un missatge a través d'un socket **rep**, una altra recepció posterior per aquest socket serà encuada localment
  - Fins que la resposta a la petició anterior s'haja enviat
    - Llavors el missatge encuat s'entregarà a l'aplicació servidora
- Cada enviament per un socket **rep** queda bloquejat mentre no arribe la seua petició associada pel mateix socket
  - El bloqueig acaba tan prompte com arribe la petició corresponent

4.3.2. Patrons bàsics: **req**uest/**rep**ly amb distribució



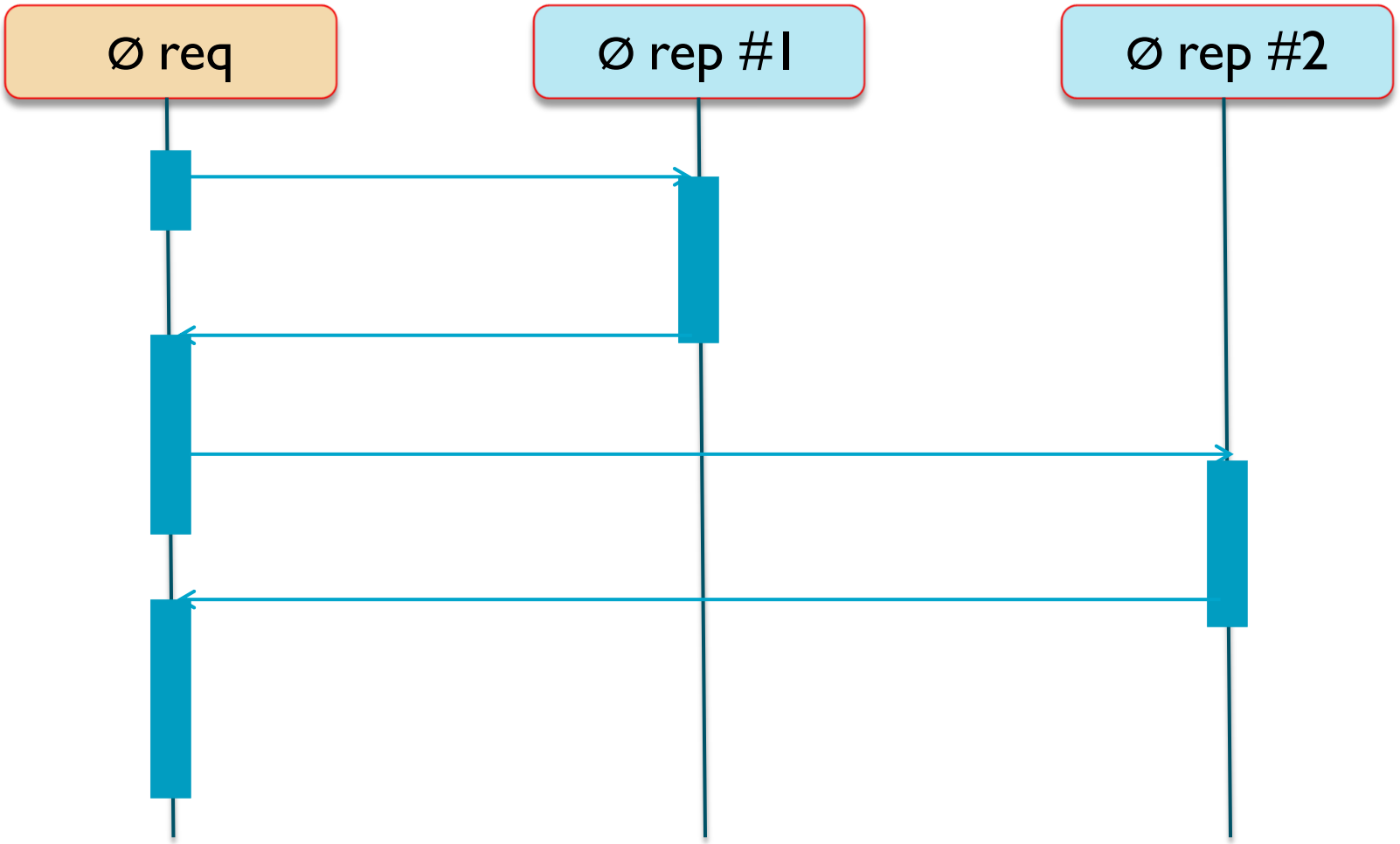
- Quan un **req** connecta amb més d'un **rep**, cada missatge de petició s'envia a **un rep** diferent
  - Se segueix una política "round-robin"
- L'operació continua sent sincrònica:
  - ØMQ no envia noves peticions fins que cada resposta siga rebuda
  - No hi ha paral·lelització de peticions

4.3.2. Patrons bàsics: múltiples peticionaris

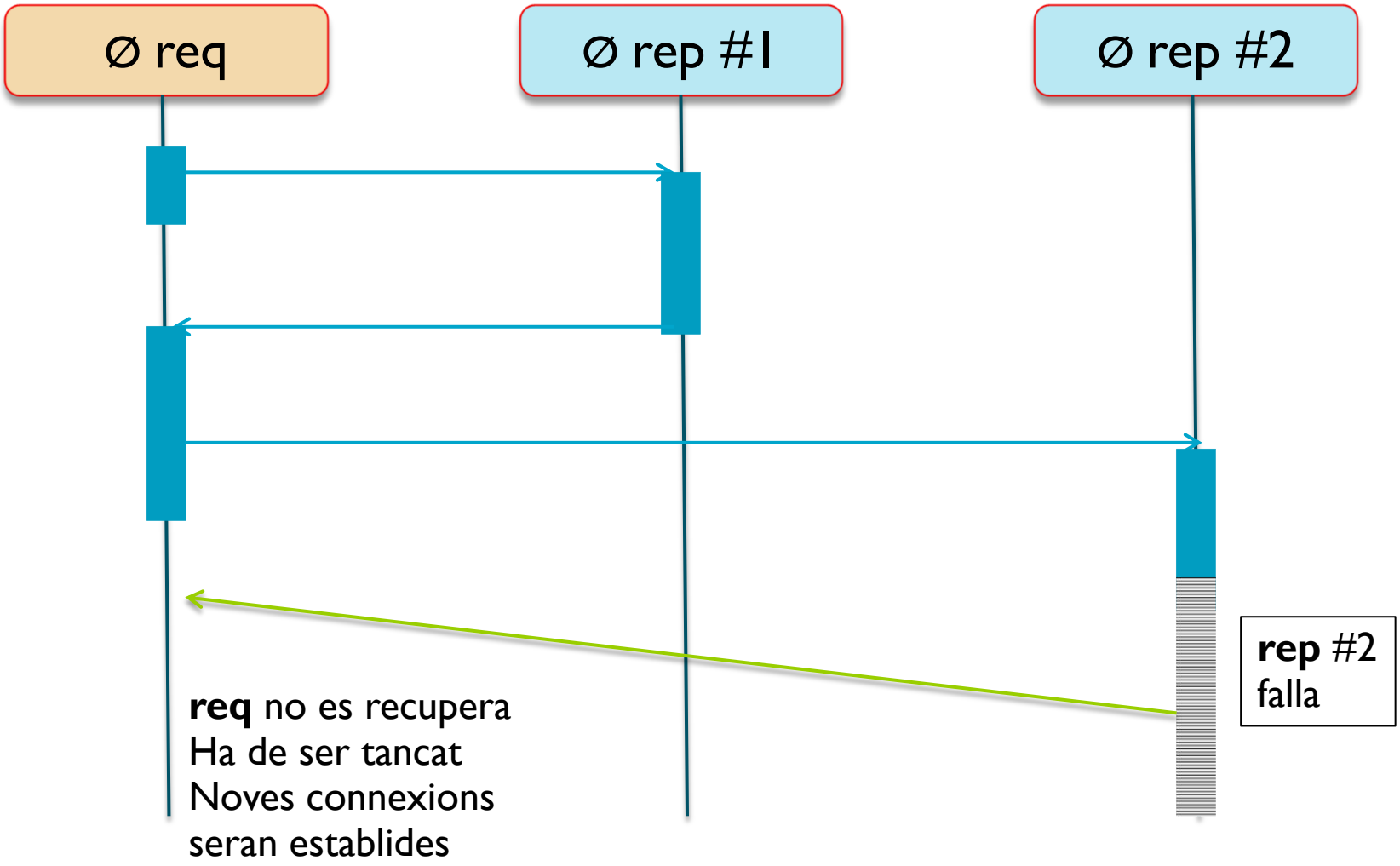


- Configuració típica per a un servidor
- El socket **rep** gestiona els missatges d'entrada amb una cua
  - Cap socket **req** tindrà inanició

4.3.2. Patrons bàsics: Seqüència petició/resposta



4.3.2. Patrons bàsics: Fallades petició/resposta



4.3.2. Patrons: req/rep bàsic

```
const zmq = require('zmq')
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('zmq')
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. Patrons bàsics: req/rep, dos servidors

```
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('zmq')
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('zmq')
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')
})
```

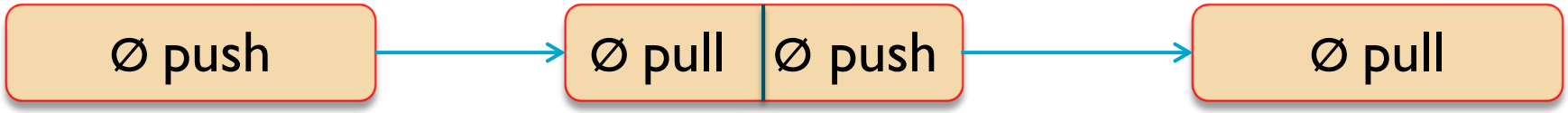
4.3.2. Patrons: req/rep, estructura dels missatges

- Els missatges tenen un primer segment buit
- És el "delimitador"
- El socket **req** ho afig, sense que intervinga l'aplicació.
- El socket **rep** l'elimina abans de passar-ho a l'aplicació.
  - Però l'afig de nou en la contestació
- El socket **req** l'eliminarà de la contestació

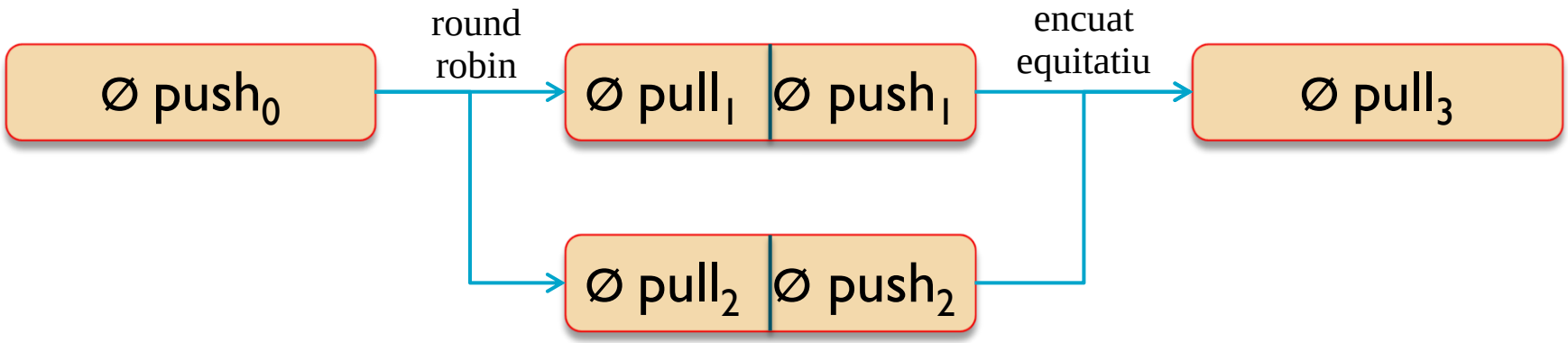
0	""
6	"açò és"
3	"una"
7	"petició"

4.3.2. Patrons bàsics: push/pull

- Distribució de dades unidireccional
- L'emissor no espera cap resposta
  - Els missatges no esperen respostes: enviaments concurrents



- S'accepten múltiples connexions
  - P.ex., organització típica map-reduce:



4.3.2: Patrons: exemple push/pull, productor/consumidors

```
const zmq = require("zeromq")
const producer = zmq.socket("push")
let count = 0

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

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

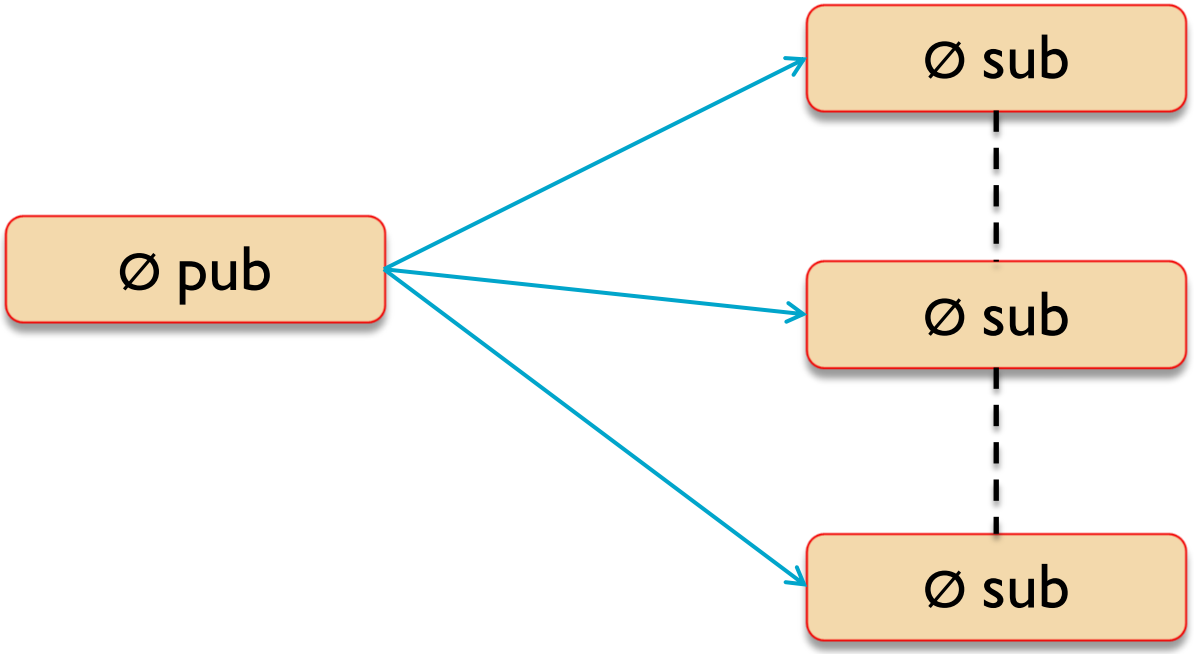
```
const zmq = require("zeromq")
const consumer = zmq.socket("pull")

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

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

4.3.2. Patrons bàsics: Publish/Subscribe (pub/sub)

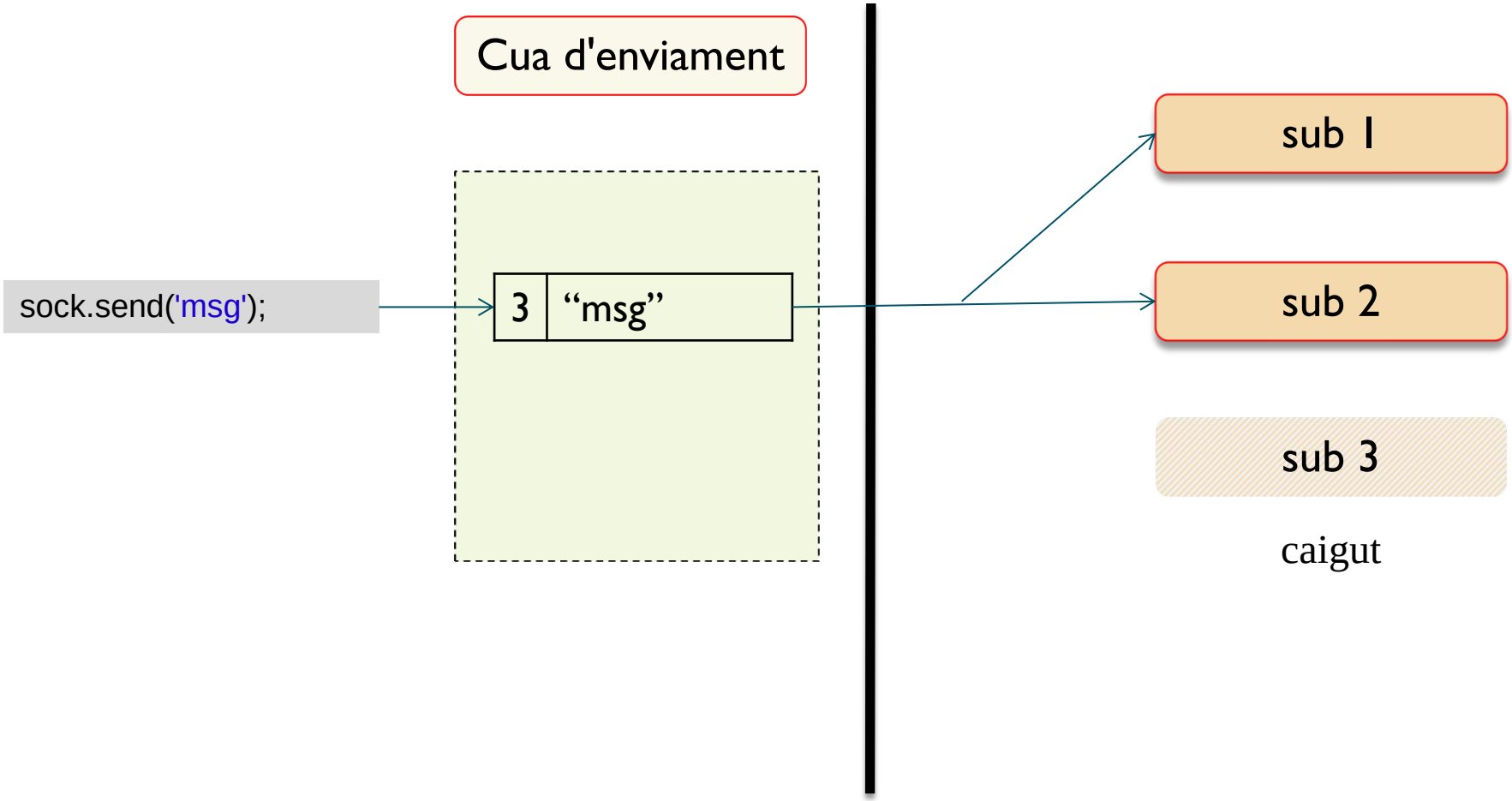
- Aquest patró implanta la difusió de missatges...



- ... amb una condició: els receptors poden decidir que se subscriuen solament a certs missatges
  - Llavors és un multienviament

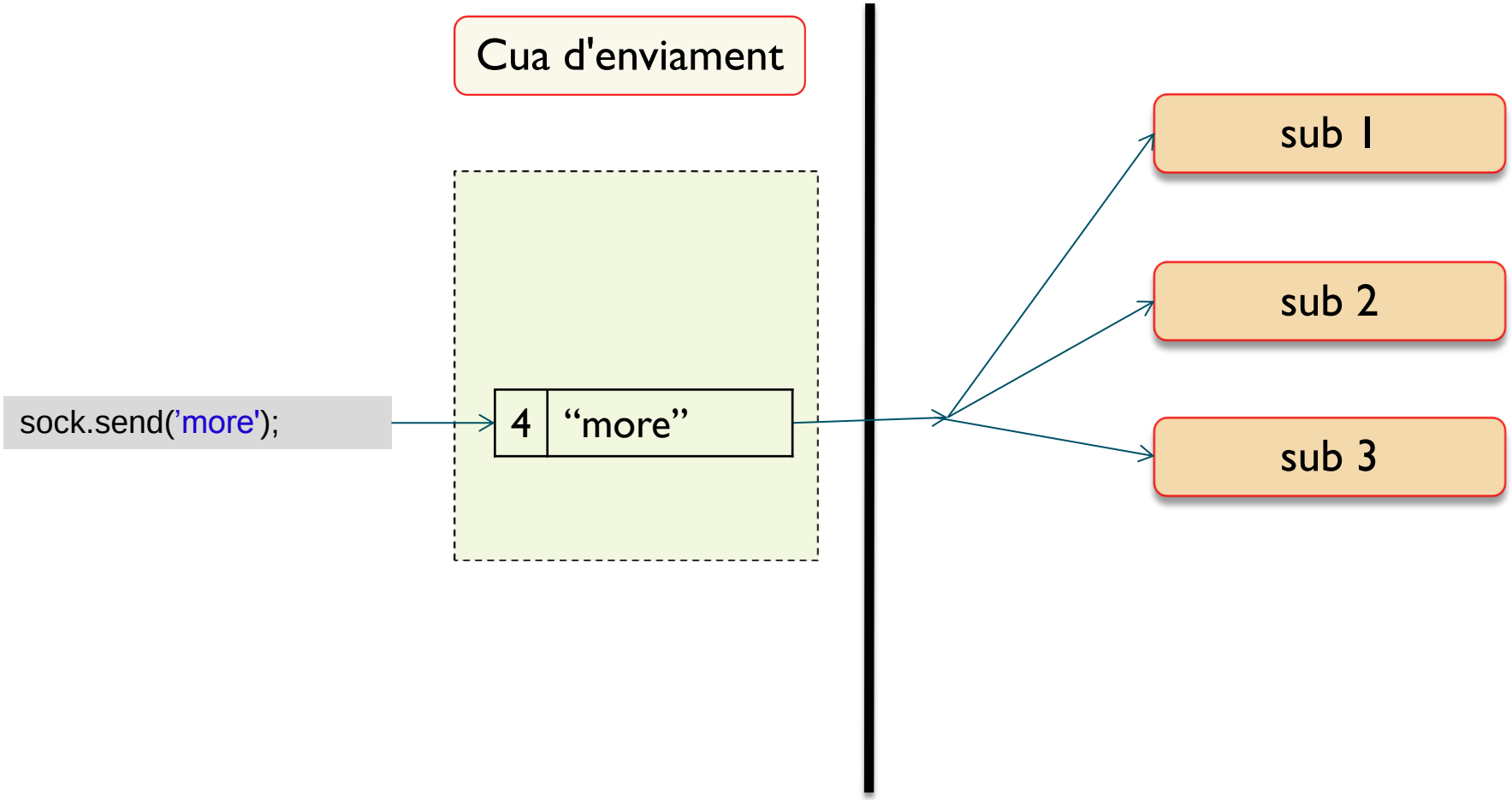
4.3.2. Patrons bàsics: pub/sub

- Els missatges són enviats a tots els agents disponibles i connectats



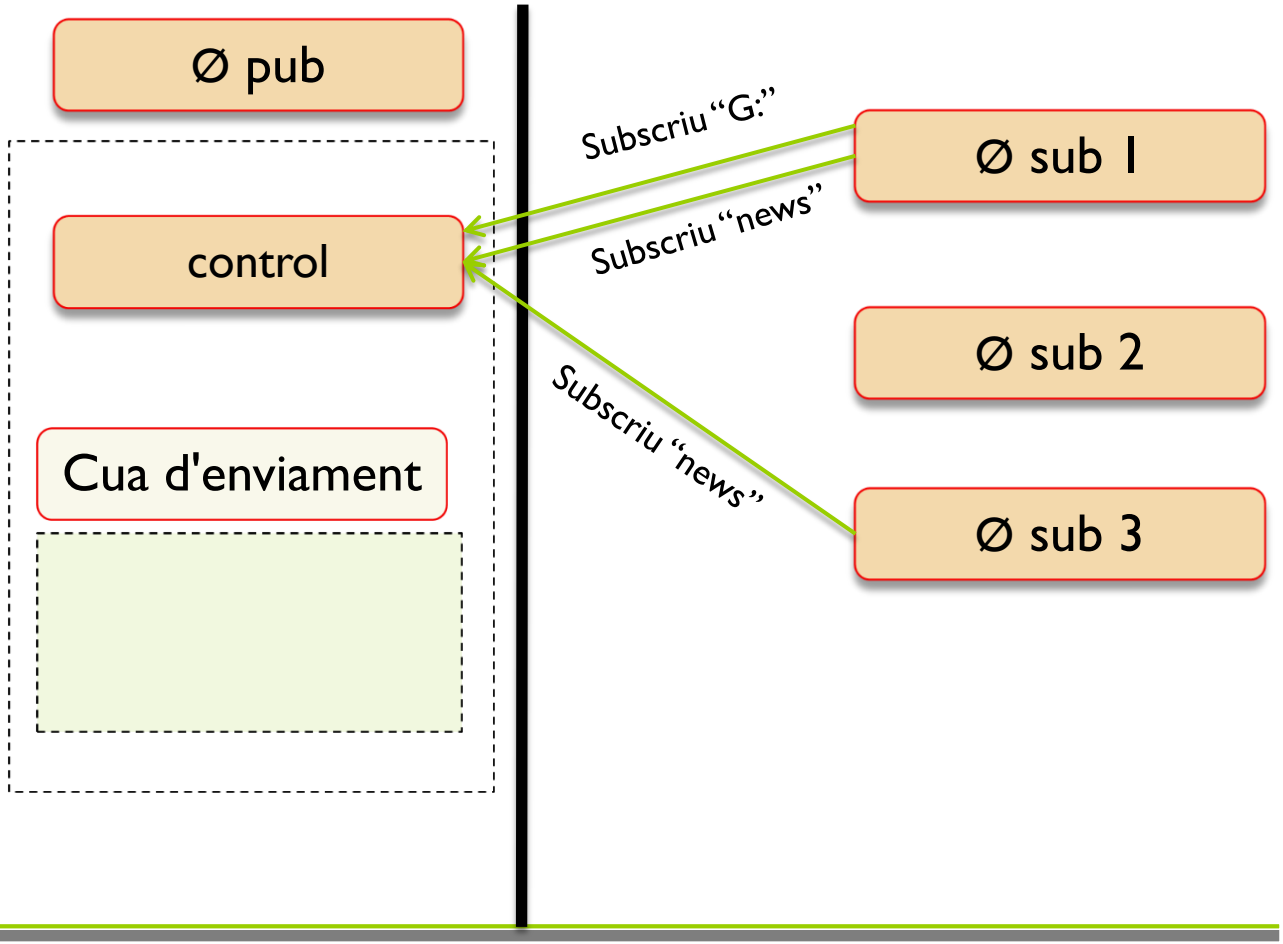
4.3.2. Patrons bàsics: pub/sub

- Els missatges són enviats a tots els agents disponibles i connectats



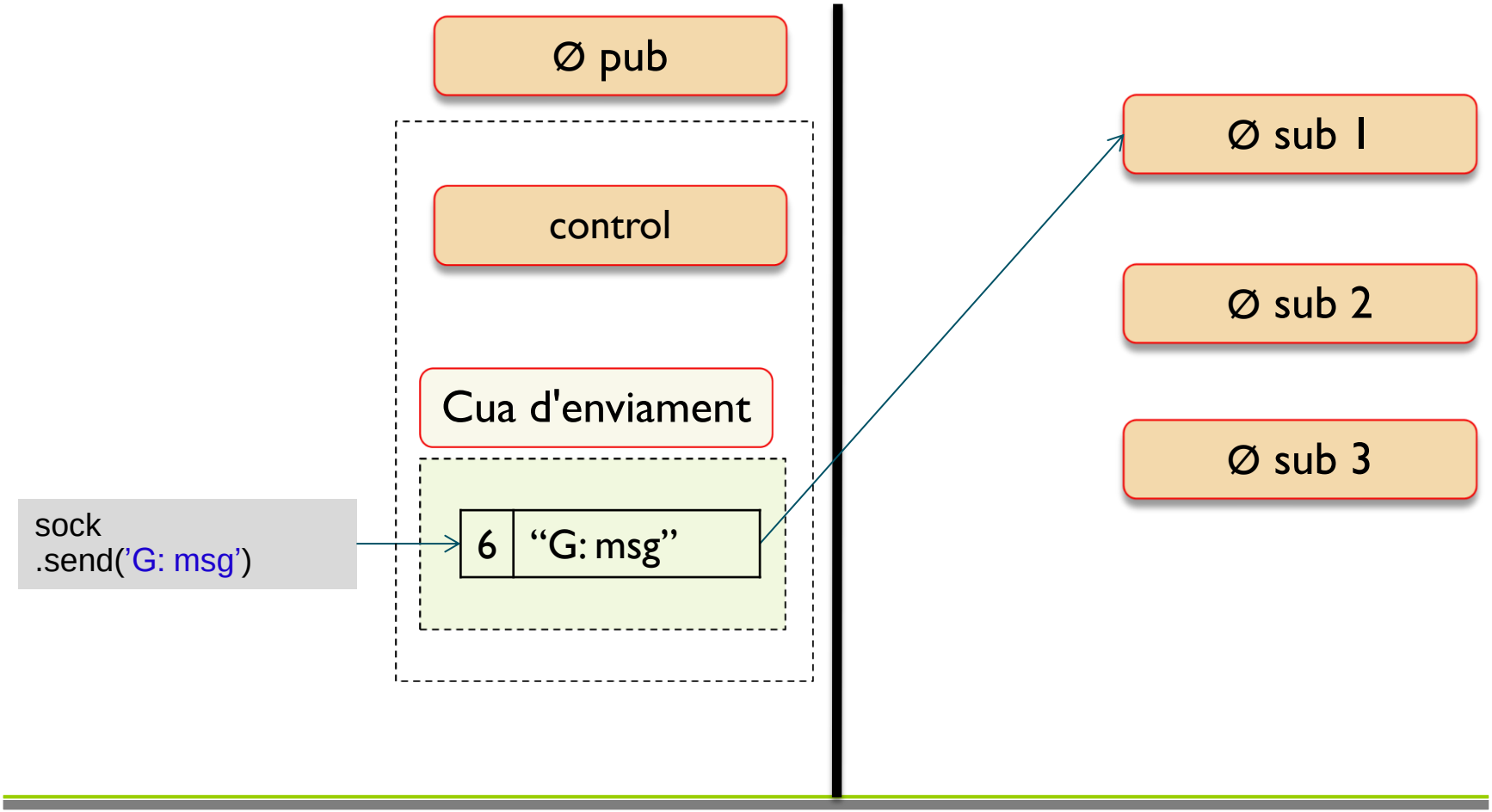
4.3.2. Patrons bàsics: pub/sub: Subscripció/filtrat

- Els subscriptors poden especificar filtres, com a prefixos dels missatges
  - Poden especificar diversos prefixos



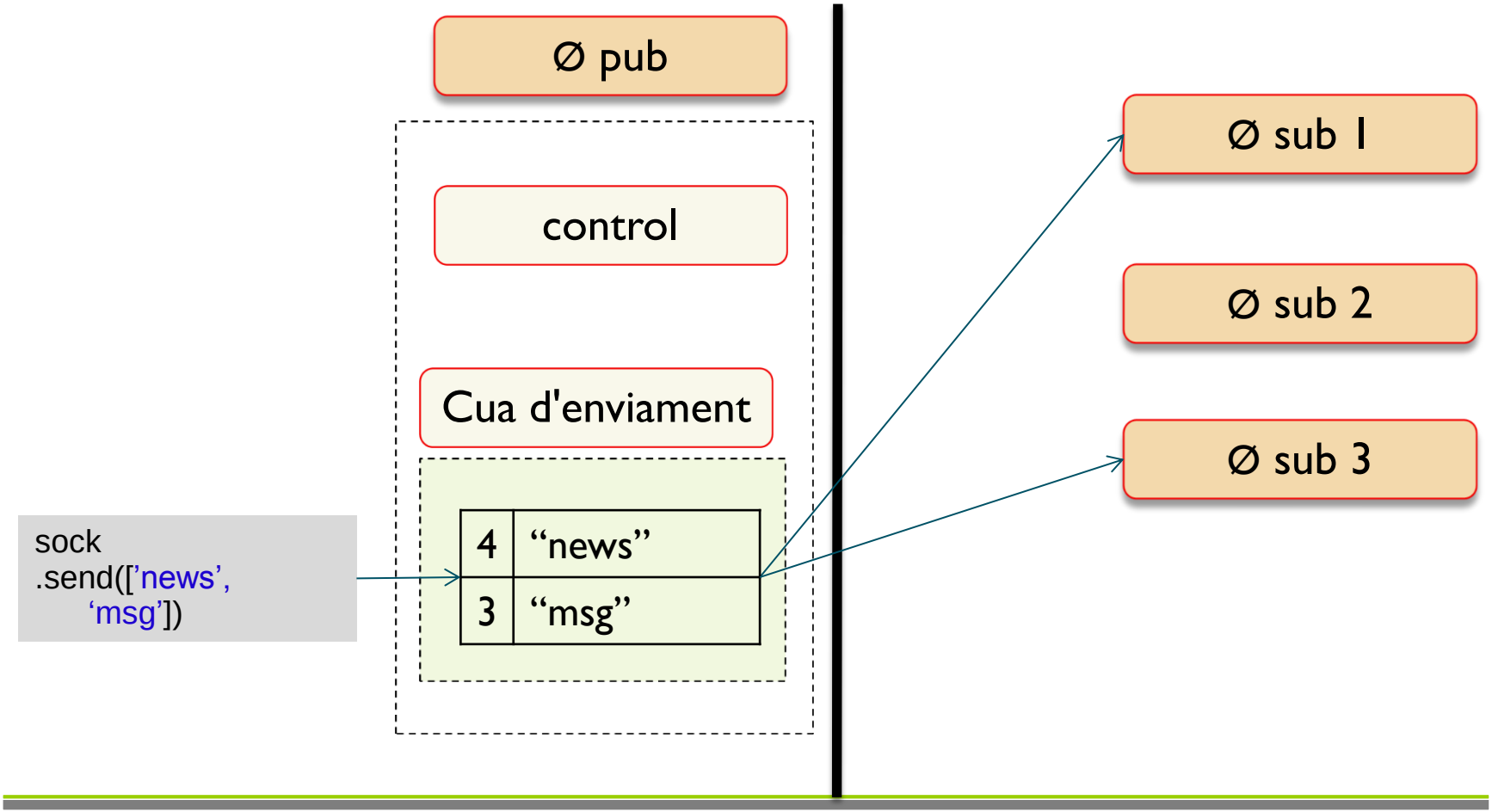
4.3.2. Patrons bàsics: pub/sub: Subscripció/filtrat

- Els subscriptors poden especificar filtres, com a prefixos dels missatges
  - Rebran solament els missatges amb aquests prefixos



4.3.2. Patrons bàsics: pub/sub: Subscripció/filtrat

- Els subscriptors poden especificar filtres, com a prefixos dels missatges
  - Rebran solament els missatges amb aquests prefixos



4.3.2. Patrons bàsics. Exemple 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)
})
```

Els missatges més antics podrien perdre's si el subscriptor comença tard

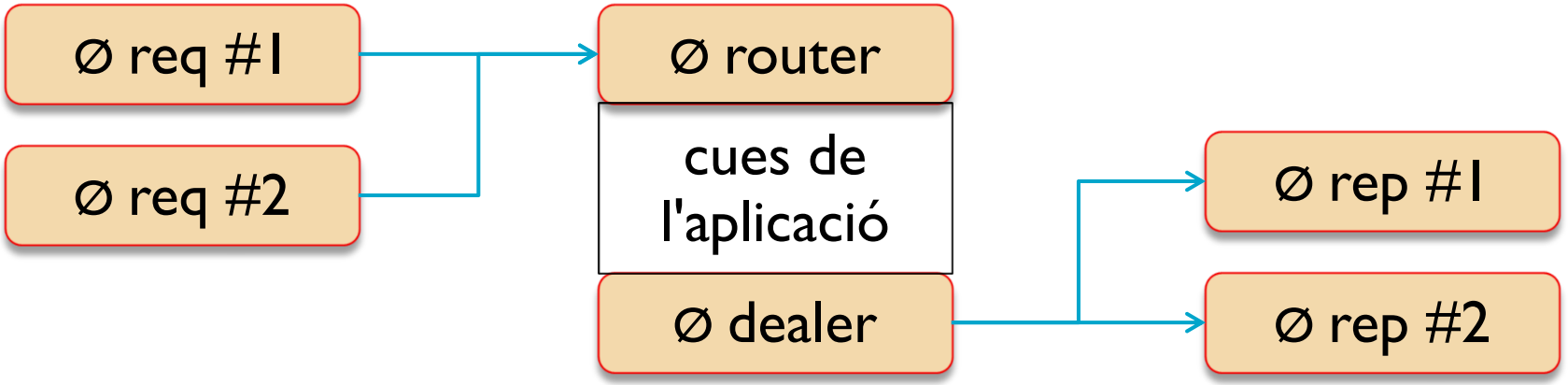
Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
   1. Introducció
   2. Missatges
   3. API de ØMQ
   4. Sockets avançats
5. Altres middleware
6. Conclusions
7. Bibliografia

4.4. Tipus de "sockets" avançats

1. Dealer
   - Similar a **req**, però asincrònic
2. Router
   - Similar a **rep**, però asincrònic i amb capacitat per a distingir entre agents (per a encaminar les respostes)

- Normalment s'implanten junts en un mateix agent

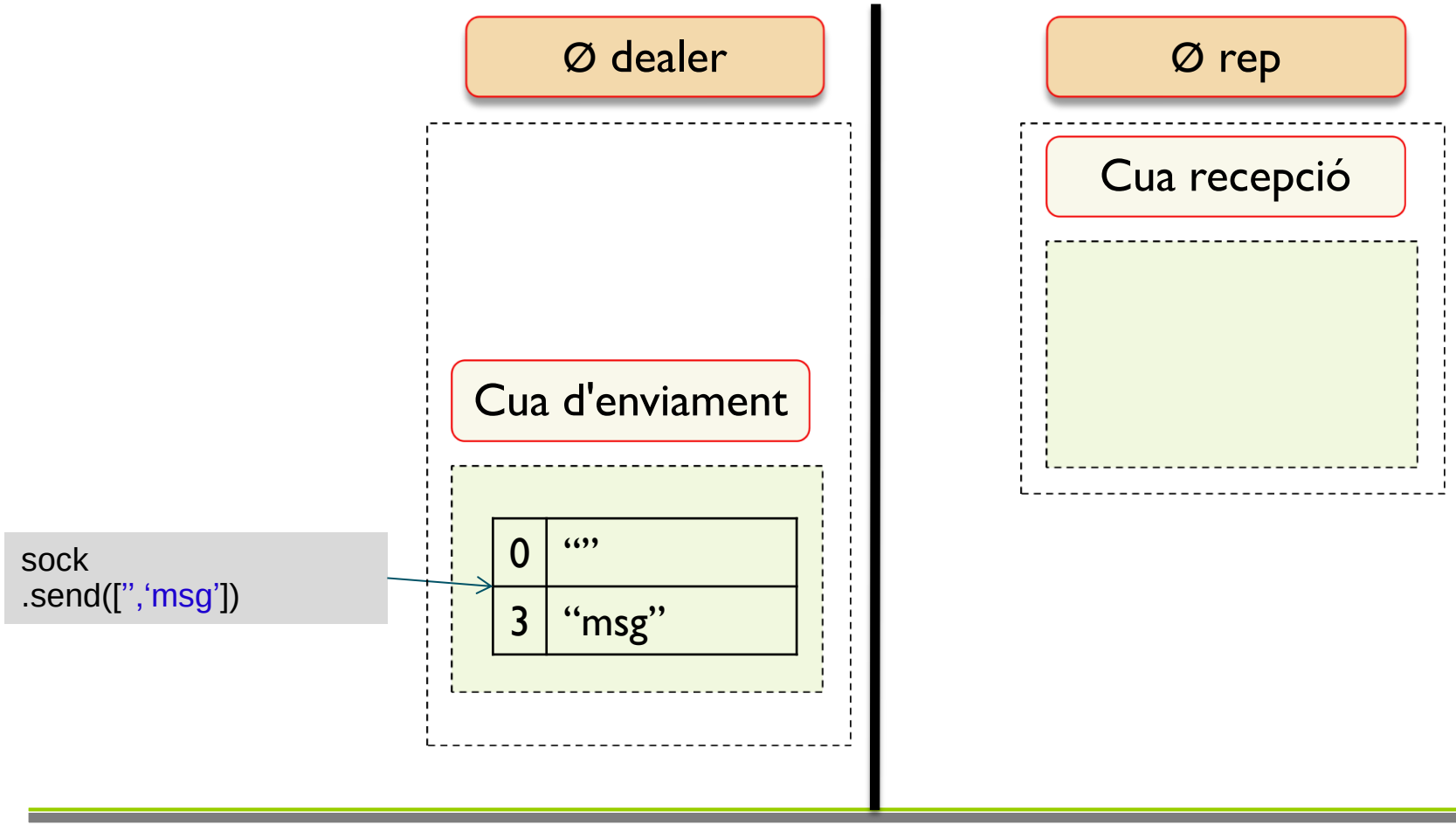


4.4.1. Sockets dealer

- És un socket asincrònic de propòsit general
- Usat freqüentment com socket **req** asincrònic
  - No es bloqueja per fallades en els agents
  - PERÒ, ha de construir un missatge de petició adequat
    - Amb segment buit (delimitador) abans del cos real del missatge
    - Pot situar després del delimitador qualsevol nombre de segments

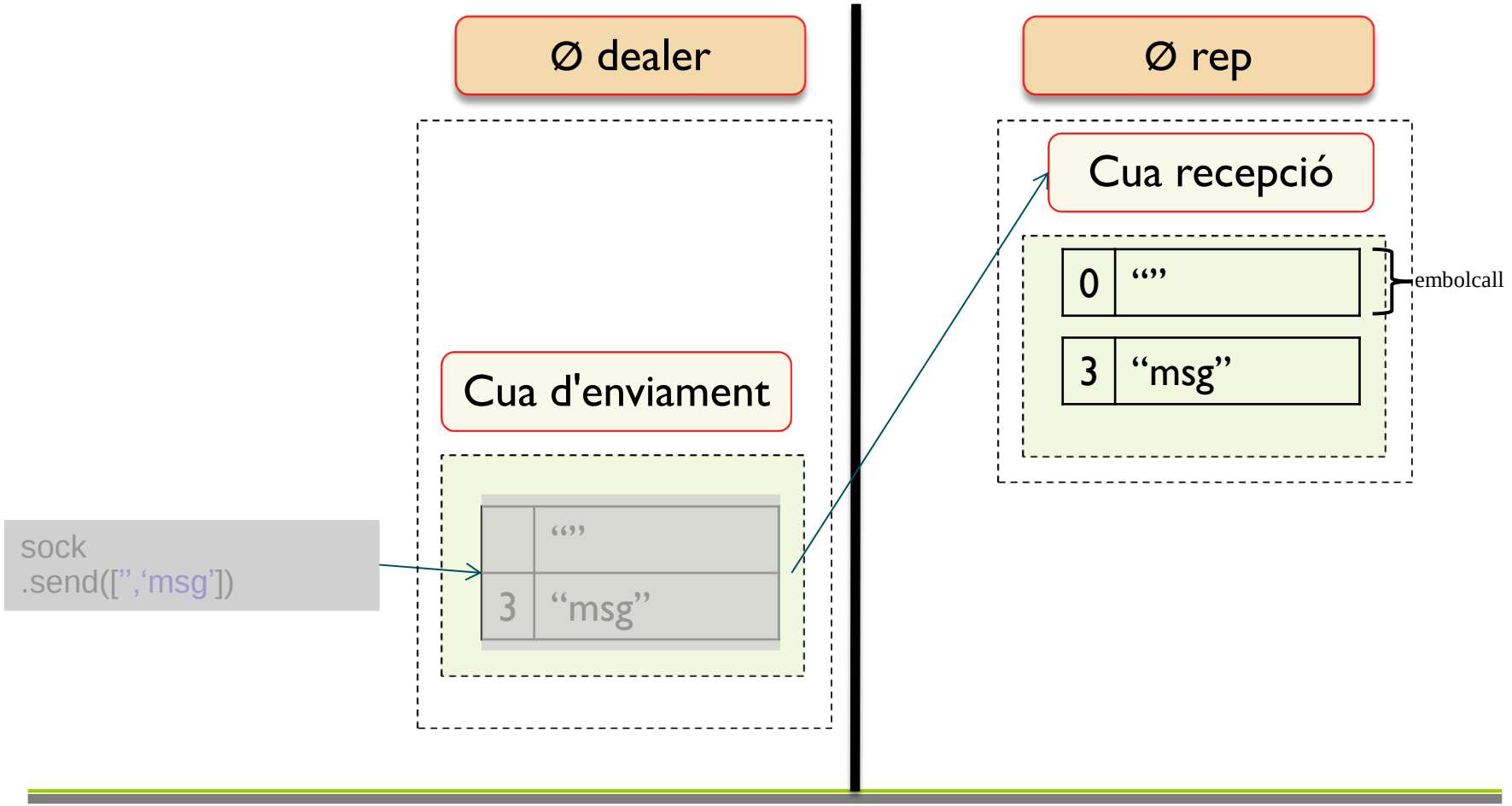
4.4.1. Sockets dealer: gestió de peticions i respostes

- El delimitador ha de ser afegit (com a capçalera) per a comunicar-se amb un **rep**:



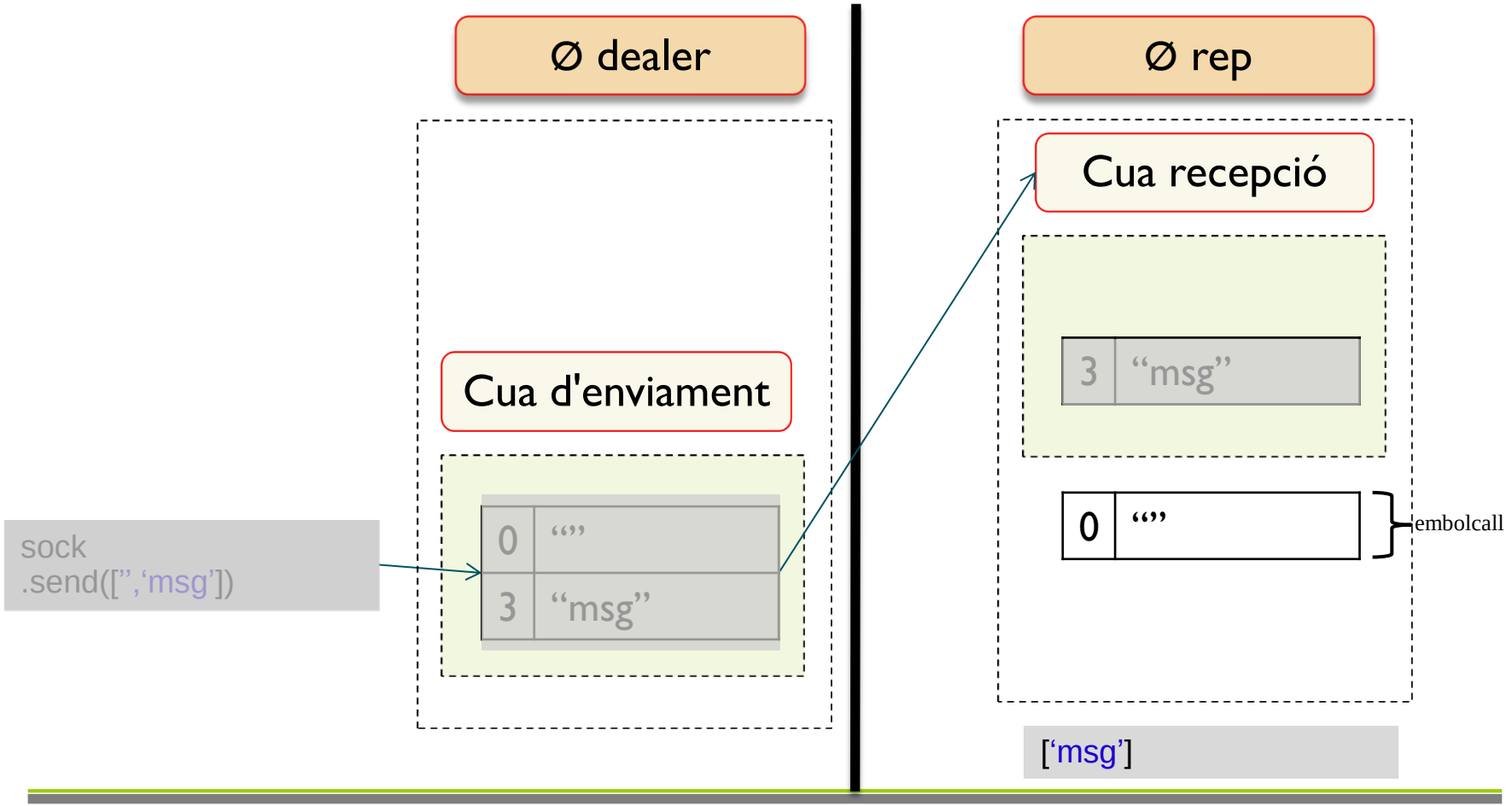
4.4.1. Sockets dealer: gestió de peticions i respostes

- Quan es reba, el socket **rep** lleva l'"embolcall"



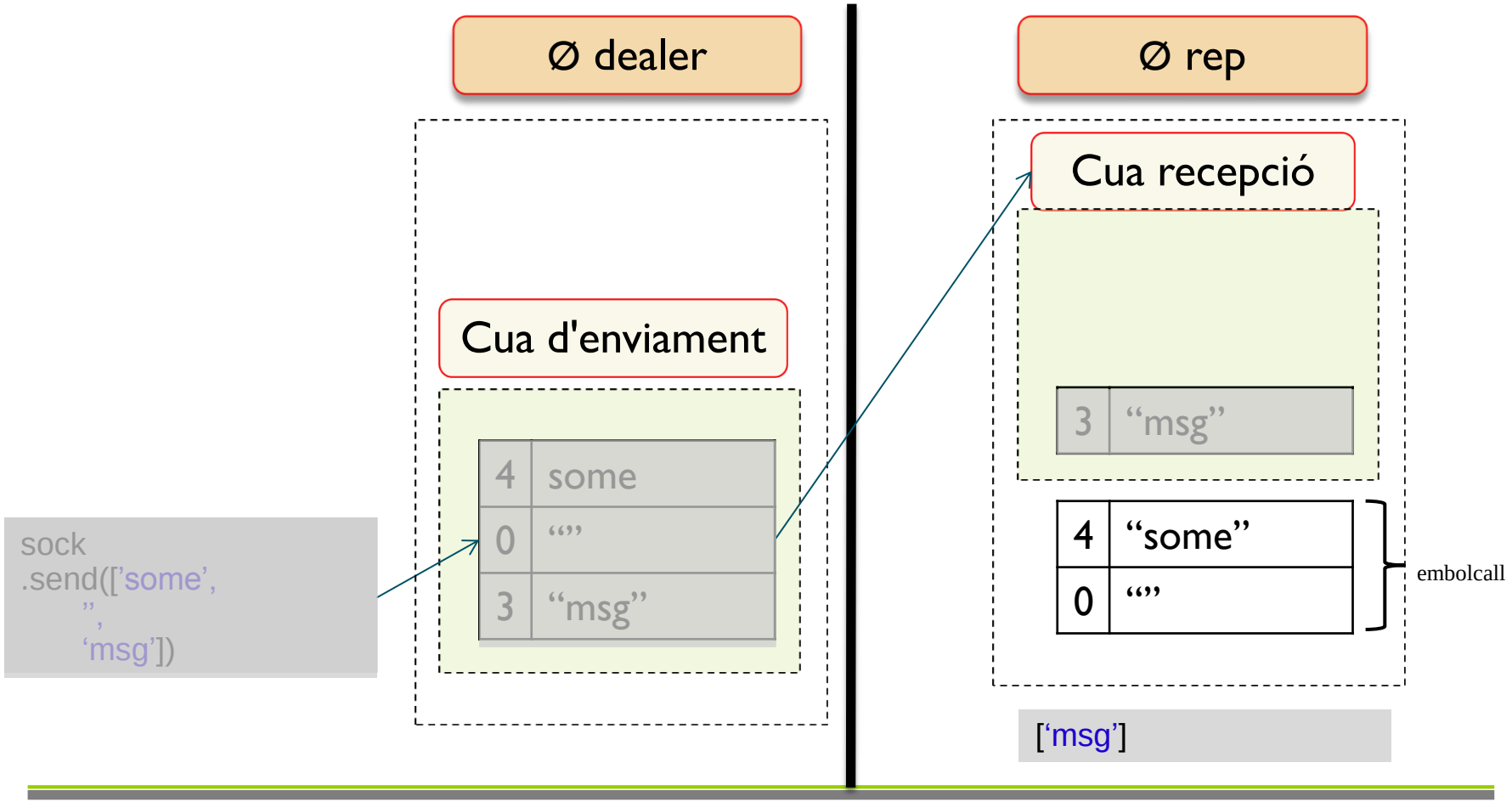
4.4.1. Sockets dealer: gestió de peticions i respostes

- Quan es reba, el socket **rep** lleva l'"embolcall"
  - L'aplicació solament rep la resta del missatge



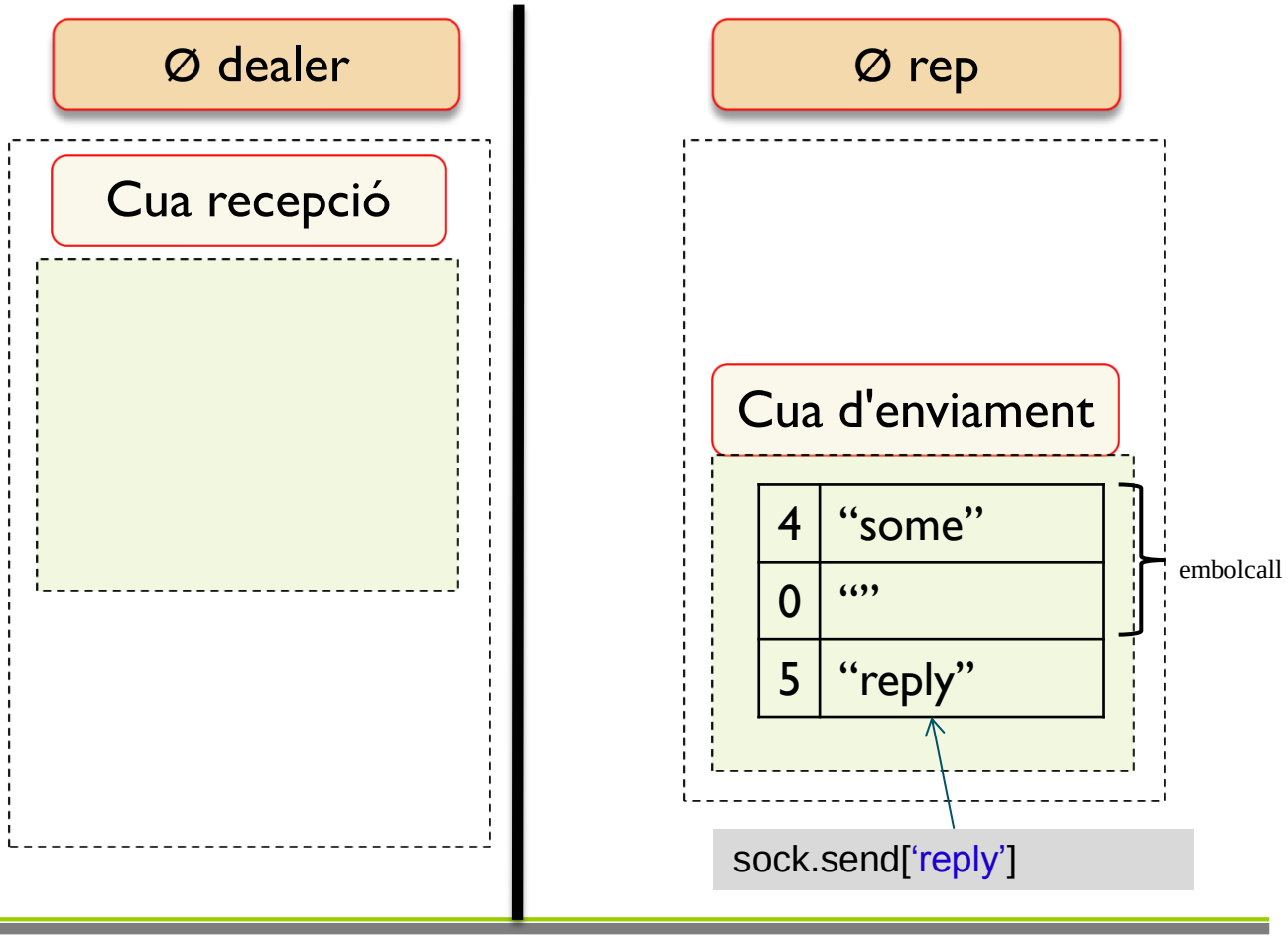
4.4.1. Sockets dealer: embolcall

- L'embolcall és més general: Tots els segments fins al primer delimitador
  - L'embolcall és guardat pel **rep**...



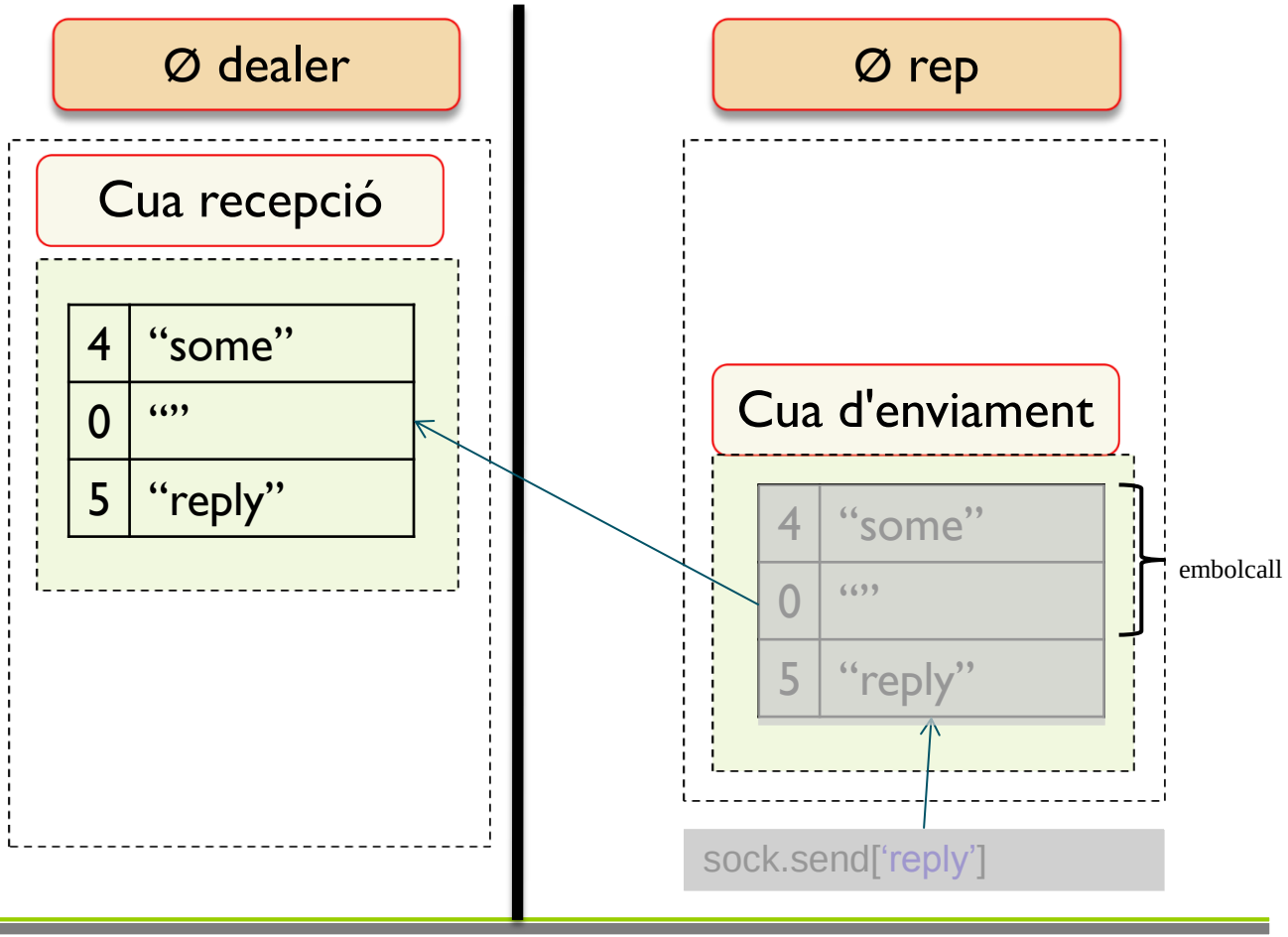
4.4.1. Sockets dealer: embolcall

- L'embolcall és més general: Tots els segments fins al primer delimitador
  - L'embolcall és guardat pel **rep**... i reinserit en la resposta



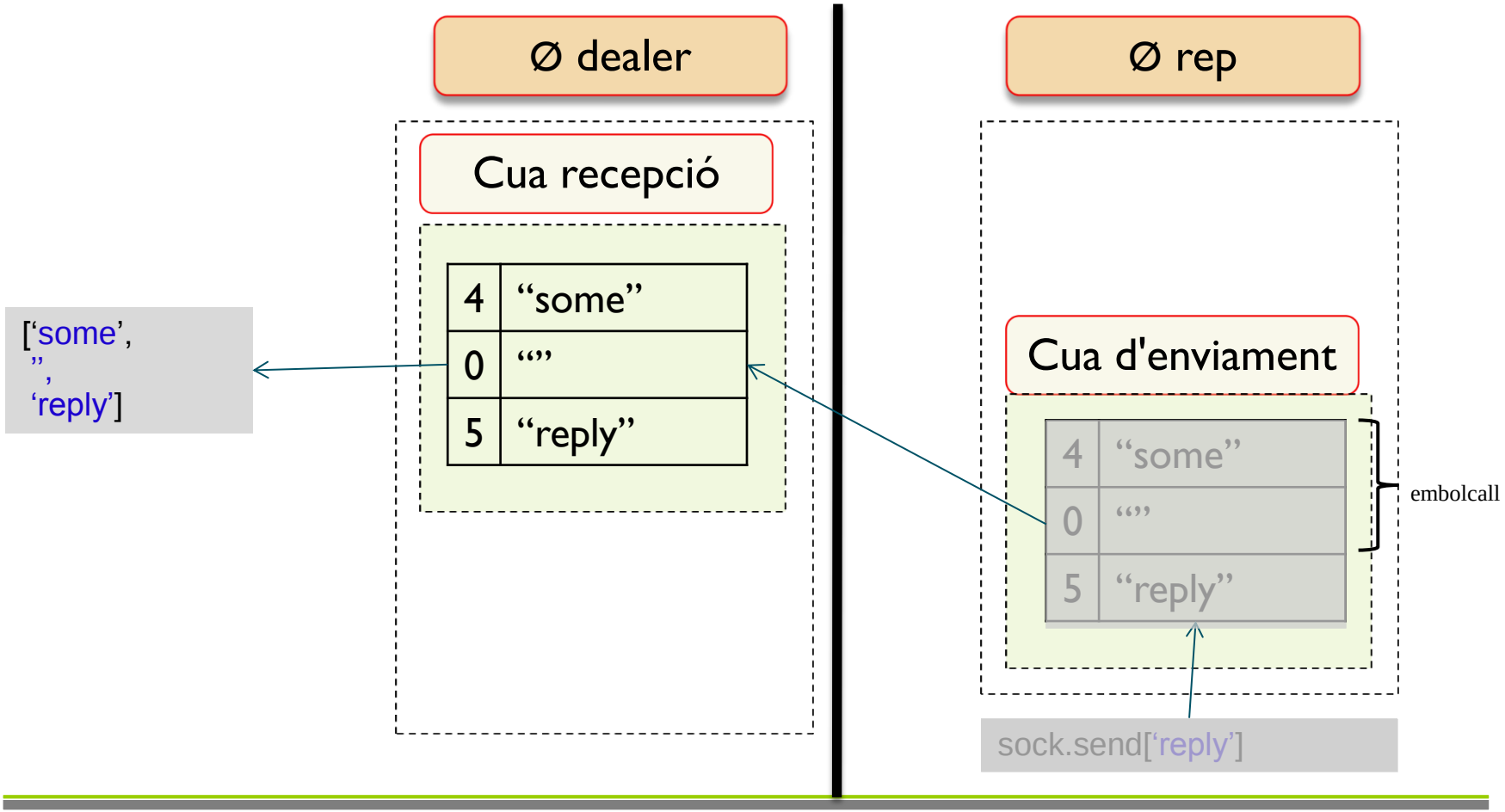
4.4.1. Sockets dealer: embolcall

- L'embolcall és més general: Tots els segments fins al primer delimitador
  - El missatge generat s'envia com a resposta



4.4.1. Sockets dealer: embolcall

- L'embolcall és més general: Tots els segments fins al primer delimitador
  - I l'aplicació dealer obté tot això



4.4.1. Sockets dealer: exemple de codi

```
const zmq  = require('zeromq')
const dealer = zmq.socket('dealer')
var msg = ["", "Hello ", 0]
const 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)
})
```

```
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

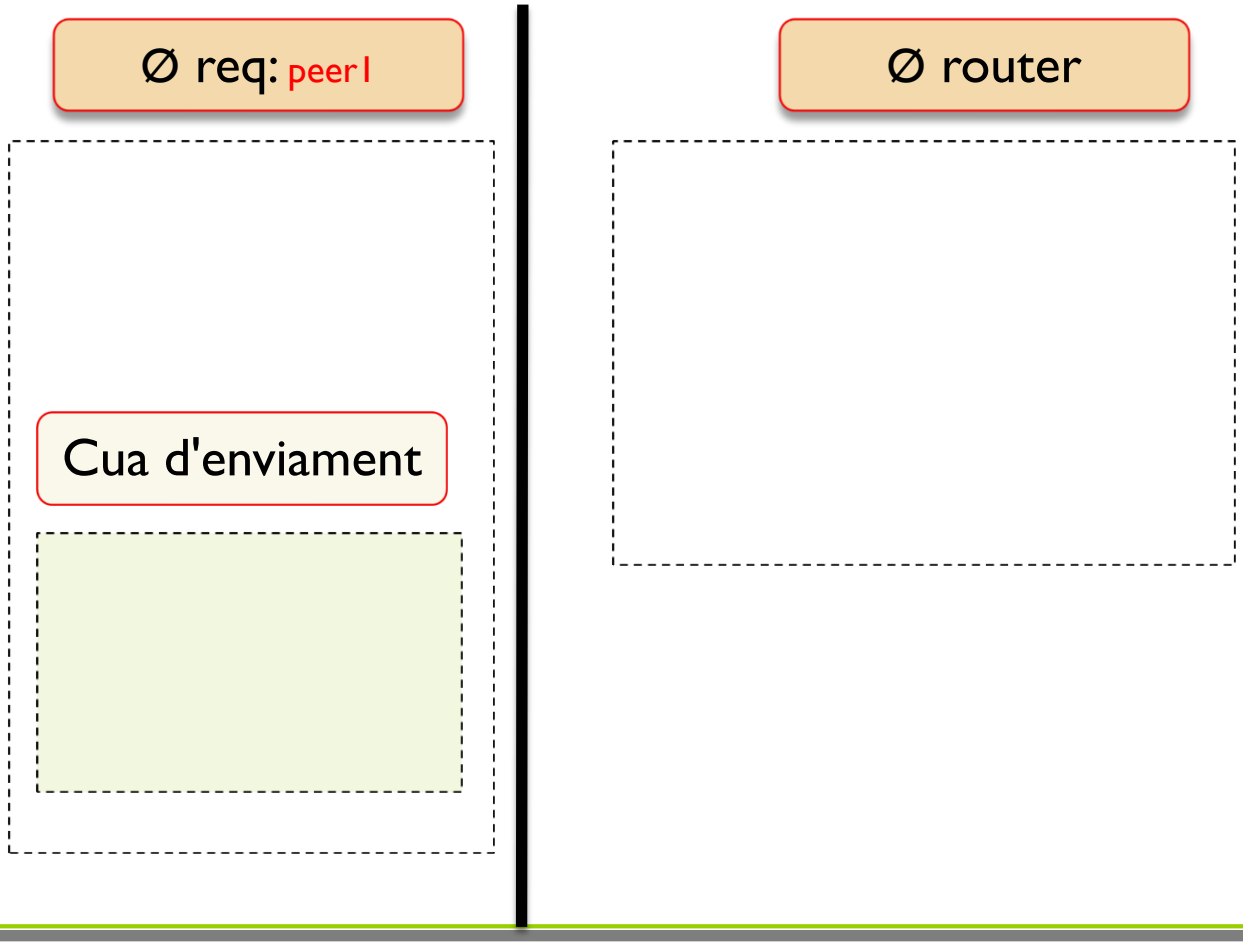
- Sockets bidireccionals asincrònics
- Permet enviar missatges a agents específics
  - Assigna una identitat a cada agent amb el qual es connecte
  - La identitat és aquella donada a l'agent en el seu programa
    - sock.identity = 'my name';
    - Quan l'agent no tinga una identitat associada
      - El socket router crea una identitat aleatòria per a aquest agent connectat
      - La identitat creada es manté mentre la connexió dure
      - En tancar la connexió i restablir-la, la identitat canvia
    - Els identificadors són cadenes arbitràries de fins a 256 octets
- Quan el socket router passe un missatge a l'aplicació
  - Afig un segment inicial amb l'identificador de l'agent emissor

4.4.2. Sockets router

- Quan el socket router envia un missatge...:
  - Utilitza el primer segment com a identitat de la connexió; així...
    - Un socket router manté un parell de cues de recepció i enviament **per connexió**.
    - El primer segment s'usa per a localitzar la connexió apropiada. Quan la troba...:
      - El primer segment és eliminat implícitament.
        - El programador no ha de fer res.
      - La resta del missatge es deixa a la cua d'enviament de la connexió.
        - Això completa l'enviament.
  - Això permet una gestió router-dealer molt senzilla als *brokers*:
    - El procés *broker* usa un socker router "frontend" i un dealer "backend".
    - Cada missatge rebut pel router s'envia pel dealer.
    - Cada missatge rebut pel dealer s'envia pel router.
    - En ambdós casos, no cal modificar cap segment del missatge.

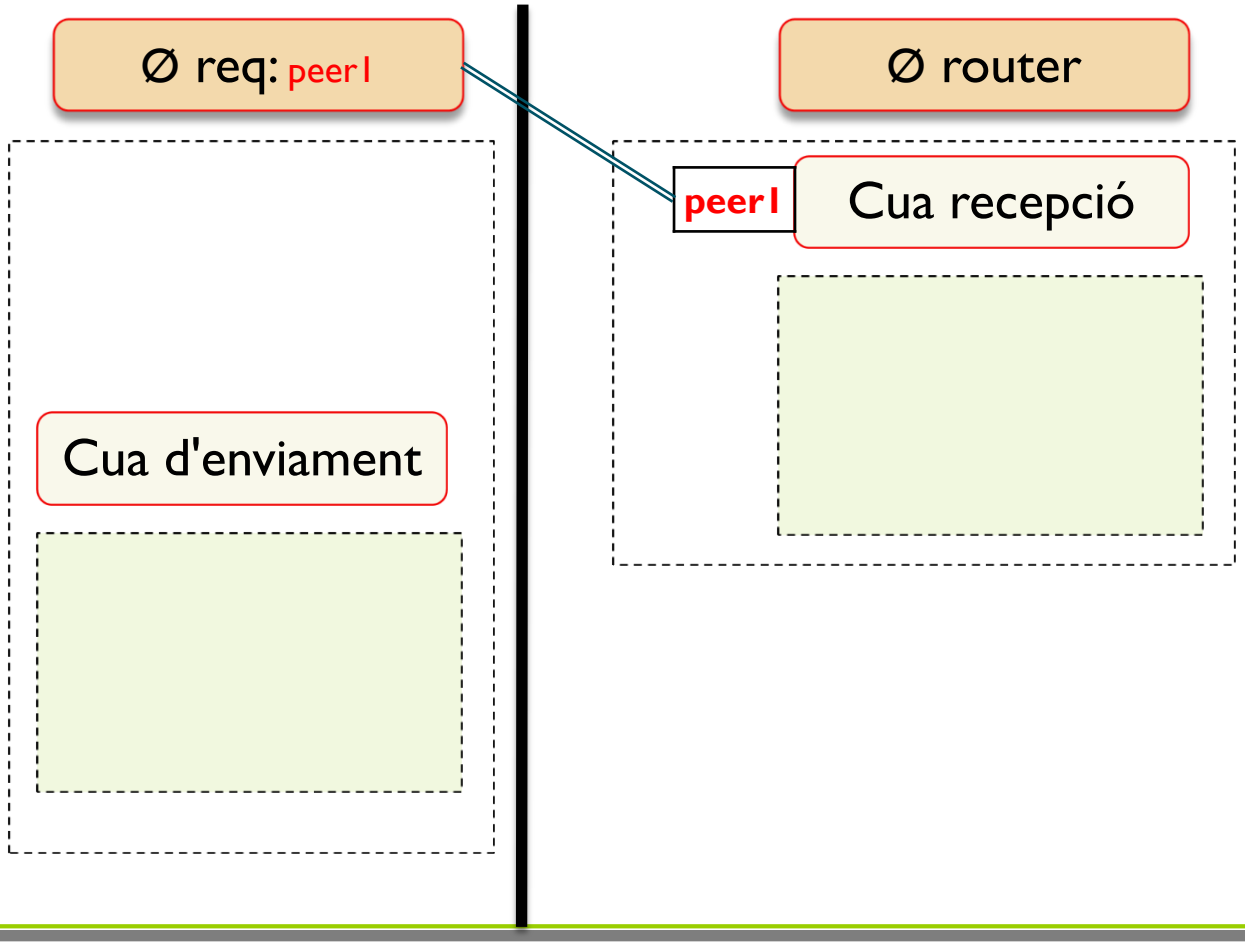
4.4.2. Sockets router: exemple amb agent req

- L'agent req té identitat "peer1"



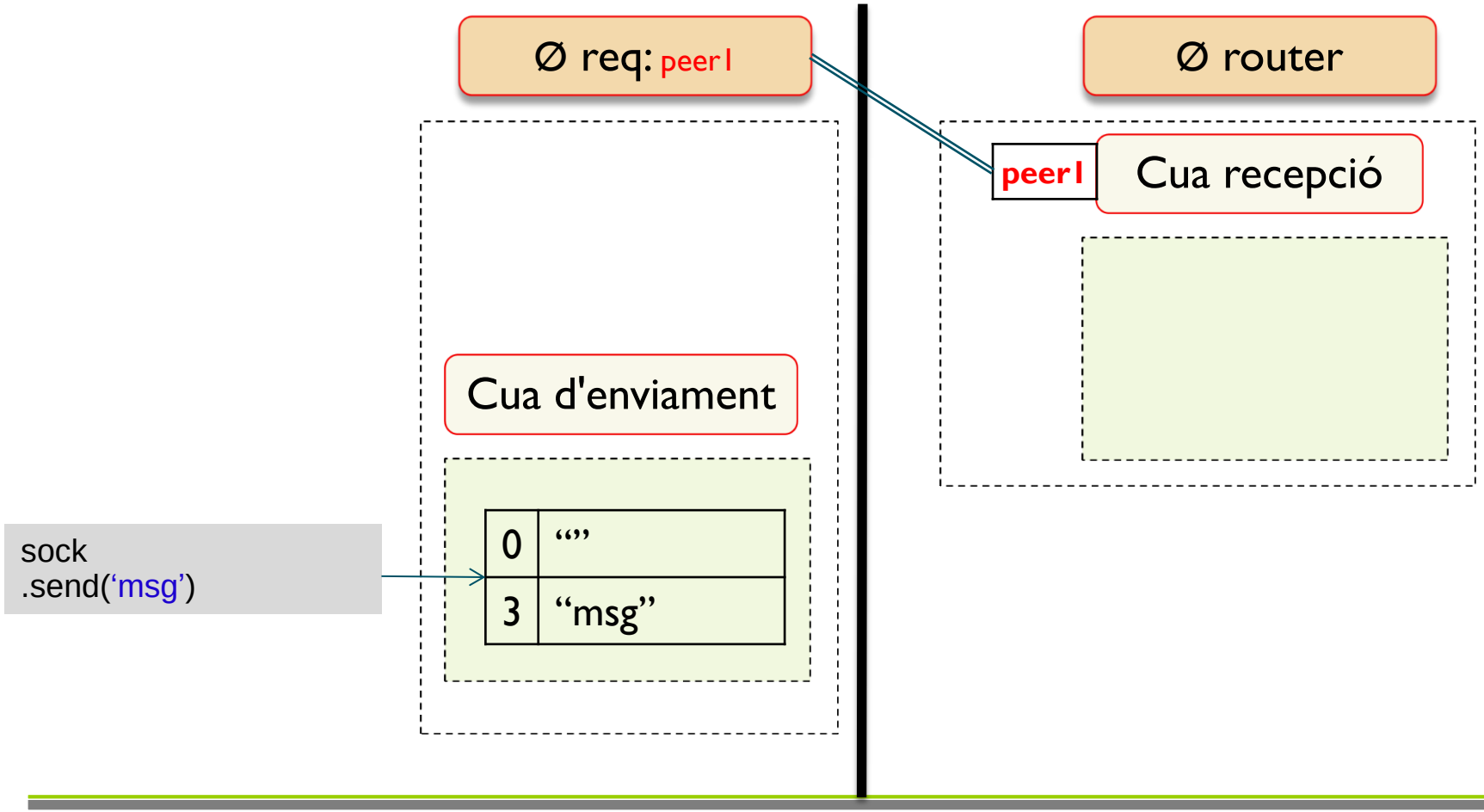
4.4.2. Sockets router: exemple amb agent req

- L'agent req connecta amb el router
  - El router obté la seua identitat, l'emmagatzema i li associa cues d'enviament i recepció



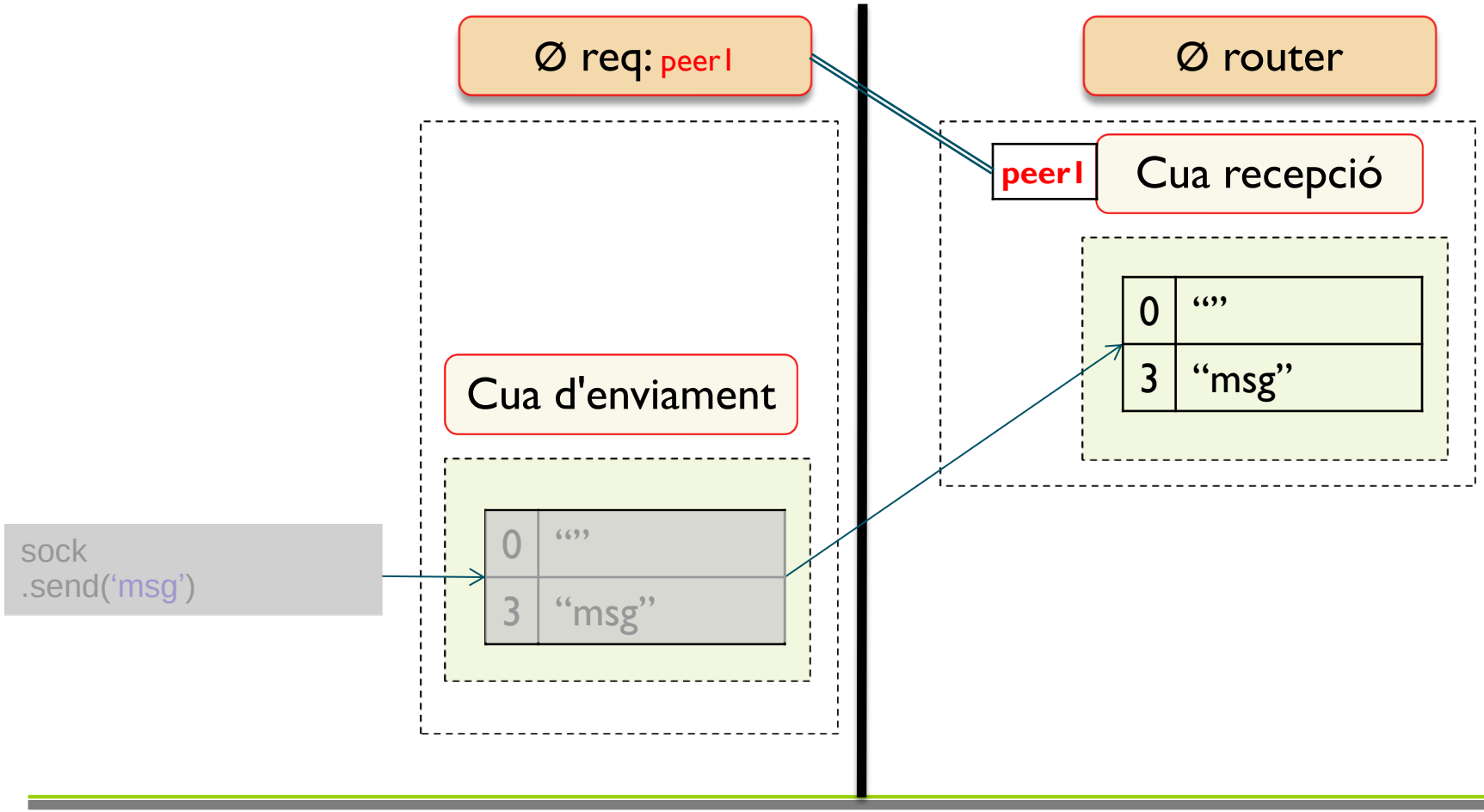
4.4.2. Sockets router: exemple amb agent req

- L'aplicació req envia un missatge
  - El socket req afig el delimitador...



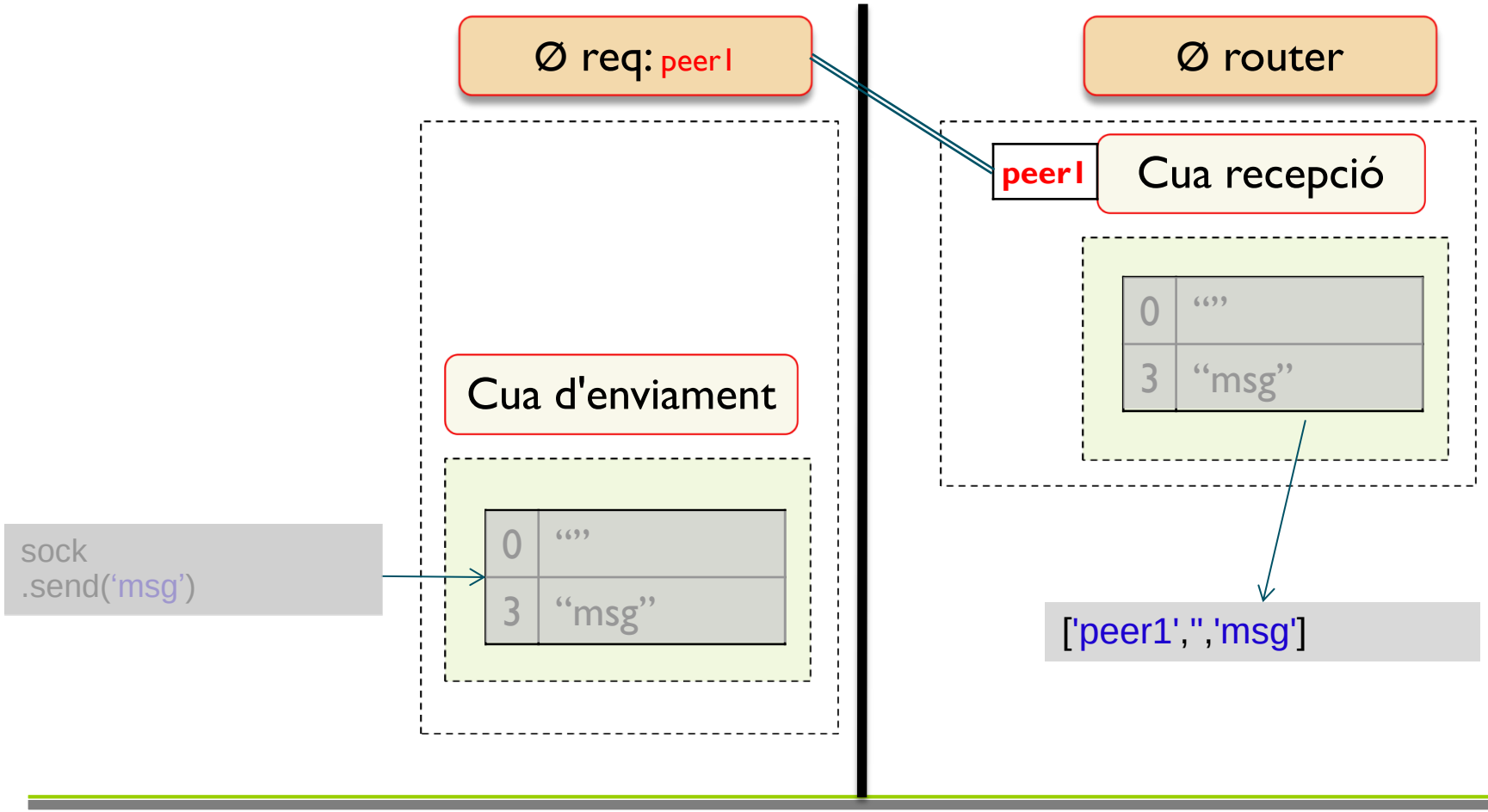
4.4.2. Sockets router: exemple amb agent req

- L'aplicació req envia un missatge
  - El socket req afig el delimitador... i ho envia



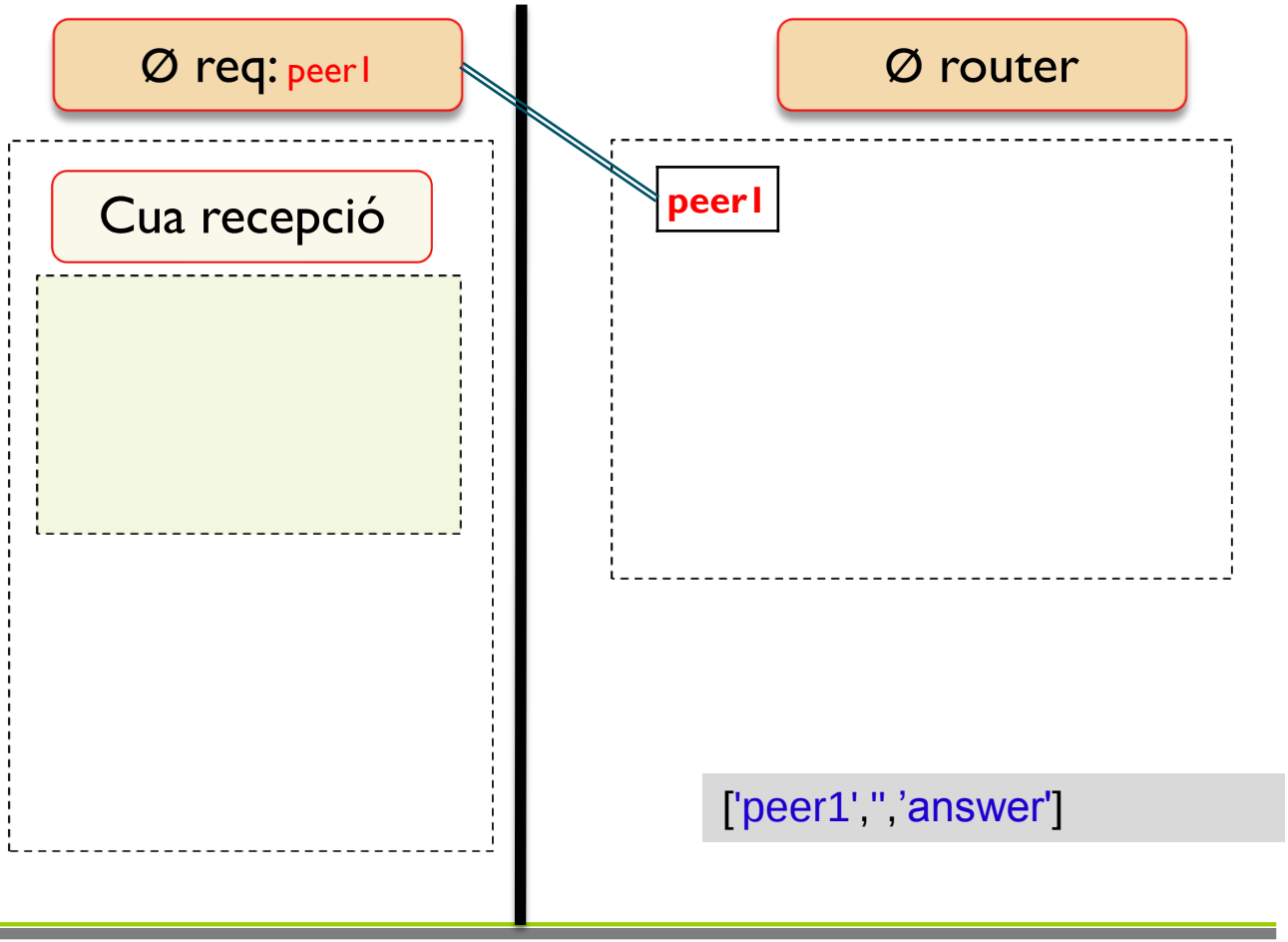
4.4.2. Sockets router: exemple amb agent req

- El socket router lliura el missatge a la seua aplicació
  - Amb la identitat de l'emissor en un nou segment inicial



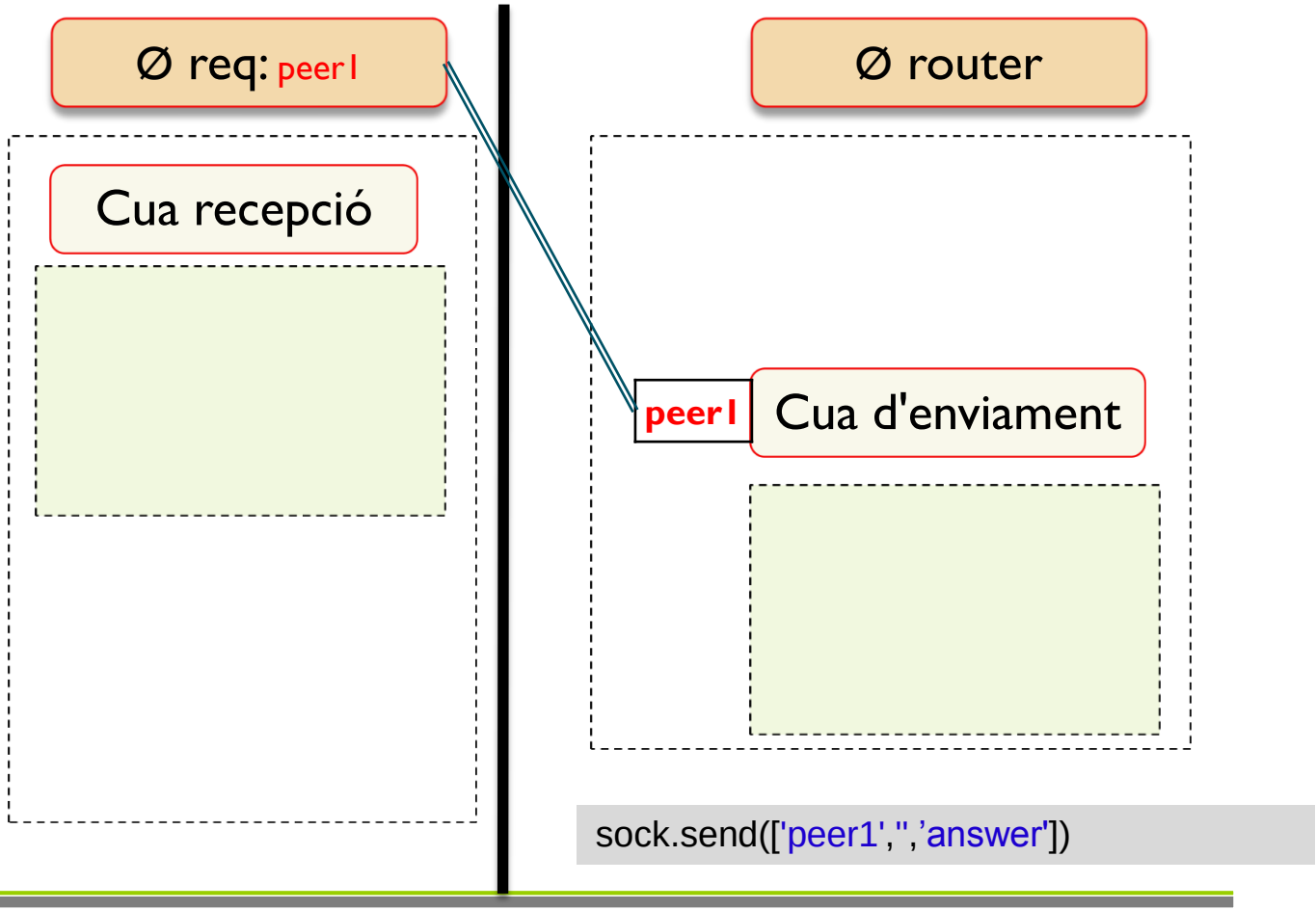
4.4.2. Sockets router: exemple amb agent req

- L'aplicació del router crea una resposta, construint el missatge de resposta
  - El primer segment conté la identitat de l'agent que rebrà la contestació



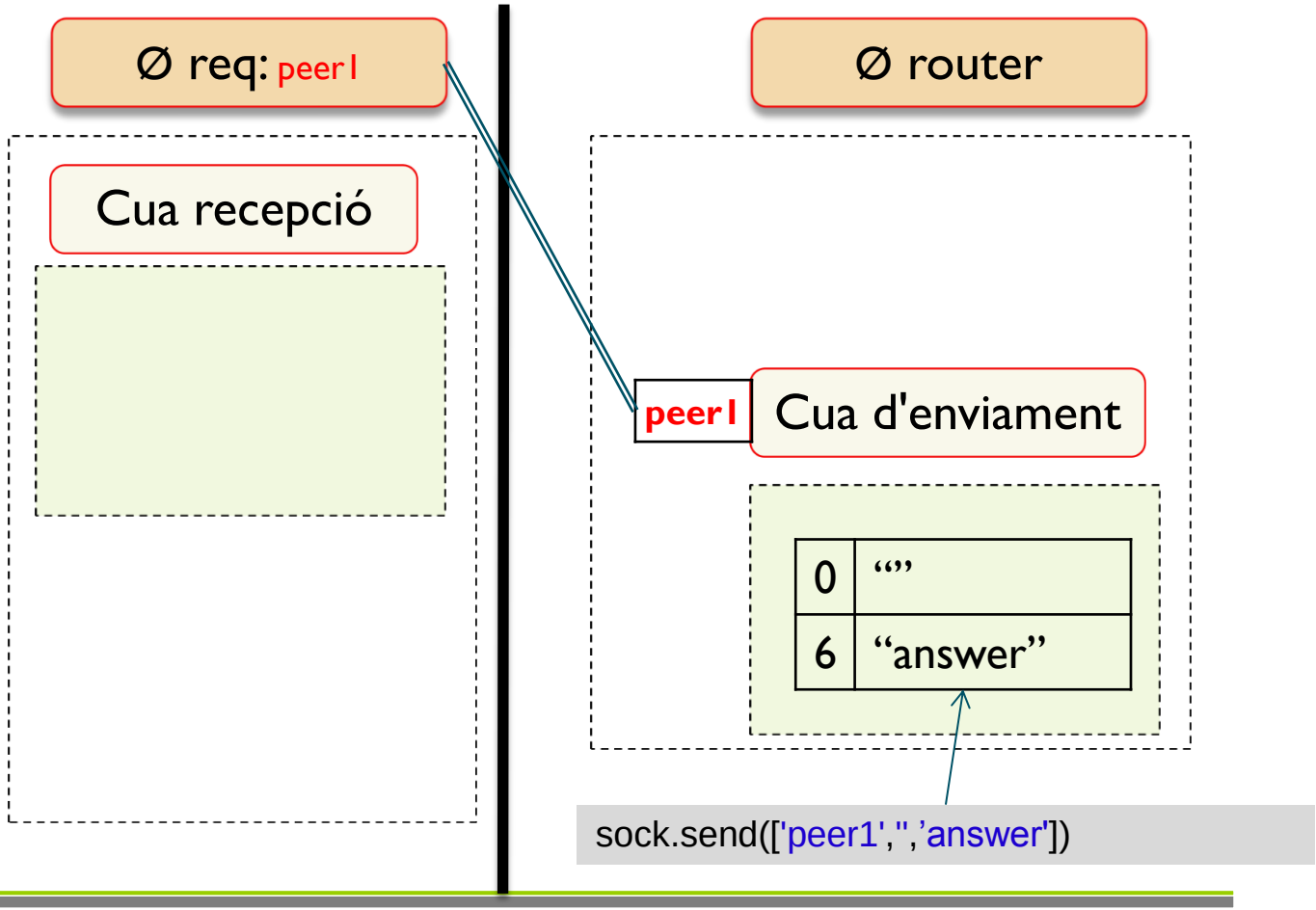
4.4.2. Sockets router: exemple amb agent req

- L'aplicació del router envia el missatge
  - El socket router selecciona la cua d'enviament basant-se en la identitat



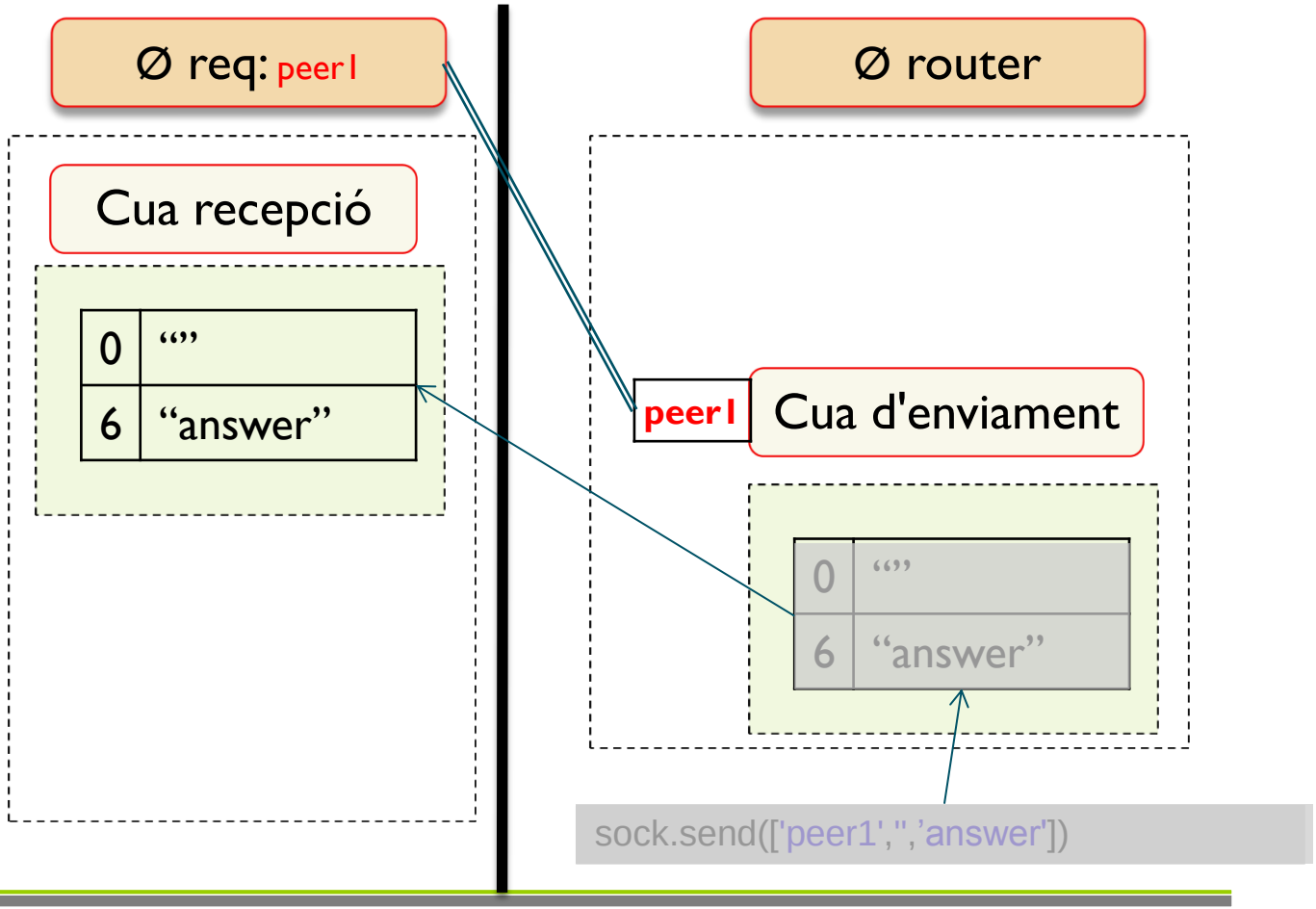
4.4.2. Sockets router: exemple amb agent req

- El socket router lleva el segment amb la identitat
  - Deixa la resta del missatge per a enviar...



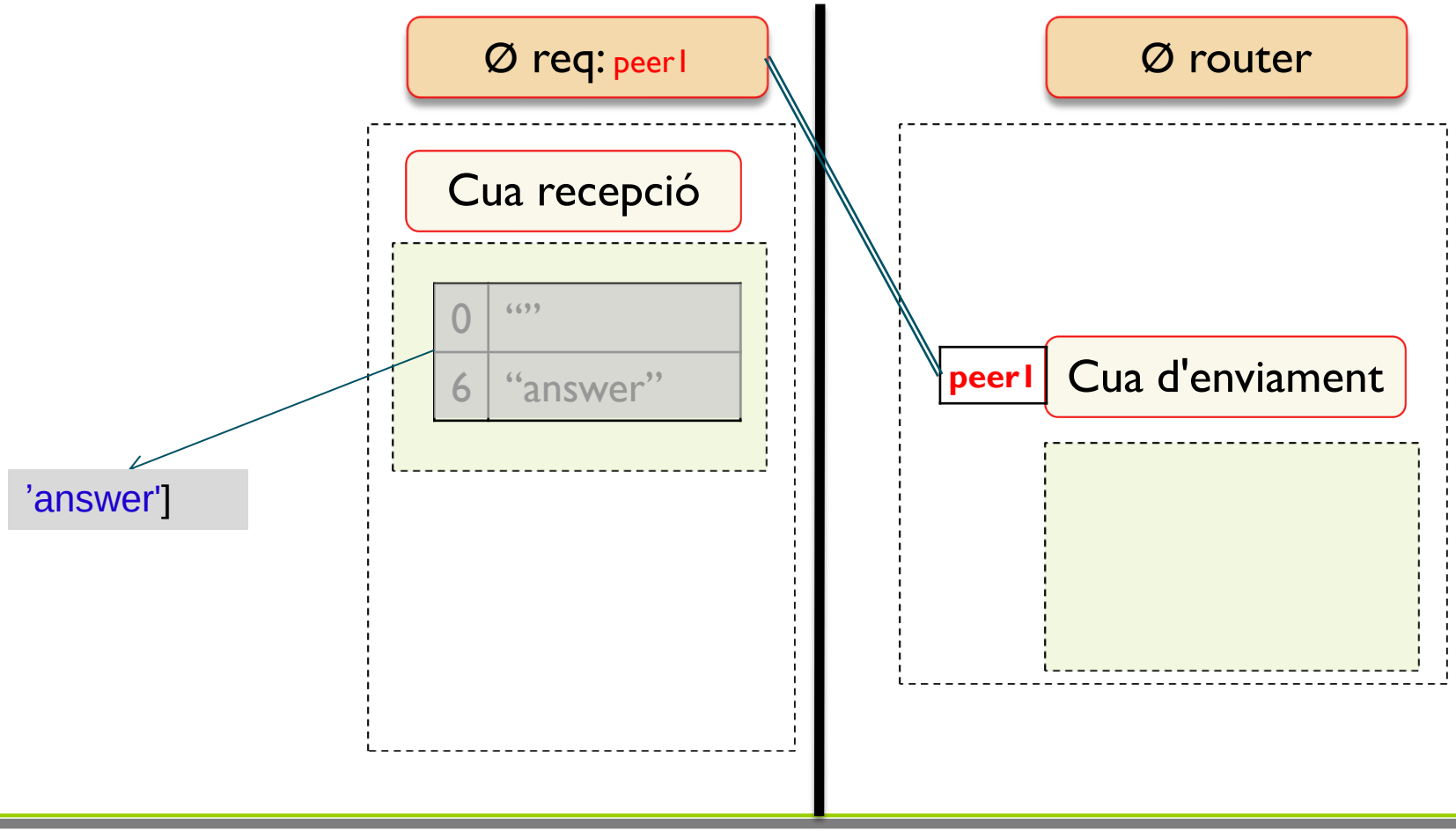
4.4.2. Sockets router: exemple amb agent req

- El socket router lleva el segment amb la identitat
  - Deixa la resta del missatge per a enviar... i ho envia



4.4.2. Sockets router: exemple amb agent req

- El socket req ho entrega a la seua aplicació
  - Eliminant el segment delimitador



Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

5. Altres middleware

- Gestió d'esdeveniments
  - Sovint inclòs en sistemes de missatgeria
    - Patró publicador-subscriptor
  - Exemple: JINI
- Seguretat
  - Autenticació
    - Una tercera part garanteix la identitat d'un agent
    - Exemple: OpenID
  - Autorització
    - Una tercera part autoritza una petició
    - Exemple: OAuth
  - Integració amb altres protocols
    - Exemple: SSL/TLS i HTTPS
- Suport transaccional
  - Coordinació de modificacions d'estat distribuïdes atòmiques
    - Suporta les situacions de fallada

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

6. Conclusions

- La complexitat de sistemes distribuïts ha de ser resolta amb una adequada gestió del codi i dels serveis
- Els estàndards simplifiquen aquest escenari familiaritzant-nos amb les tècniques a utilitzar
- Els "middleware" (nivell de l'arquitectura entre l'aplicació i les comunicacions) implanten solucions comunes per a aquests problemes
- Principals objectius dels middleware
  - Tasques de comunicacions
  - Petició de serveis
- Principals variants
  - Missatges
    - Gestió persistent/transitòria
    - Basats en gestor / Sense gestor
- Altres middleware
  - Seguretat
  - Transaccions

Índex

1. Programació distribuïda fiable
2. Middleware
3. Sistemes de missatgeria
4. ZeroMQ
5. Altres middleware
6. Conclusions
7. Bibliografia

7. Bibliografia

- http://zguide.zeromq.org/page:all
  - Permet lectura on-line
  - Existeix una versió en PDF
  - El lloc web manté informació addicional