Unit 3 – Middleware. ZeroMQ

Network Information System Technologies

Index

1. Introduction

- Large Scale Distributed Systems Components are
  - Specified and Developed independently
    - No big committee nailing all possible use cases
  - Deployed autonomously
    - Each one designed as a Distributed Systems Agent
    - But Establish dependencies among themselves
      - Consuming or providing functionality from/to other agents
- Need to easily interact usefully
  - E.g. a routing planning service may depend on a GIS service providing basic distance information.
  - E.g. an entrance authorization system may need to contact a remote biometrical recognition service

1. Introduction

- Even close-knit systems
  - Developed by more than one developer
  - Multiple components clearly interdependent
- In all cases:
  - Need to deal with the complexities of the distributed environment
  - The product needs to be as error-free as possible
    - Debugging problems is orders of magnitude more complex in a distributed system
  - Submission to strict development schedules
    - Lose the least amount of time in coding

1. Introduction

- How to make sure we do not need absolute geniuses
  - To write the millions of lines of code needed
  - To manage the systems deployed
- One Problem is the Complexity of the details
  - Dealing with complexity is a source of errors
  - Dealing with too many details at once is a recipe for disaster
    - And long debugging sessions
      - Often in production
- Another problem is the fact that many tasks are repetitive
  - And we can save resources by providing common implementations

1. Introduction

- Complexity sources
- <u>Mainly from Making requests to a service</u>
  - Thus, communications related
  - Finding servers implementing services
    - How can clients contact services without knowing the exact machine of their servers
    - How are servers addressed
    - How are clients addressed back in a request
  - Specifying the functionality of services
    - What is the API of a service
    - How to find is the version has changed

1. Introduction

- Formatting information transmitted
  - How can developers build and interpret requests to services
    - E.g., how is byte 8 interpreted?
  - Compatibly with the programming environments being used at both ends
    - Eg. Client uses Java, Server uses C
- Synchronization between requester and server
  - Ordering of actions: how do clients know services have reacted to their requests
  - Reception of results: how do servers return results, and clients retrieve them
- Security
  - How can developers of services know when to process a requests
  - How can security properties be ensured
- Failure handling
  - How do agents know when to take remedial actions for failures
  - E.g. how to figure out when a server dies
  - E.g. How to figure out if information transmitted has been lost?
  - …

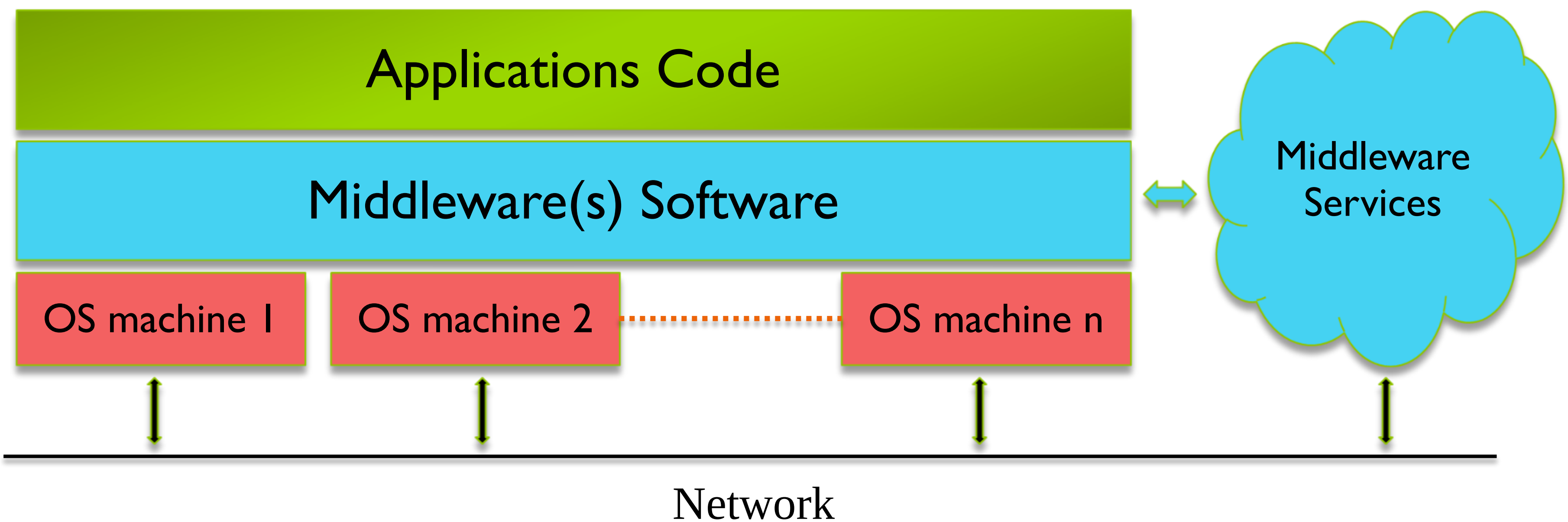
1. Introduction

- Complexity-taming approaches
  - Standards (de facto and de jure)
    - Introduce streamlined ways to do things
      - Practice makes masters
    - Helps build familiarity for developers
      - They do not have to learn something new every time
    - Help interoperability
      - Greater choice for system's integrators/managers
    - Providing High Level functionality
      - Less code to write for the developer
      - Less complexity to deal with
  - Middleware

Index

2. Middleware

- Layer(s) of software and services between application software and Network/Communications layer
- Introduces various "Transparencies"
- Transparency
  - Reduction of complexity by hiding and dealing with details in a uniform way



2. Middleware: Desirable Characteristics

- Developer's perspective
  - Easy to write
    - Clear and well defined concepts
    - Low complexity in elements handled
      - Avoid coding errors
  - Reliable result
    - Well understood defined/established/standardized ways of doing things
  - Maintainable
    - Changes in the API should have low impact in modifications
- Manager's perspective
  - Easy set-up, change
  - Easy/possible to interact with products from third parties (interoperability)

Index

1. Producing reliable DS Software
2. Middleware
3. Messaging Systems
4. ZeroMQ
5. Other Middleware
6. Conclusions
7. References

3. Messaging systems

- Asynchronous in nature
  - Decoupling between sender and receiver
- Sending discrete pieces of information
  - Message: atomically transmitted (all or nothing)
  - Arbitrary sizes
  - Support for structuring messages
  - Queuing
    - With some guarantees of order
- No shared state imposed view
  - Potentially better to produce scalable systems
  - Potentially better to avoid concurrency difficulties
- Examples
  - Big Standard: AMQP
    - E.g. RabbitMQ
    - Apache ActiveMQ
  - STOMP
  - 0MQ

3. Messaging systems: Approaches

- Two main approaches
  - Transient (stateless) systems
    - Require the recipient to be up to receive a message
  - Persistent
    - Messages saved in buffers: recipient does not have to be up at sending time
- Within Persistent implementations
  - Broker-based
    - Specific servers store messages and provide strong guarantees
    - Overhead derived from the need to persistently store in secondary storage
    - E.g. AMQP
  - Broker-less
    - Senders/receivers maintain the queues
      - Typically in memory
    - Weaker persistence guarantees
      - But still, decoupling between sender and receiver readiness to send/receive
    - Can be used to build broker-based system when needed
    - E.g. 0MQ

Index

4. ZeroMQ

4.1. Introduction: Goals of ØMQ

- Simple communications middleware
  - Easy configuration: URLs to name endpoints
  - Easy & familiar to use: BSD-like socket API
- Widely Available
  - Portable implementation
- Support basic patterns
  - Eliminate need for each developer to re-invent the wheel
  - Make it very useful right away
- Performance
  - No unnecessary overhead
  - Tradeoff between reliability and performance
- The same code can be used to connect
  - Threads within a process
  - Processes, within a machine
  - Machines, through an IP network.
  - Only URL changes needed

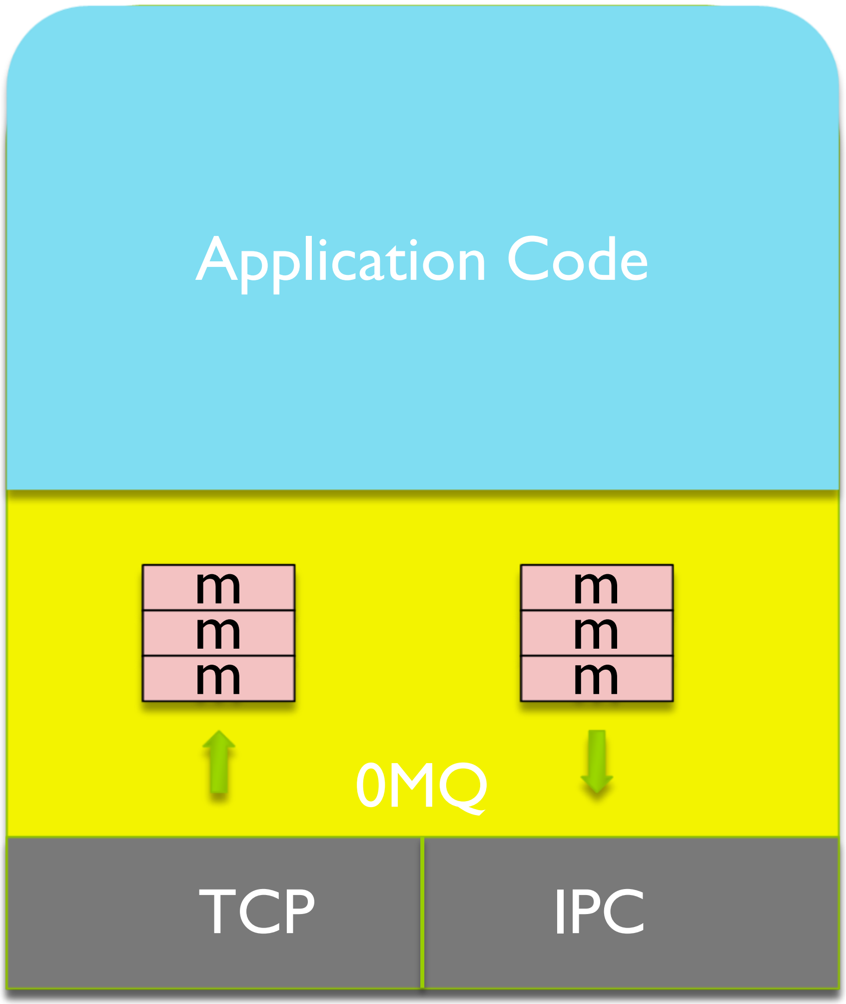
4.1. Introduction: Main Characteristics

- Message - based
  - Weakly persistent: main memory queues
- It is just a library
  - No need to start specific middleware servers anywhere
  - Implemented in C++
  - Available on most OS
    - Linux_XYZ
    - Windows
    - BSD
    - MacOSX
  - Bindings available for most programming languages/environments

4.1. Introduction: Technology

- Provides sockets to send and receive messages
  - send/receive, bind/connect interface for a socket
- Can use the following transports:
  - Inter-process
    - TCP/IP
    - IPC (Unix sockets)
- Transport used to instantiate socket
  - Easily changed by a configuration change

4.1. Introduction: View of a ØMQ Process



- Application links with ØMQ library
- ØMQ maintains in-memory queues
  - At sender
  - At receiver
- ØMQ uses communication layers

4.1. Introduction: Installation

- **ØMQ is a library**
  - It must be installed before its usage in processes
  - In order to use it in NodeJS programs, a module should be imported: **zeromq**
    - To this end, this line is needed:
      - const zmq = require('zeromq')
    - When that module is installed, it also installs the library.

- **To install the module, use this command:**
  - **npm install zeromq**

4. ZeroMQ

4.2. Messages: Message oriented middleware

- Messages are what's sent
  - No framing problem for the app
  - Buffering is taken care of
- Messages can be "multi-part" (i.e., multi-segment)
  - Simple structuring support for messages
- Messages are atomically delivered
  - All parts are delivered, or nothing is delivered
- Message send/receive is asynchronous
  - Internally, 0MQ moves messages back-and-forth the in-memory queues to the transports
- Connections/reconnections among peers automatically handled

4.2. Messages

- Message content is transparent to 0MQ
- No support for marshaling
  - No format required
  - Messages are Blobs to 0MQ
    - But 0MQ API supports simple string serialization into a message

```
zsock.send([“this is", "a", "message“]);
```

7	this is
1	a
7	message

- NOTE
  - Some socket types use the first segment

4.2. Messages: Consequences

- You must design your own payload format/protocol
- In many cases it may be as simple as just using plain strings
- It is possible to use ANY encoding
  - Binary is fine
- Simple approach: XML messages
  - Use XML parsers
- Simpler approach: JSON messages
- Simplest approach
  - Use each segment for a different piece of information, encoded its own way: e.g
    - Segment 1 is the name of the interface being invoked, as a string
    - Segment 2: is the version of the API interface, as a string
    - Segment 3: Is the name of the operation
    - Segment 4: is an integer: the first argument
    - Segment …

4. ZeroMQ

1. Producing reliable DS Software
2. Middleware
3. Messaging Systems
4. ZeroMQ
   1. Introduction
   2. Messages
   3. 0MQ API
   4. Advanced socket types
5. Other Middleware
6. Conclusions
7. References

4.3. 0MQ API

1. Sockets
   - Sending/receiving uses sockets
   - Several socket types
   - Sockets can bind/connect
2. Patterns of communication
   - Supported by specific socket types

4.3.1. 0MQ Sockets

- Creating a socket is simple:

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

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

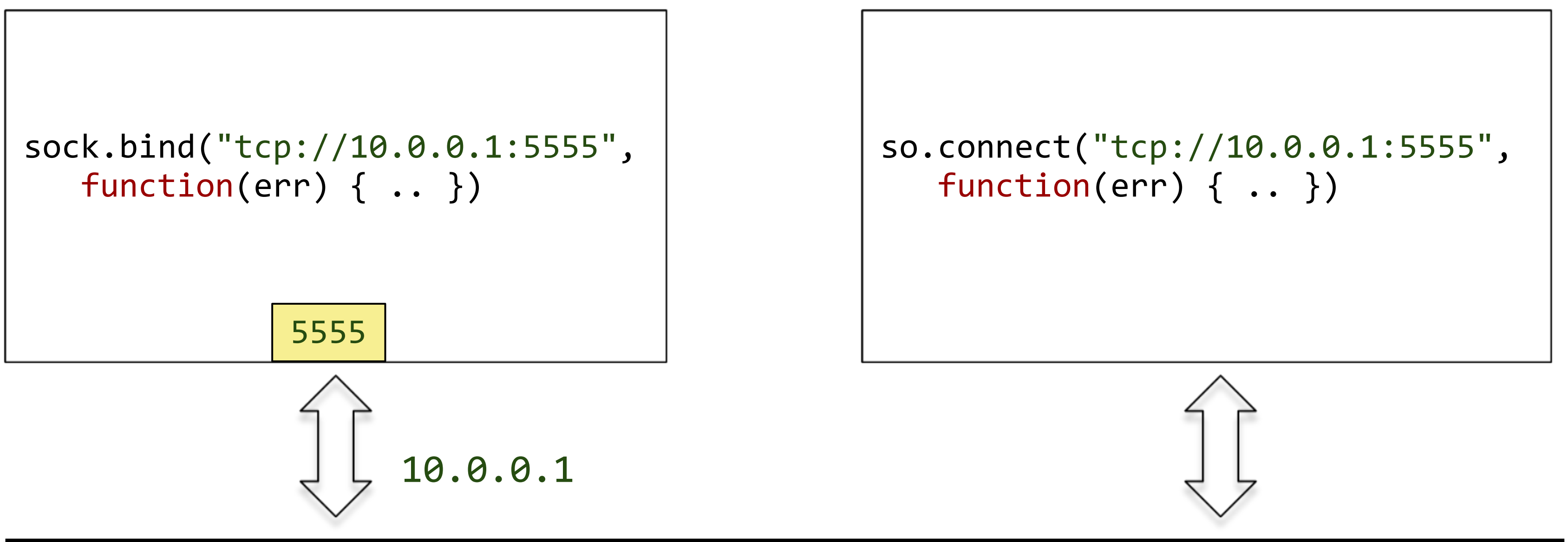
- Where <SOCKET TYPE> is one of

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

- What types to use will depend on the connection patterns they intervene in

4.3.1. Sockets: Establishing communication paths

- One process performs a **bind**
- Other processes perform **connect**
- When done, **close**
- **bind/connect** are decoupled: no ordering requirements



4.3.1. Sockets: multiple connections are possible

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

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

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

10.0.0.1

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

5555

10.0.0.2

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

5556

4.3.1. Sockets: Connections and queues

- Sockets have message queues associated
  - incoming queues, to hold messages from connected peers
    - They raise the "message" event when holding some message
  - outgoing queues, holding messages to be sent to peers
    - Where messages sent from the application are held
- router sockets keep one pair of outgoing/incoming queues per connected peer
  - The rest of the sockets do not distinguish among peers
  - pub sockets fall out of this discussion
- pull and sub sockets only set up an incoming queue
- push and pub sockets only have outgoing queues

4.3.1. Sockets: binding or connecting

- When to **bind** and when to **connect**
  - Most times indifferent: configuration convenience
- Observations
  - All peers must meet at an endpoint
  - Endpoints are referred to by their URL
  - For TCP transport
    - The IP address must belong to one of the bind socket's interfaces
      - The bind socket only needs local IP configuration (or none)
        - Does not need to know where the other peers are
    - The connect socket needs to know where the binding socket is (IP)

4.3.1. Sockets: Transports: TCP

- URL: `tcp://<address>:<port>`
- Three ways to specify the address

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

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

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

- *: binds to all interfaces
- “eth0”: binds to all addresses associated with interface “eth0”

4.3.1. Sockets: Transports: IPC

- Inter Process Communication (Unix sockets)
- URL: ipc://<path-to-socket>

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

- Need *rw* (read & write) permissions over socket at <path-to-socket>

4.3.1. Sockets: Sending messages

- Segments can be extracted in one call from an array, …

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

- Segments must be **buffers** or **strings**.
  - Strings are converted to buffers, using UTF8 encoding
    - Non-strings are first converted to strings

4.3.1. Sockets: Receive

- Based on “message” events on the socket
  - **arguments** of handler contain the segments of the message
  - NOTE: segments are binary buffers

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

- For variable number of segments, use “arguments” directly…

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

- … or convert to array first

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

4.3.1 Sockets: socket options

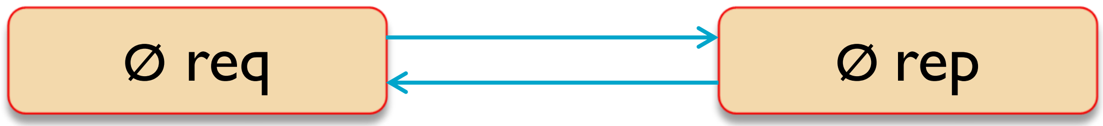
- Many.
- Two important ones: **identity**, and **subscribe**

```
sock.identity = ’frontend’
sock.subscribe(‘SOCCER’)
```

- **identity** convenient when connecting to *router* sockets
  - Sets the ID of a connecting peer to the router
  - **It should be set before invoking the connect() method!!!**
- **subscribe**, used by *sub* sockets.
  - Sets the prefix filter applied by the *pub* socket

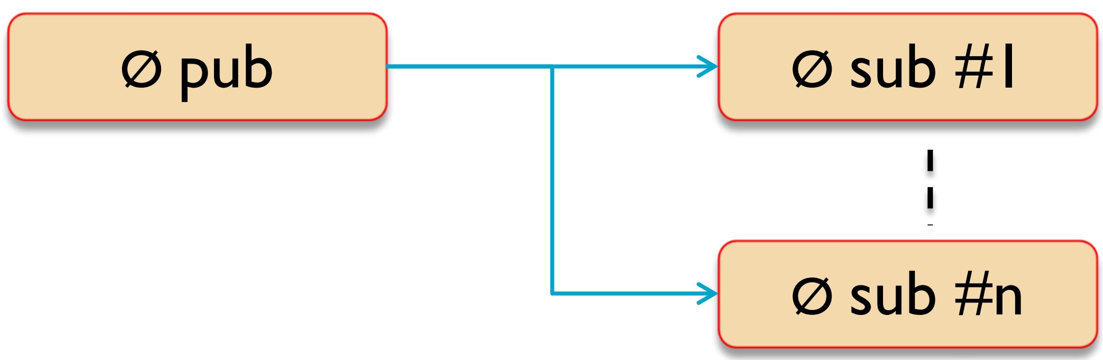
4.3.2. Basic patterns

- Request/Reply (synchronous)



- Push-pull

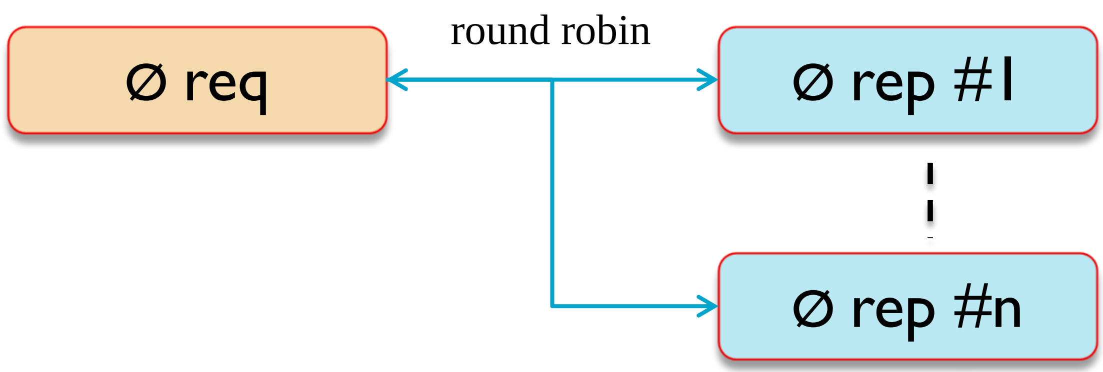
- Pub-Sub



4.3.2. Basic Patterns: request/reply

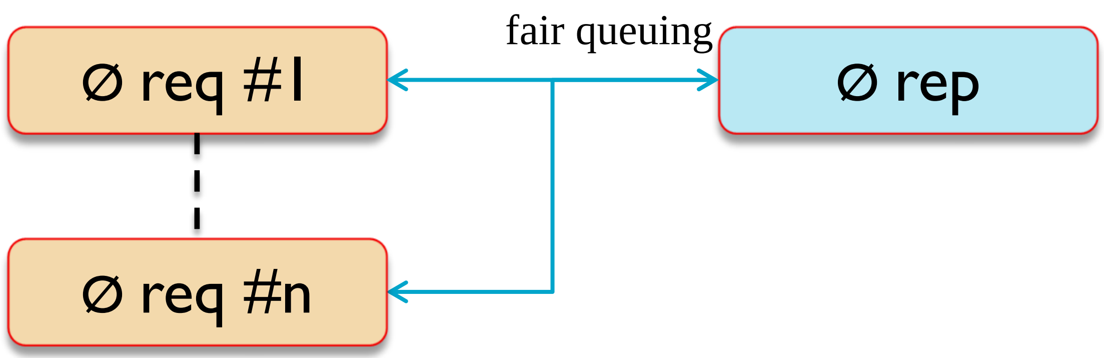
- Implemented using **req** sockets for the client
  - **rep** sockets for the server
- Each message sent via **req** needs to be matched with a corresponding reply from the **rep** socket of the server
- Synchronous communication pattern
  - All request/reply pairs are totally ordered
  - Endpoints can react asynchronously, though.
- When a message has been sent through a **req** socket sending a new message through that socket queues it locally
  - Until the response message is received
    - Then the waiting message is sent
- When a request has been received through a **rep** socket, the arrival of a new request through that socket gets it queued
  - Until the first request is answered
    - Then the waiting message is delivered
- Each reply message sent beforehand through a **rep** socket remains enqueued until its associated request is received
  - When such request arrives, that reply propagation is resumed

4.3.2. Basic Patterns: request/reply with distribution



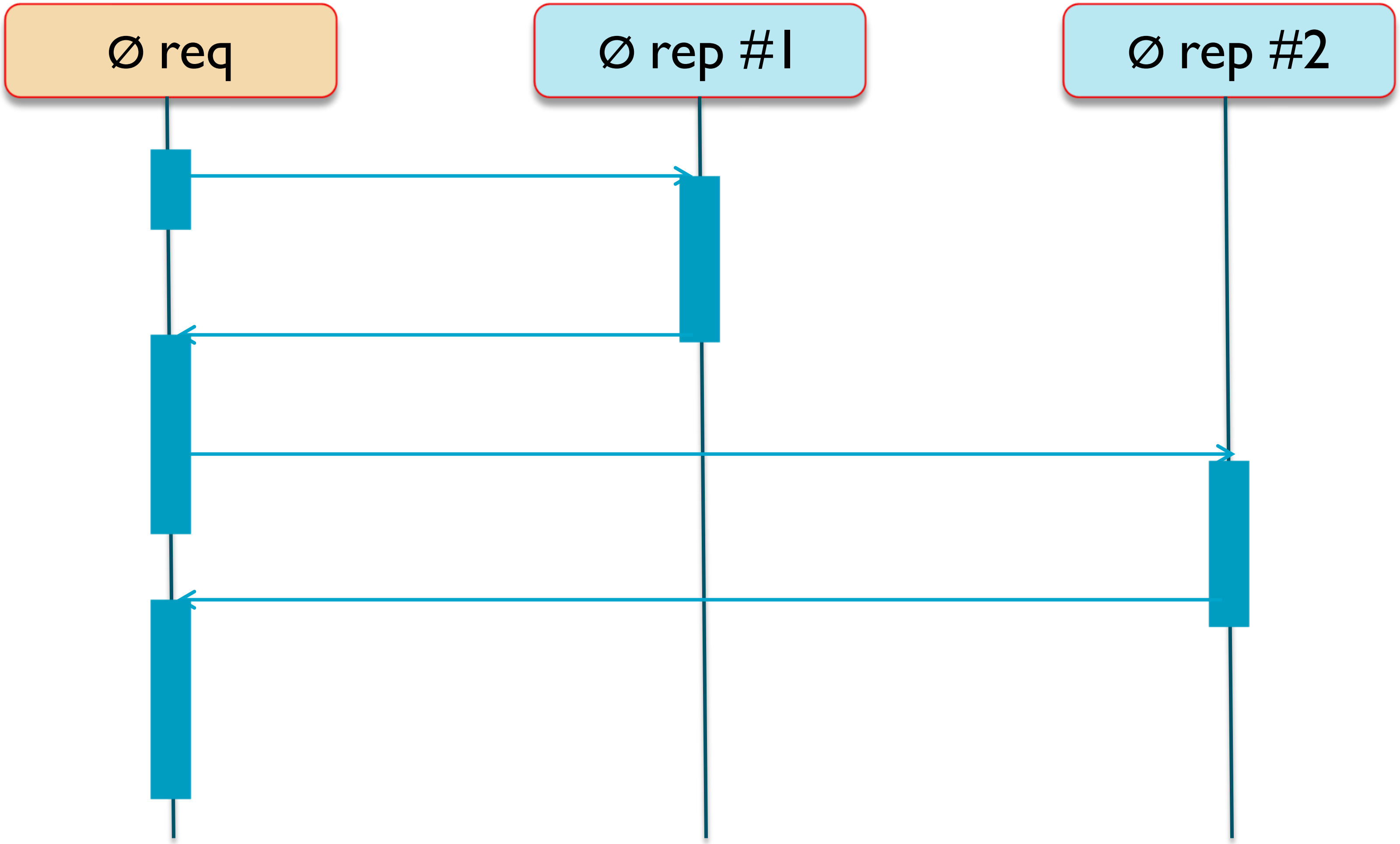
- When a **req** connects to more than one rep, each request message is sent to a different **rep**,
  - Follows a round-robin policy.
- Operation continues being synchronous:
  - 0MQ will not send new requests until each reply is received
  - No parallelization of requests

4.3.2. Basic Patterns: multiple requesters

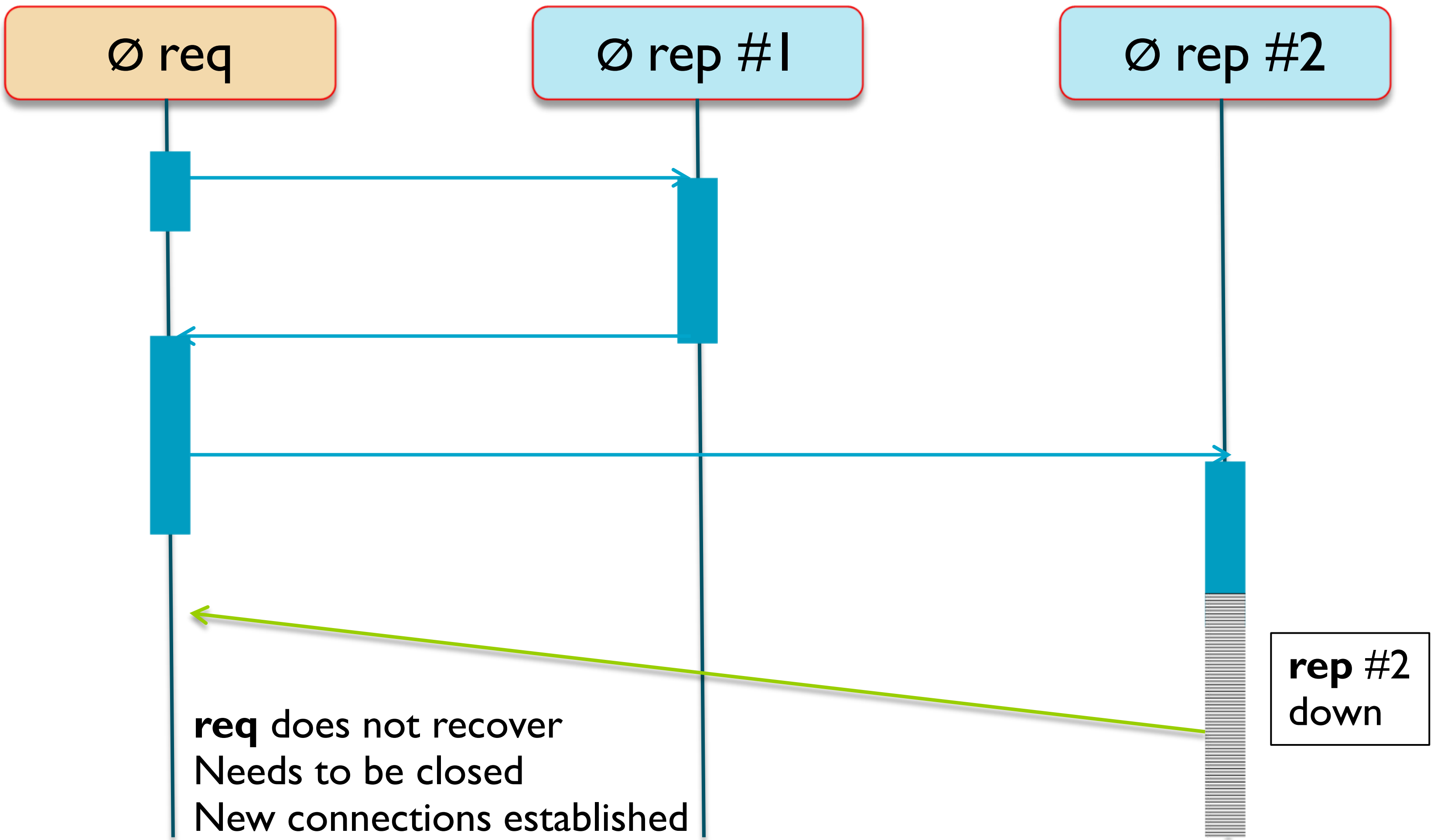


- Typical set-up for a server
- rep socket fairly queues incoming messages
  - No req socket is starved

4.3.2. Basic Patterns: Request/reply sequence



4.3.2. Basic Patterns: Request/reply failures



4.3.2. Patterns: Basic req/rep

```
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. Basic Patterns: Basic req/rep, two servers

```
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')
})
```

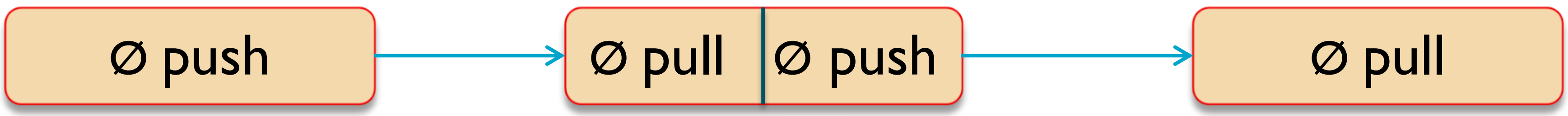
4.3.2. Patterns: req/rep, structure of message

- Messages exchanged have a first segment empty
- Referred to as the delimiter
- The req socket adds it, without app intervention
- The rep socket removes it before handing it to the application
  - But it adds it again on the reply
- The req socket removes it from the reply.

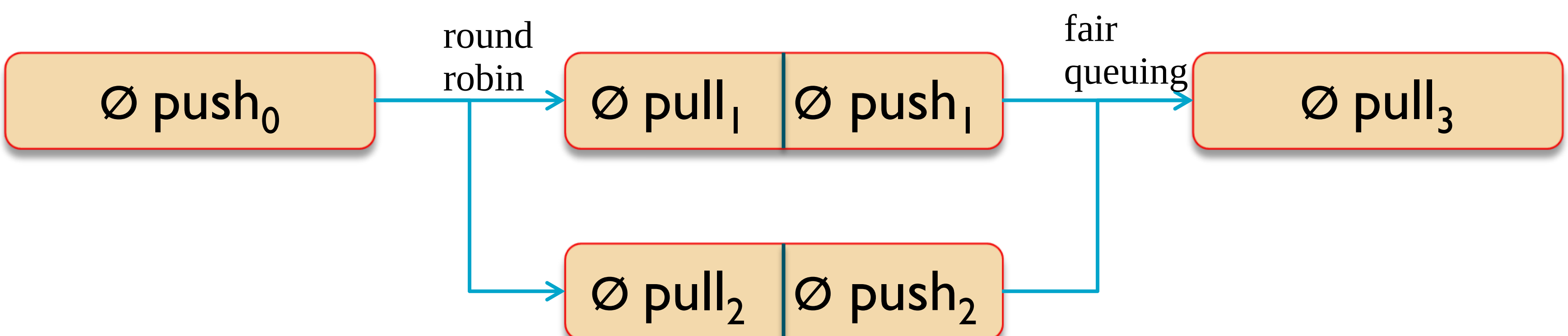
0	“”
7	“this is”
1	“a”
7	“request”

4.3.2. Basic Patterns: push/pull

- Unidirectional data distribution
- Sender does not expect a reply back.
  - Messages do not wait for replies: concurrent sending.



- Accepts also multiple connections.
  - E.g., typical map-reduce organization:



4.3.2: Patterns: push/pull example, producer/consumers

```
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)
})
```

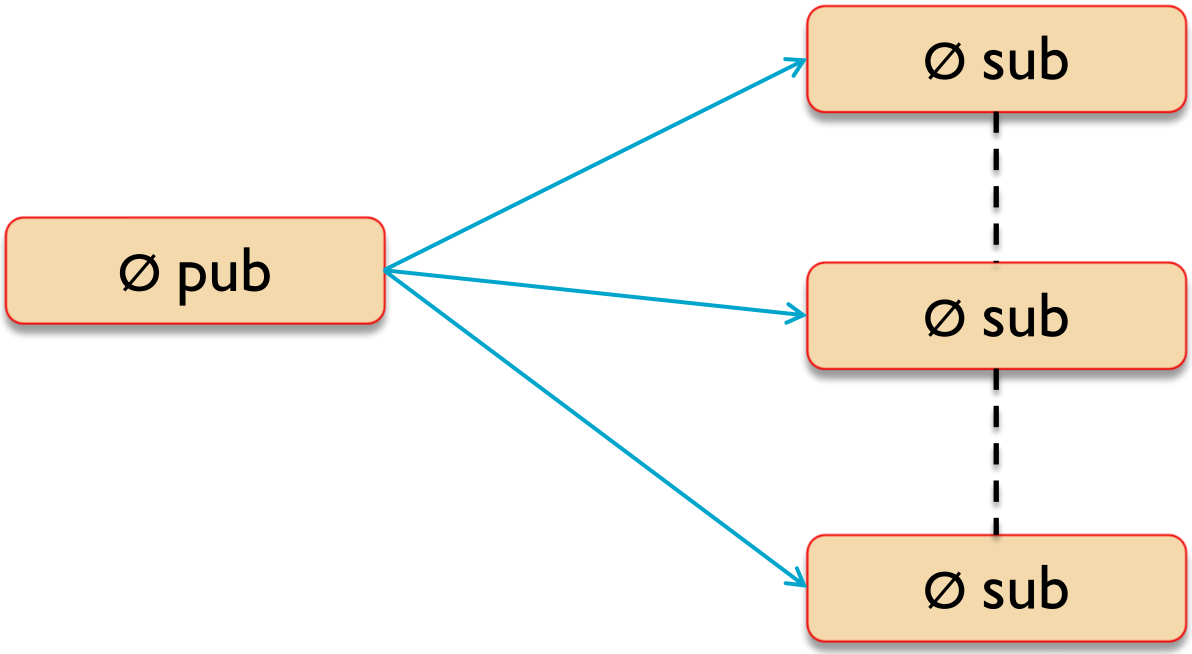
```
const zmq = require("zeromq")
const consumer = zmq.socket("pull")

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

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

4.3.2. Basic Patterns: Publish/Subscribe (pub/sub)

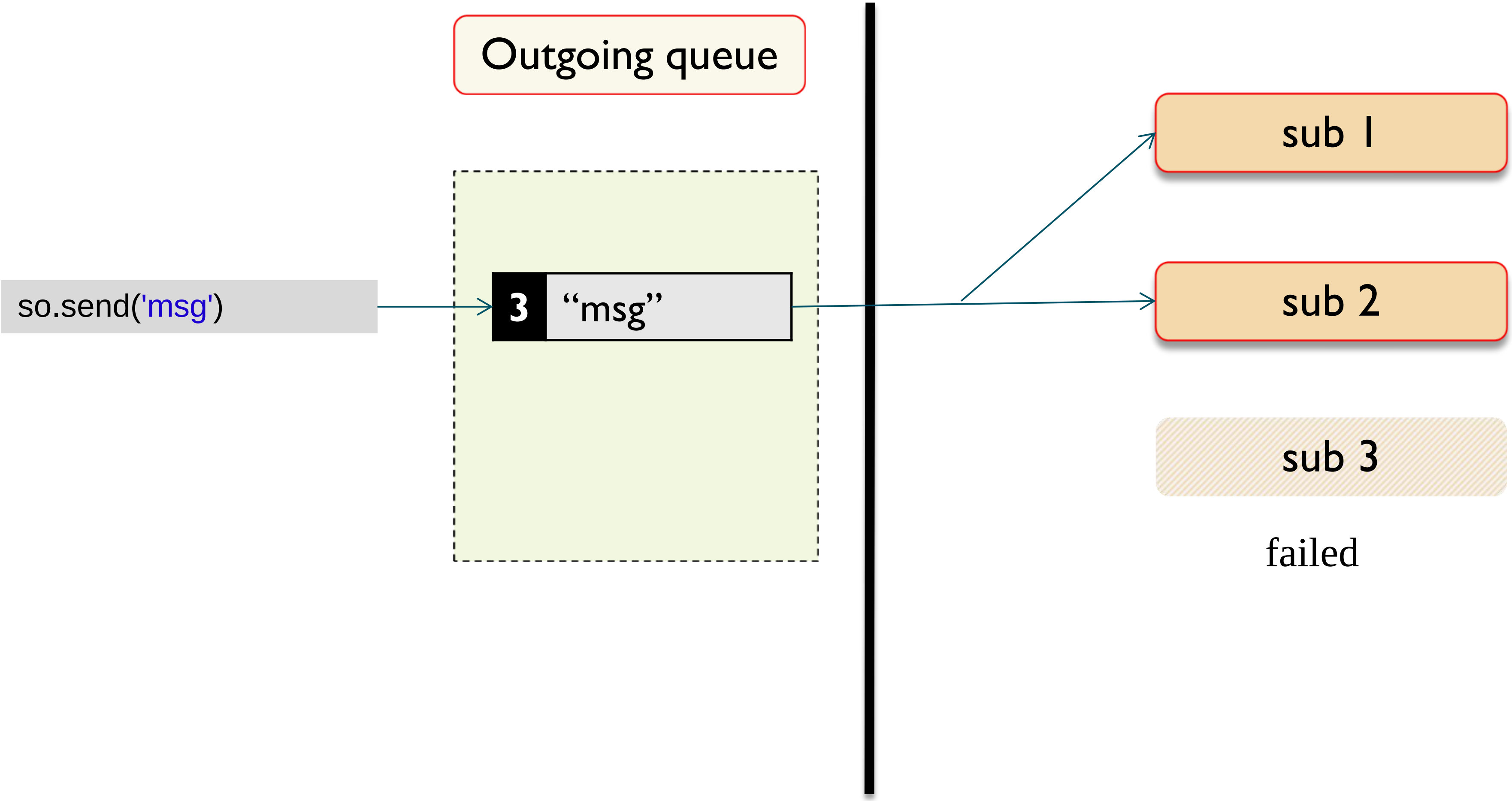
- Pattern implements message broadcasting…



- … with a twist: receivers can decide to subscribe to only some messages
  - Thus multicast

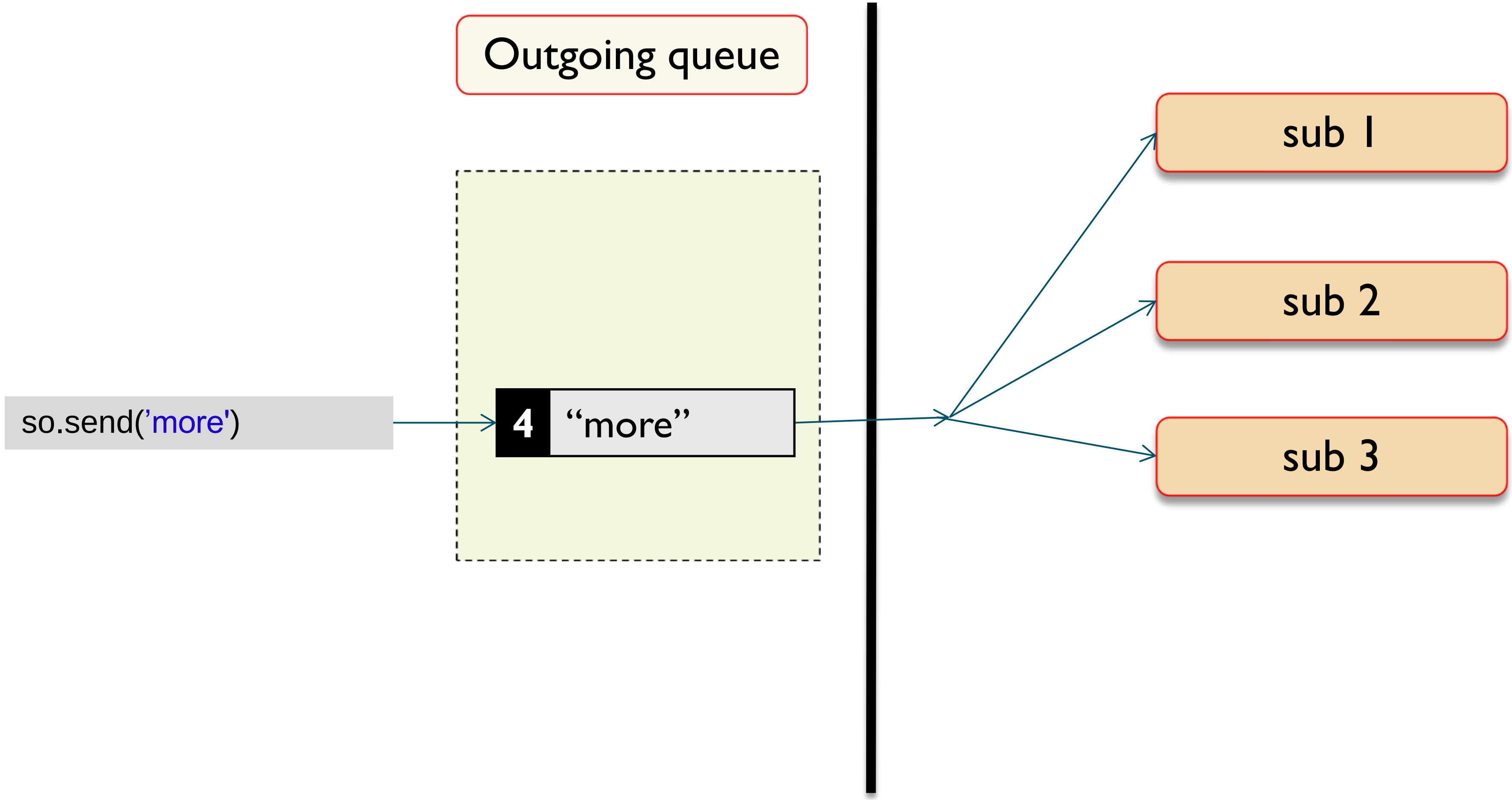
4.3.2. Basic Patterns: pub/sub

- Messages are sent to all connected and available peers



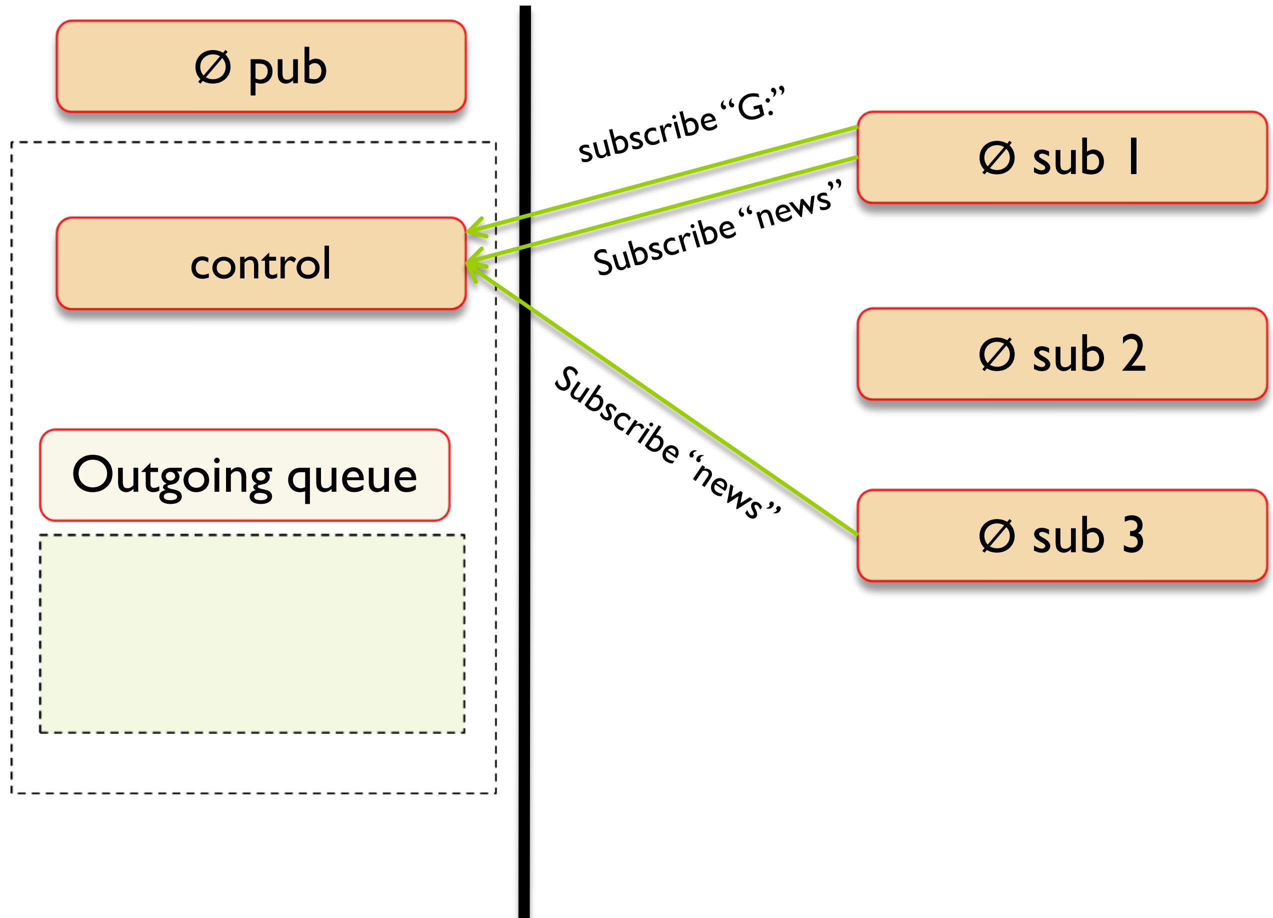
4.3.2. Basic Patterns: pub/sub

- Messages are sent to all connected and available peers



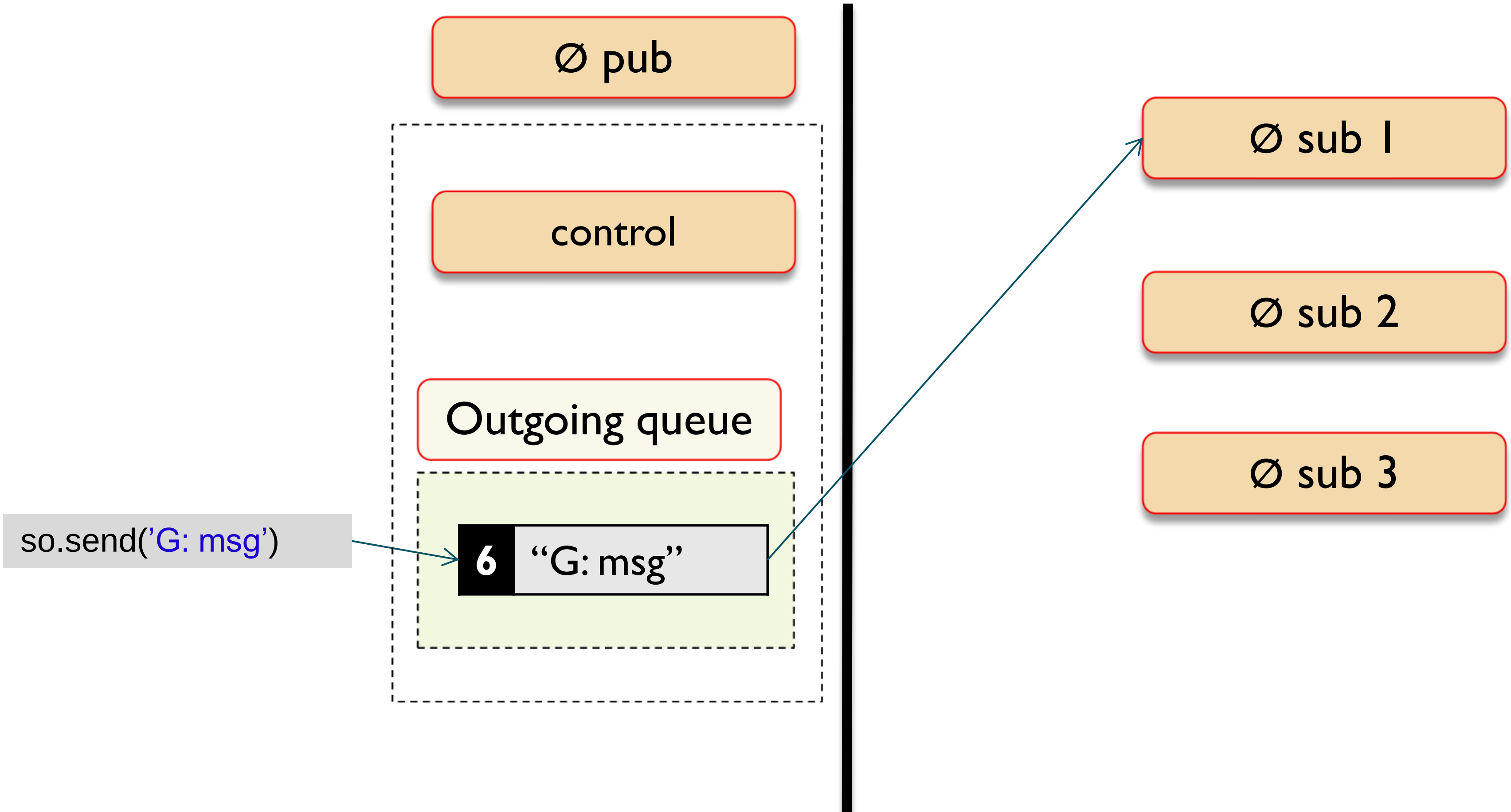
4.3.2. Basic Patterns: pub/sub: Subscribing/filtering

- Subscribers can specify filters as prefixes of messages
  - They can specify several prefixes



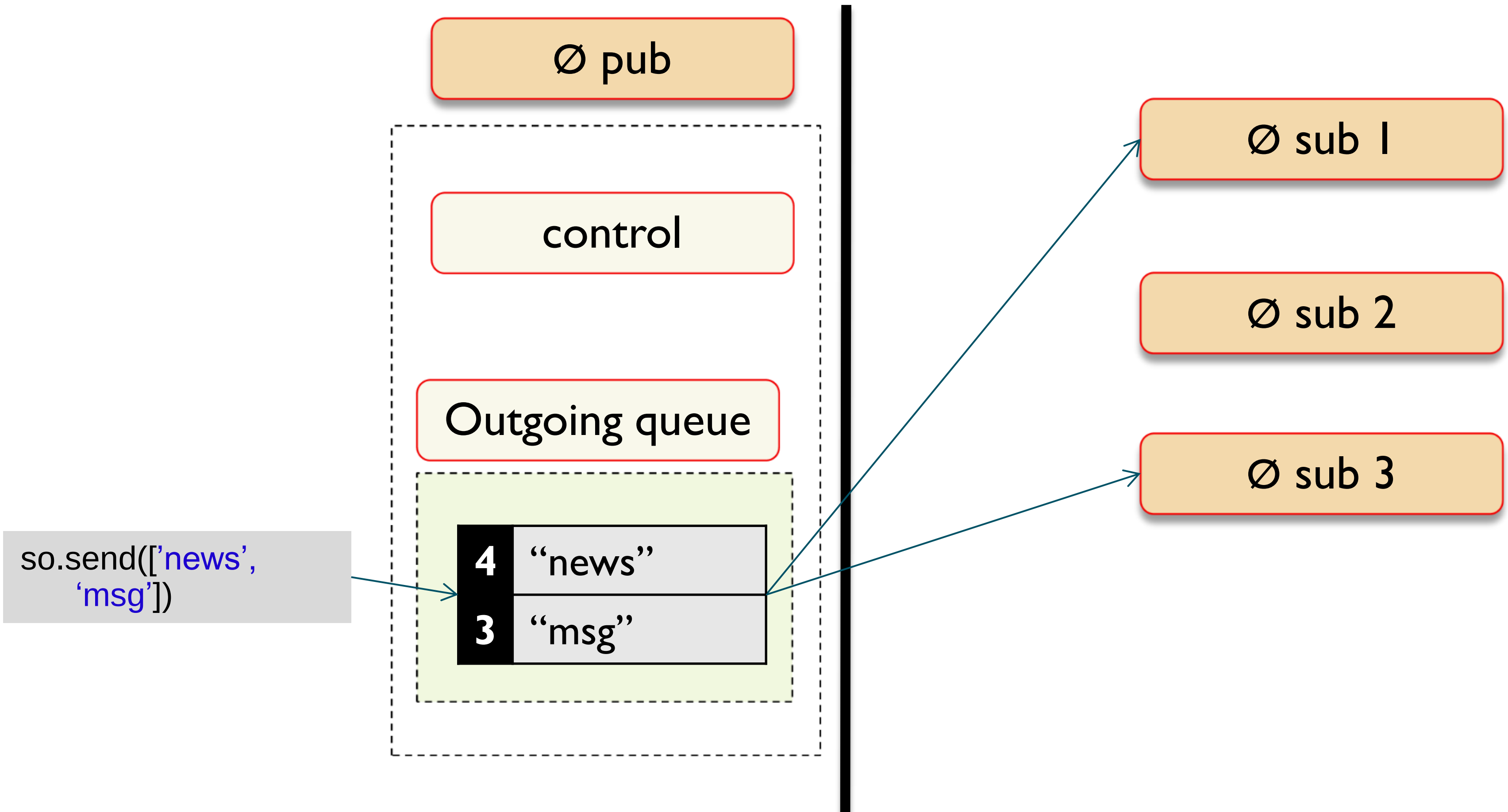
4.3.2. Basic Patterns: pub/sub: Subscribing/filtering

- Subscribers can specify filters as prefixes of messages
  - They will receive only messages with those prefixes



4.3.2. Basic Patterns: pub/sub: Subscribing/filtering

- Subscribers can specify filters as prefixes of messages
  - They will receive only messages with those prefixes



4.3.2. Basic Patterns. pub/sub coding example

```
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)
})
```

Older messages might be lost, if subscriber starts late

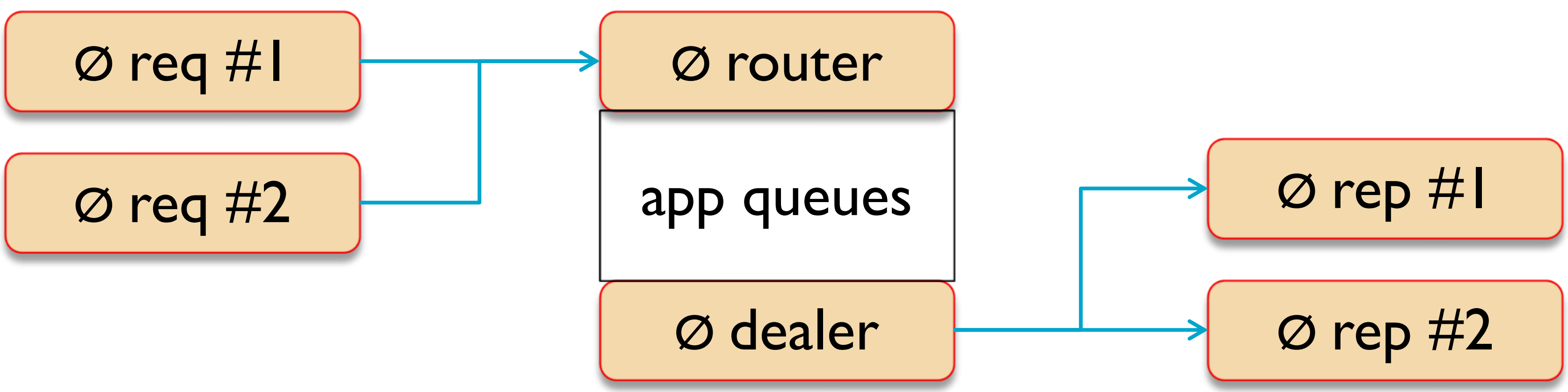
4. ZeroMQ

1. Producing reliable DS Software
2. Middleware
3. Messaging Systems
4. ZeroMQ
    1. Introduction
    2. Messages
    3. 0MQ API
    4. Advanced socket types
5. Other Middleware
6. Conclusions
7. References

4.4. Advanced socket types

1. Dealer
   - Similar to a req, but asynchronous
2. Router
   - Similar to a rep, but asynchronous, and with ability to distinguish its peers (for routing replies)

- Typically found together in the same agent

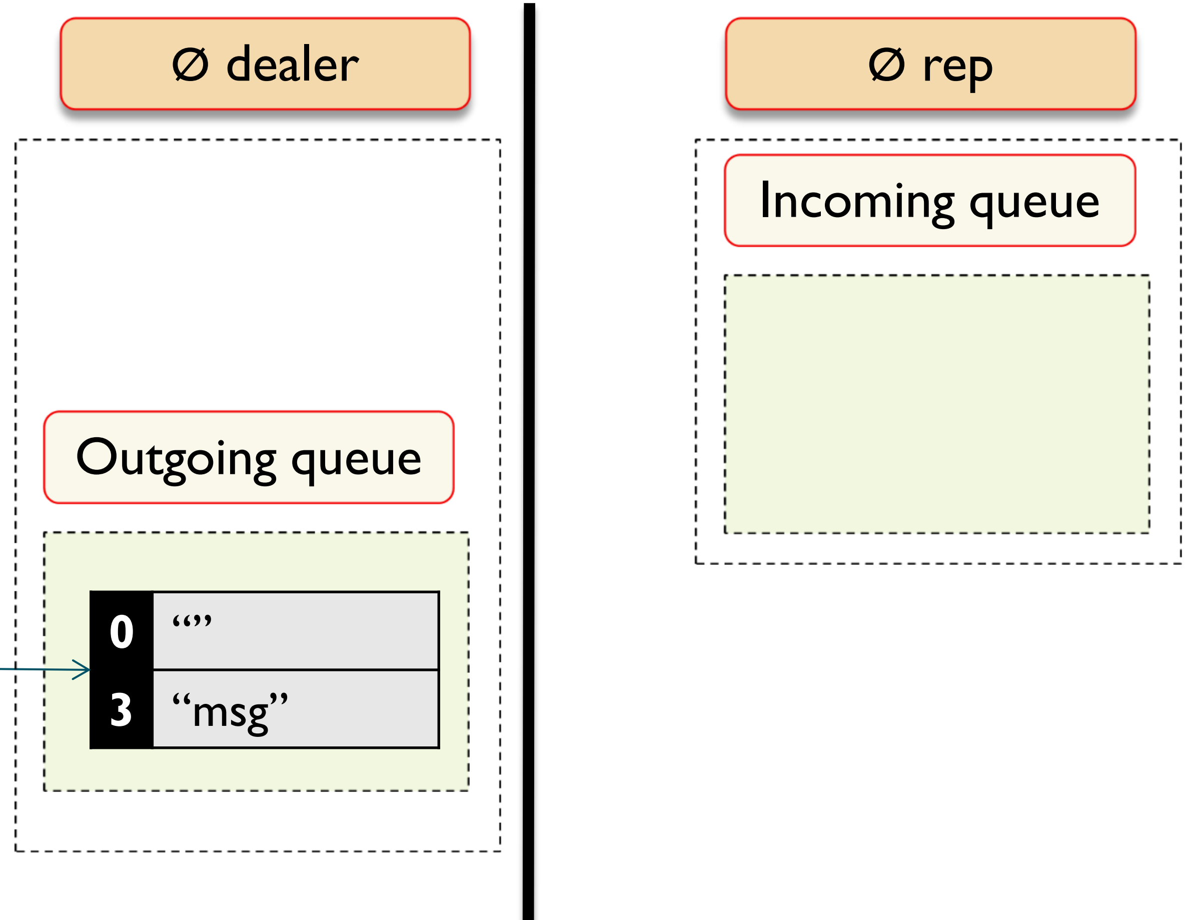


4.4.1. Dealer sockets

- General purpose async socket
- Used frequently as an async **req** socket
  - Does not get blocked by failures of peers
  - BUT, must build a proper request message
    - Empty segment (delimiter) before the actual message body
    - Can prepend the delimiter with any number of segments

4.4.1. Dealer sockets: handling request/reply

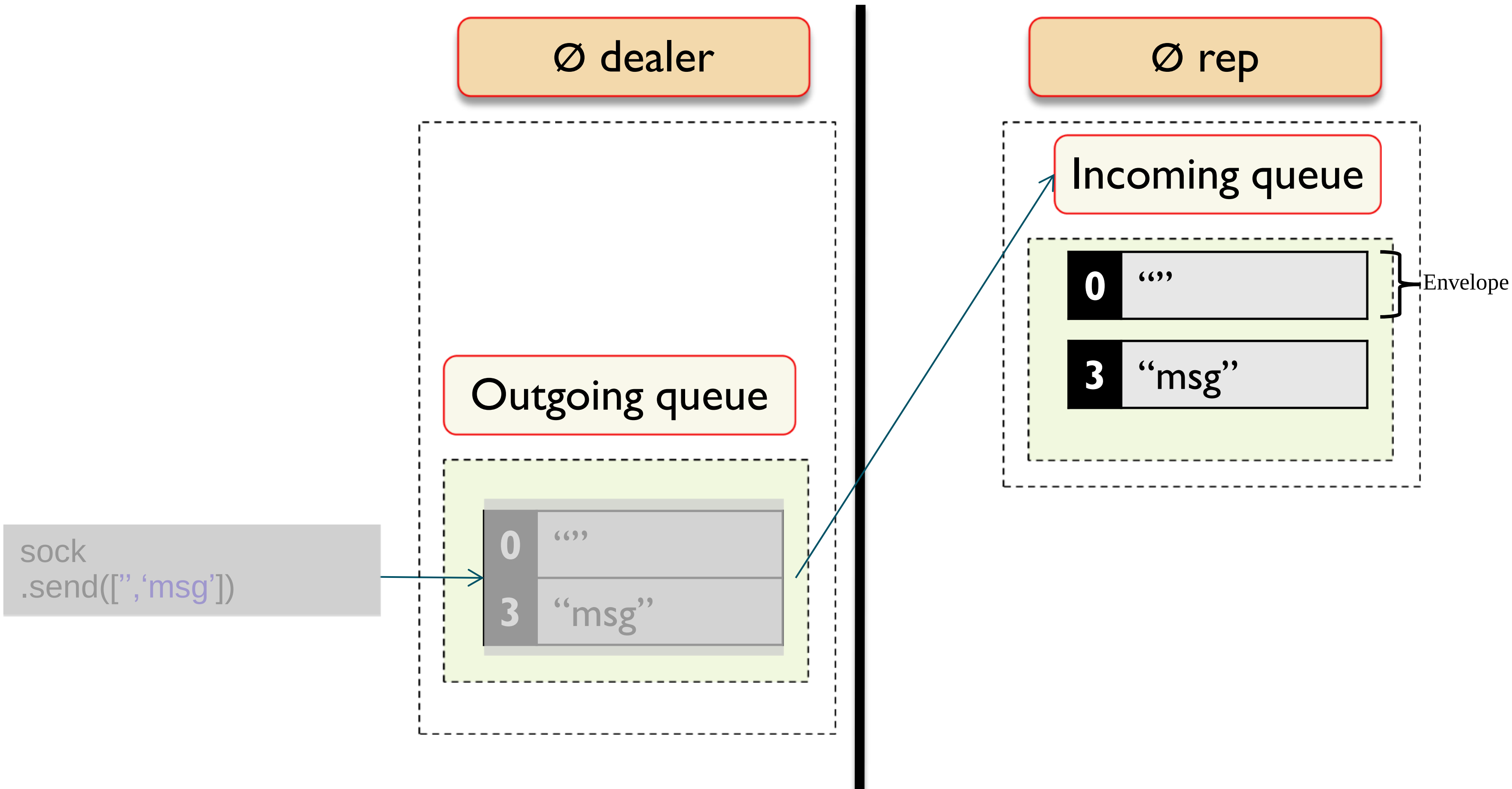
- Delimiter must be prepended to talk to a rep:




```
sock
.send([’’,‘msg’])
```

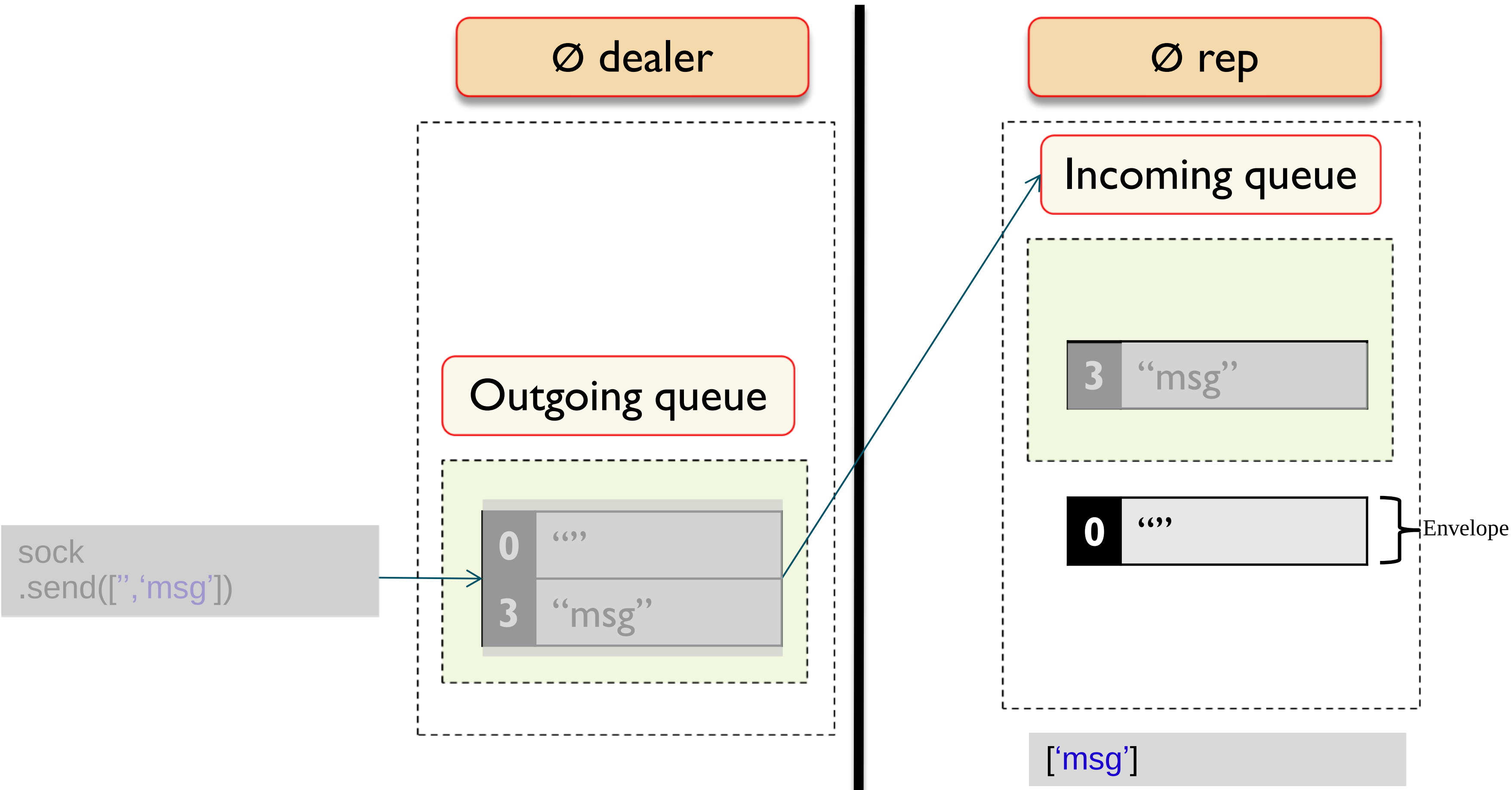
4.4.1 Dealer sockets: handling request/reply

- When received, the rep socket splits the *Envelope*



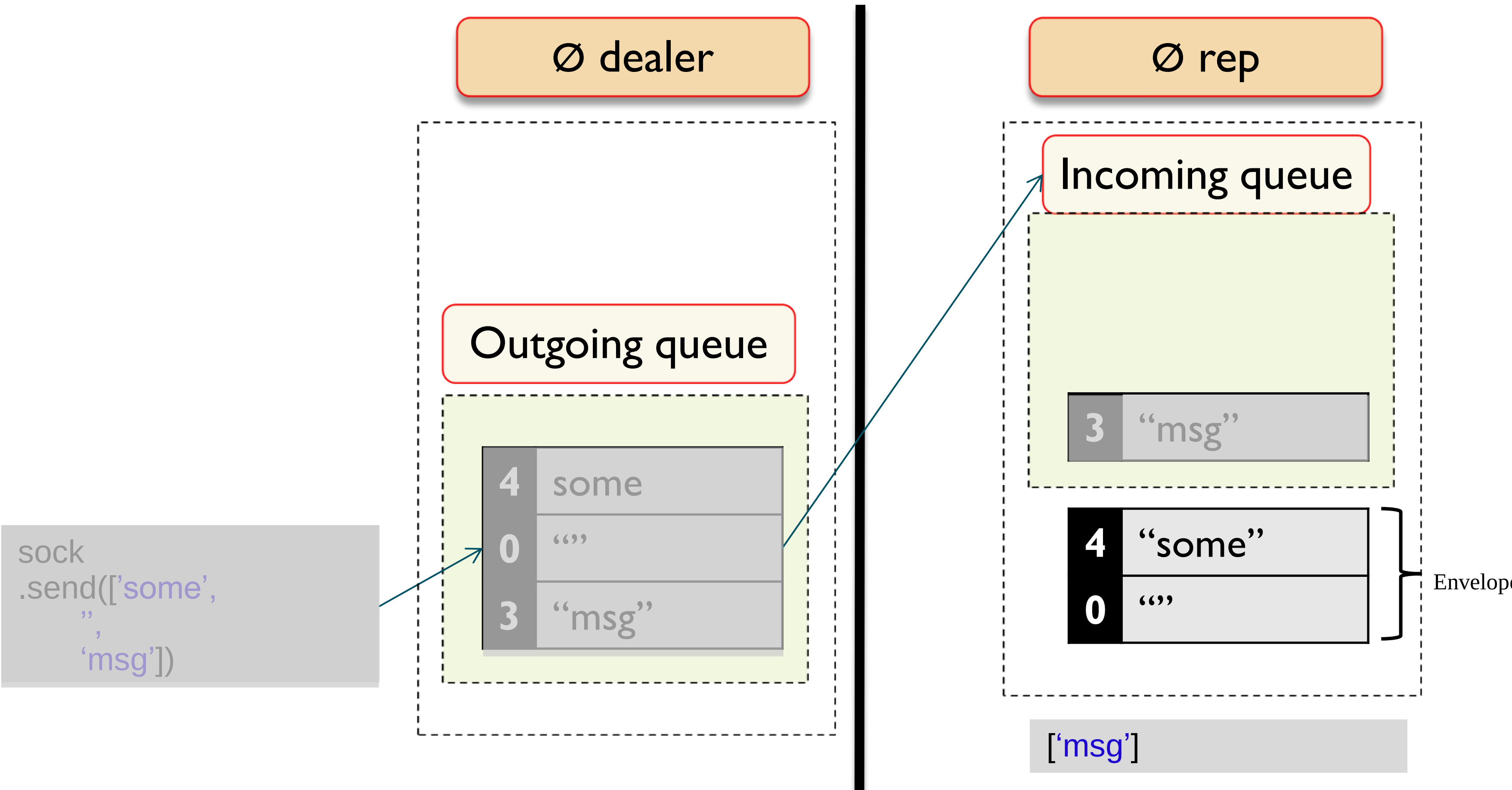
4.4.1 Dealer sockets: handling request/reply

- When received, the rep socket splits the *Envelope*
  - App sees only the rest of the message



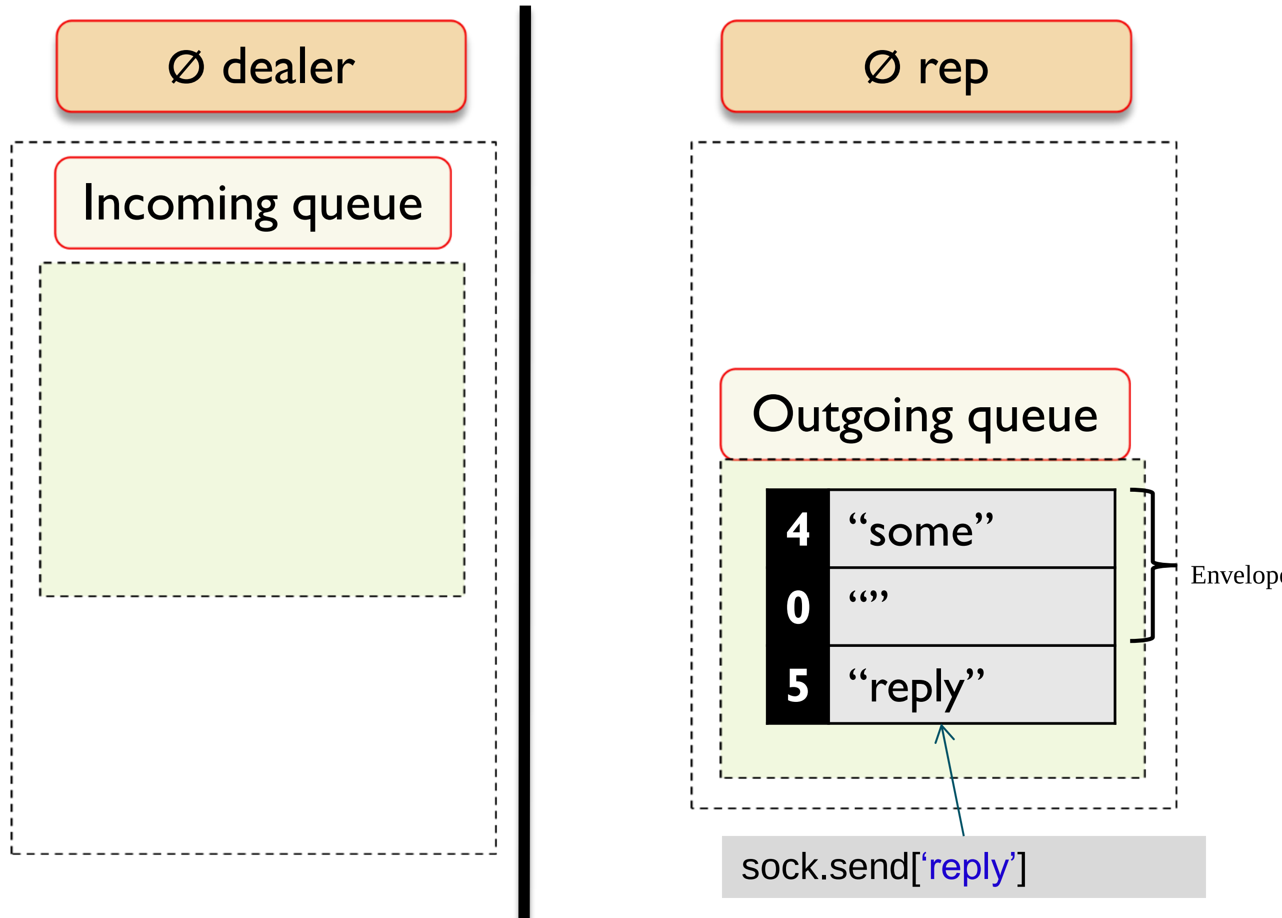
4.4.1. Dealer sockets: Envelopes

- Envelope is more general: All segments up to the first delimiter
  - Envelope is saved by rep…



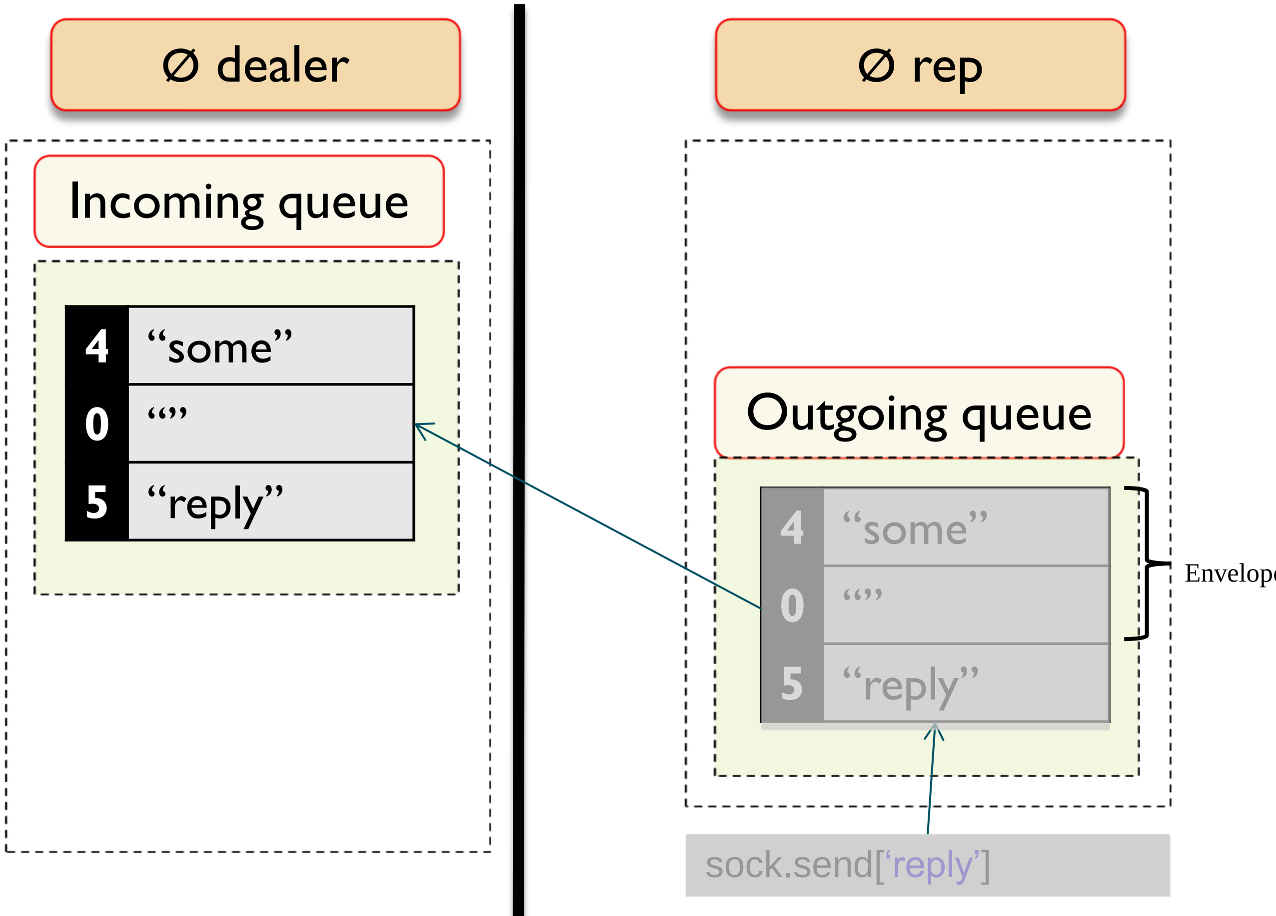
4.4.1. Dealer sockets: Envelopes

- Envelope is more general: All segments up to the first delimiter
  - Envelope is saved by rep… and reinserted to the reply



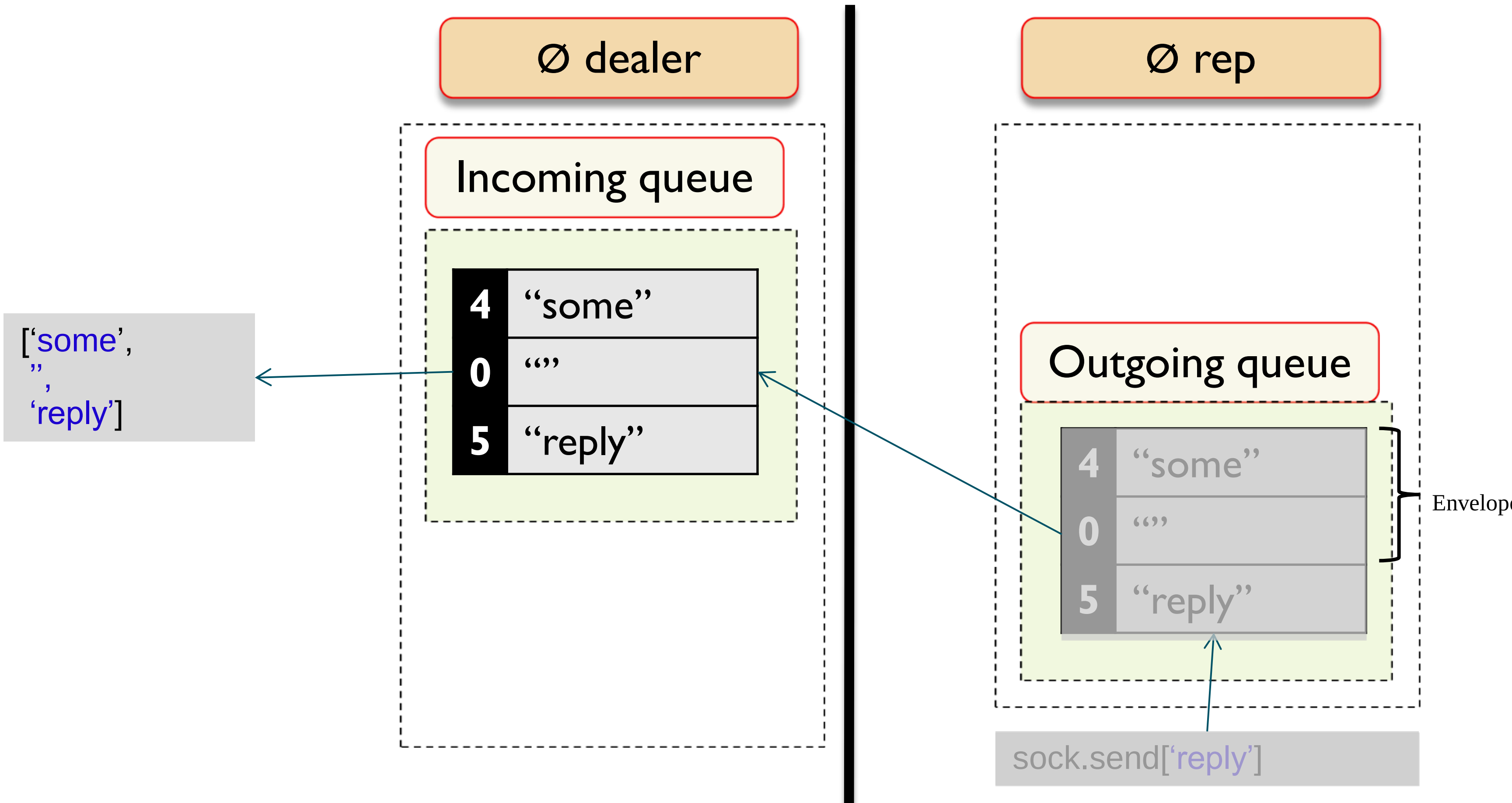
4.4.1. Dealer sockets: Envelopes

- Envelope is more general: All segments up to the first delimiter
  - The composed message is sent back



4.4.1. Dealer sockets: Envelopes

- Envelope is more general: All segments up to the first delimiter
  - And the dealer application gets all of it



4.4.1. Dealer sockets: coding example

```
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)

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. Router sockets

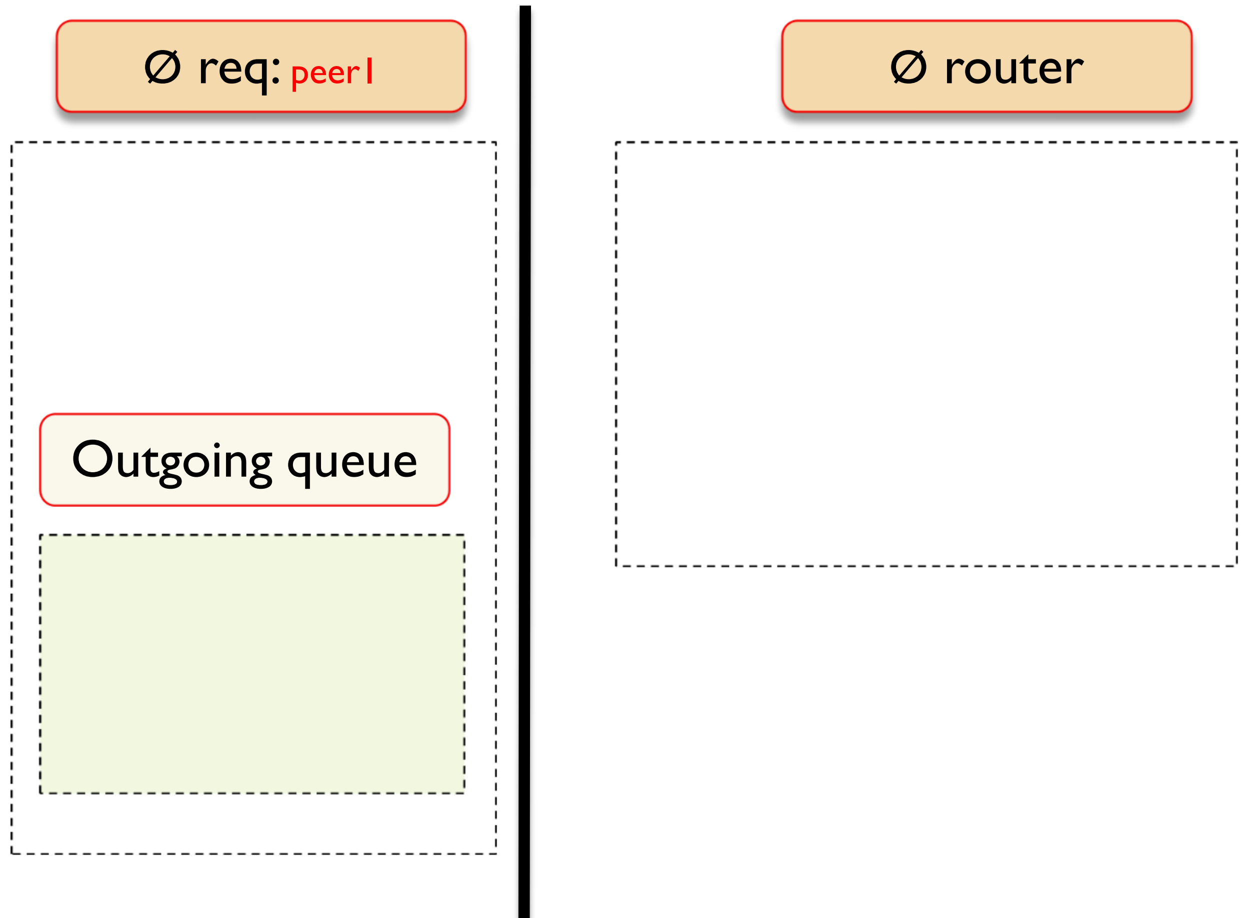
- Async bidirectional sockets
- Allow sending messages to specific peers
  - Assigns an identity to every peer it connects to.
  - The identity is that given to the peer in its program
    - sock.identity = 'my name';
    - When peer has no associated identity socket option
      - Router creates a random identity for the connected peer
      - The created identity lives while the connection is happening
      - When the connection is cut, and re-established the identity changes
    - ID’s are arbitrary binary strings up to 256 bytes long
- When the router socket passes a message to the app
  - It prepends an additional segment with the ID of the sending peer

4.4.2. Router sockets

- When the router socket is sending a message…:
  - It uses its first segment as a connection identity, thus…
    - A router socket maintains a pair of incoming and outgoing message queues **per connection**.
    - Such first segment is used to locate the appropriate connection (i.e., pair of queues). Once it is found…:
      - That first segment is implicitly removed.
        - The programmer does not need to do any thing.
      - The rest of the message is put in the outgoing queue of that connection.
        - This completes the message sending.
  - This allows a trivial router-dealer brokering:
    - The broker process uses a frontend router socket and a backend dealer.
    - Each message received from the router is sent through the dealer.
    - Each message received from the dealer is sent through the router.
    - In both cases, no message segment needs to be modified.

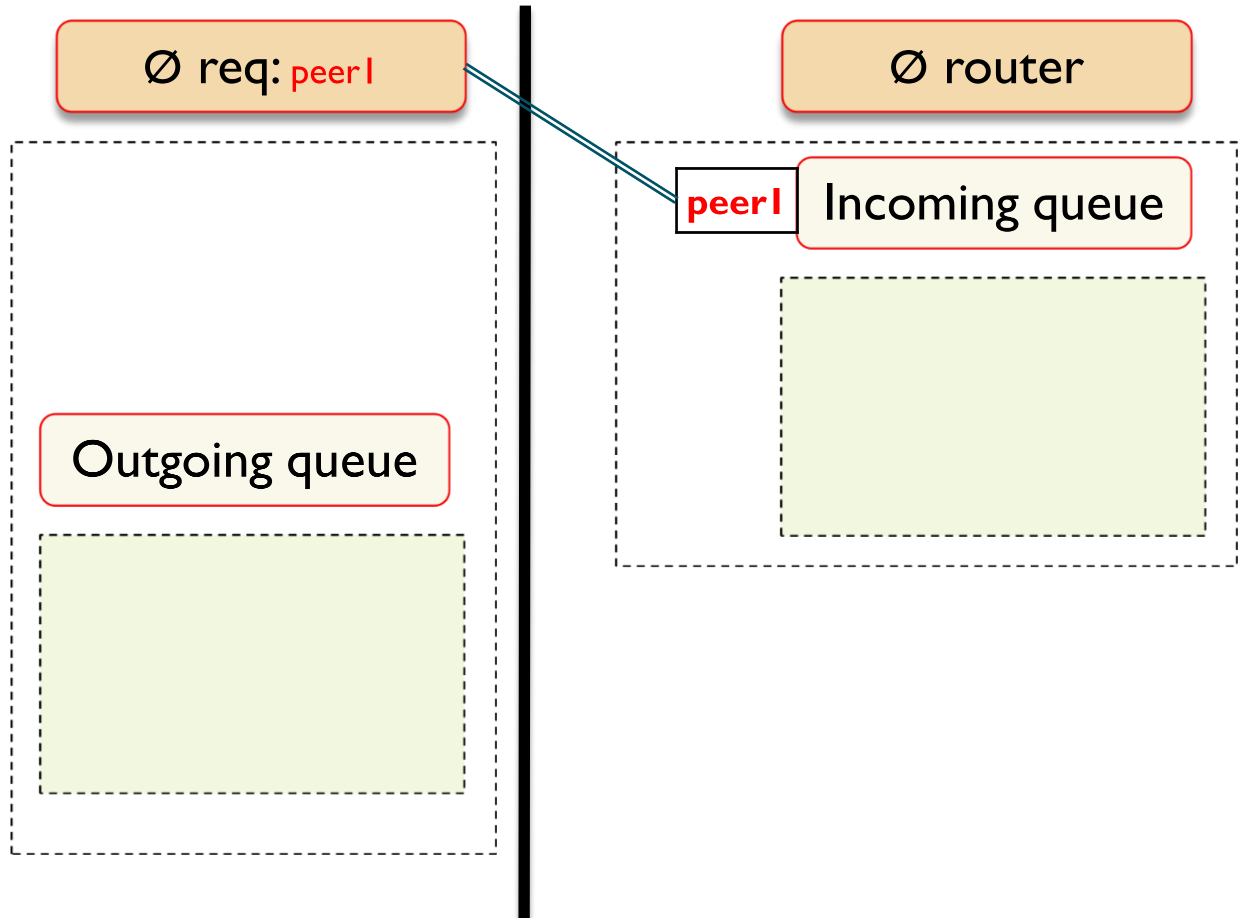
4.4.2. Router sockets: example req peer

- req peer has identity set to "peer1"



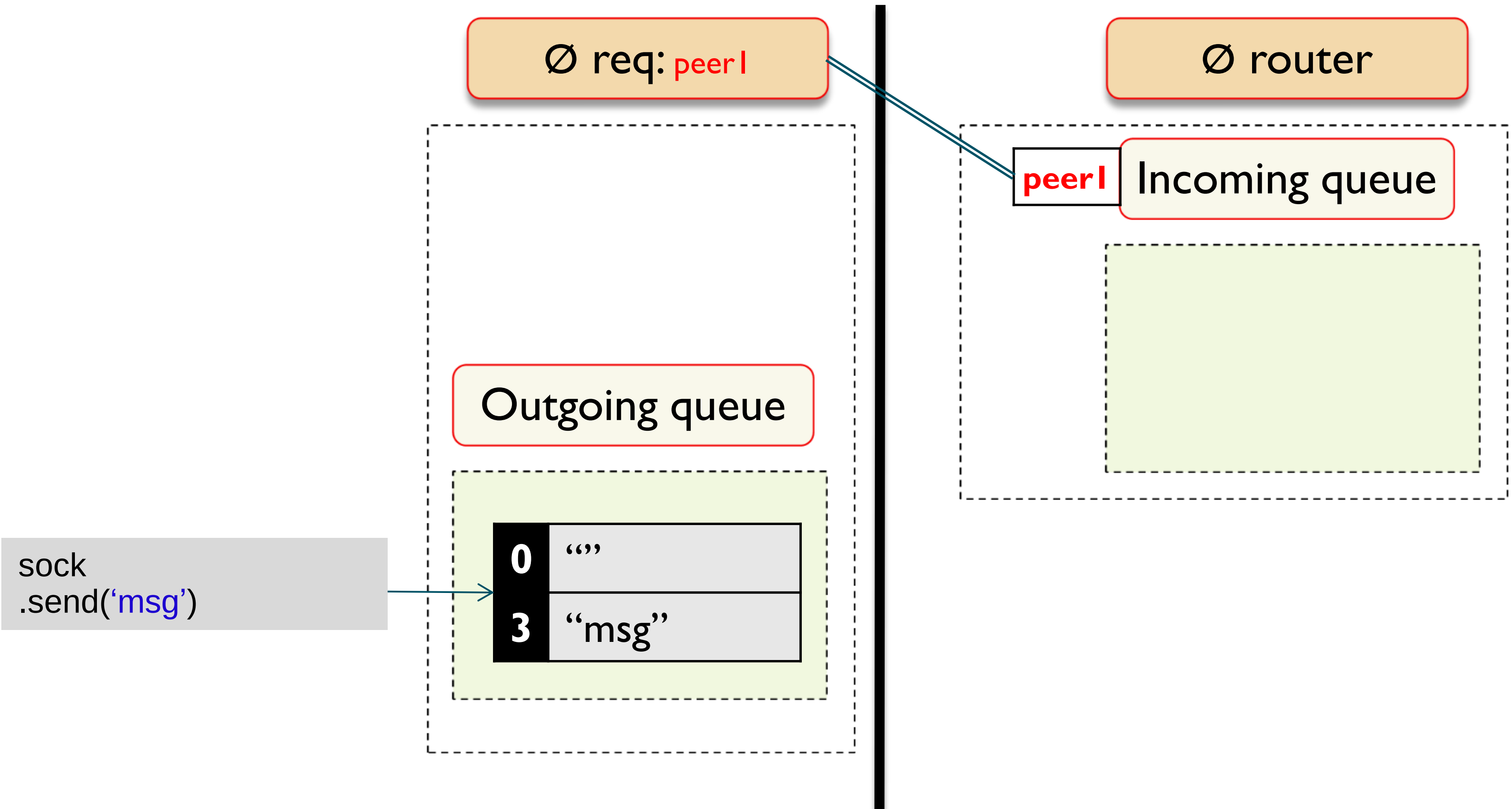
4.4.2. Router sockets: example req peer

- Req peer connects with router
  - Router gets its identity, and stores it, associating in/out queues



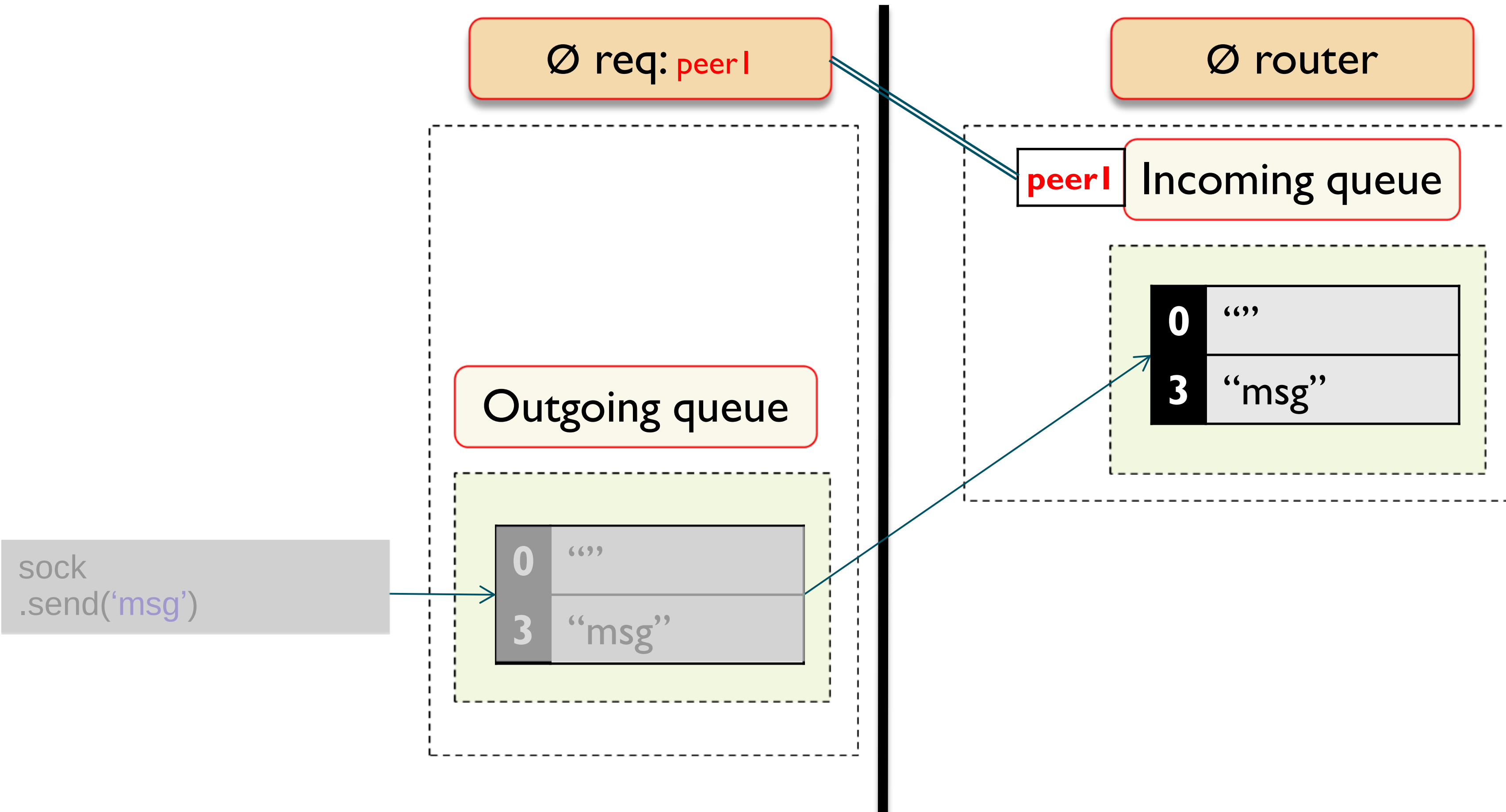
4.4.2. Router sockets: example req peer

- Req app sends a message
  - Req socket adds delimiter…



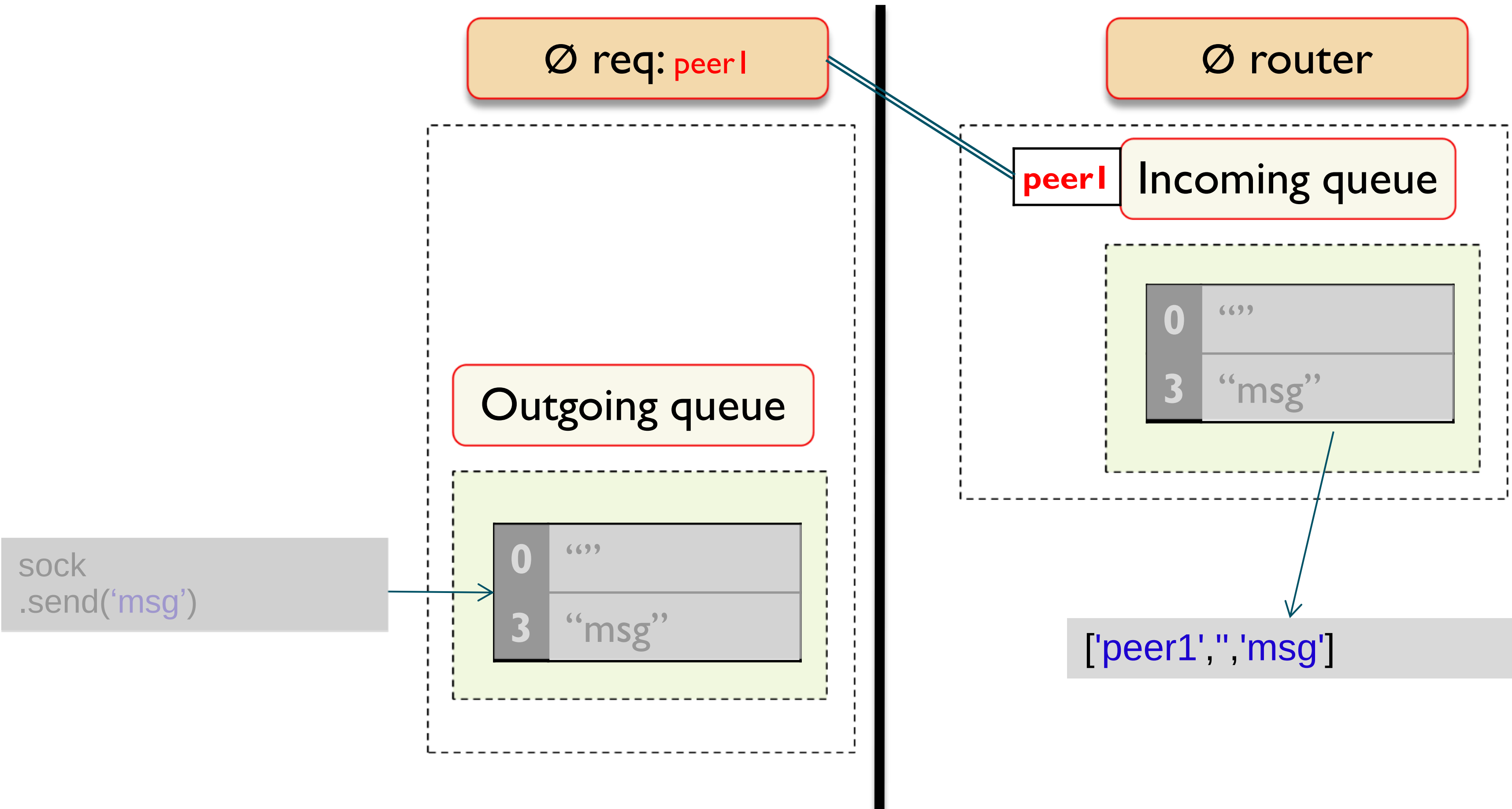
4.4.2. Router sockets: example req peer

- Req app sends a message
  - Req socket adds delimiter… and sends it



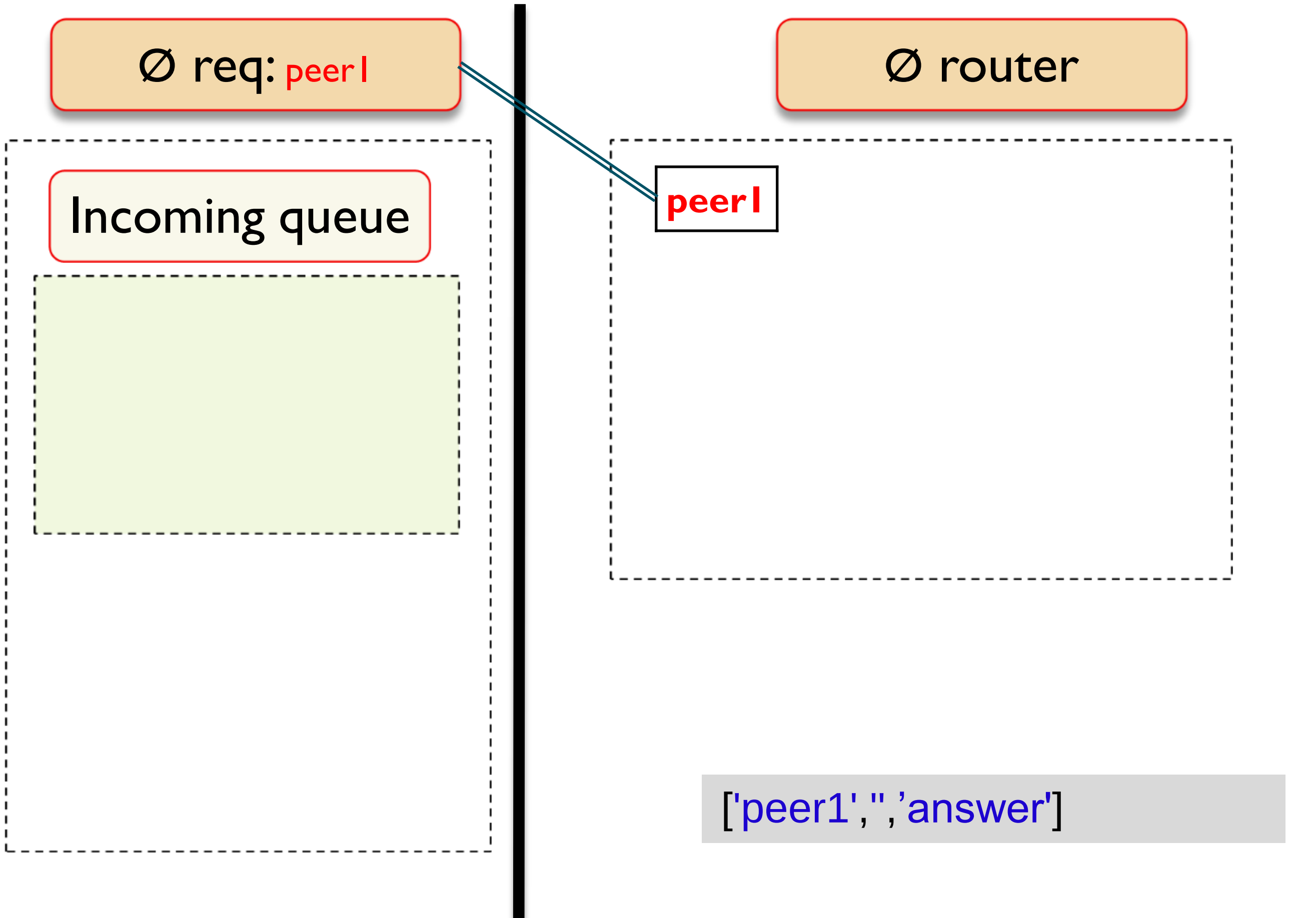
4.4.2. Router sockets: example req peer

- Router socket gives message to its app
  - With the identity of the sender prepended in a segment



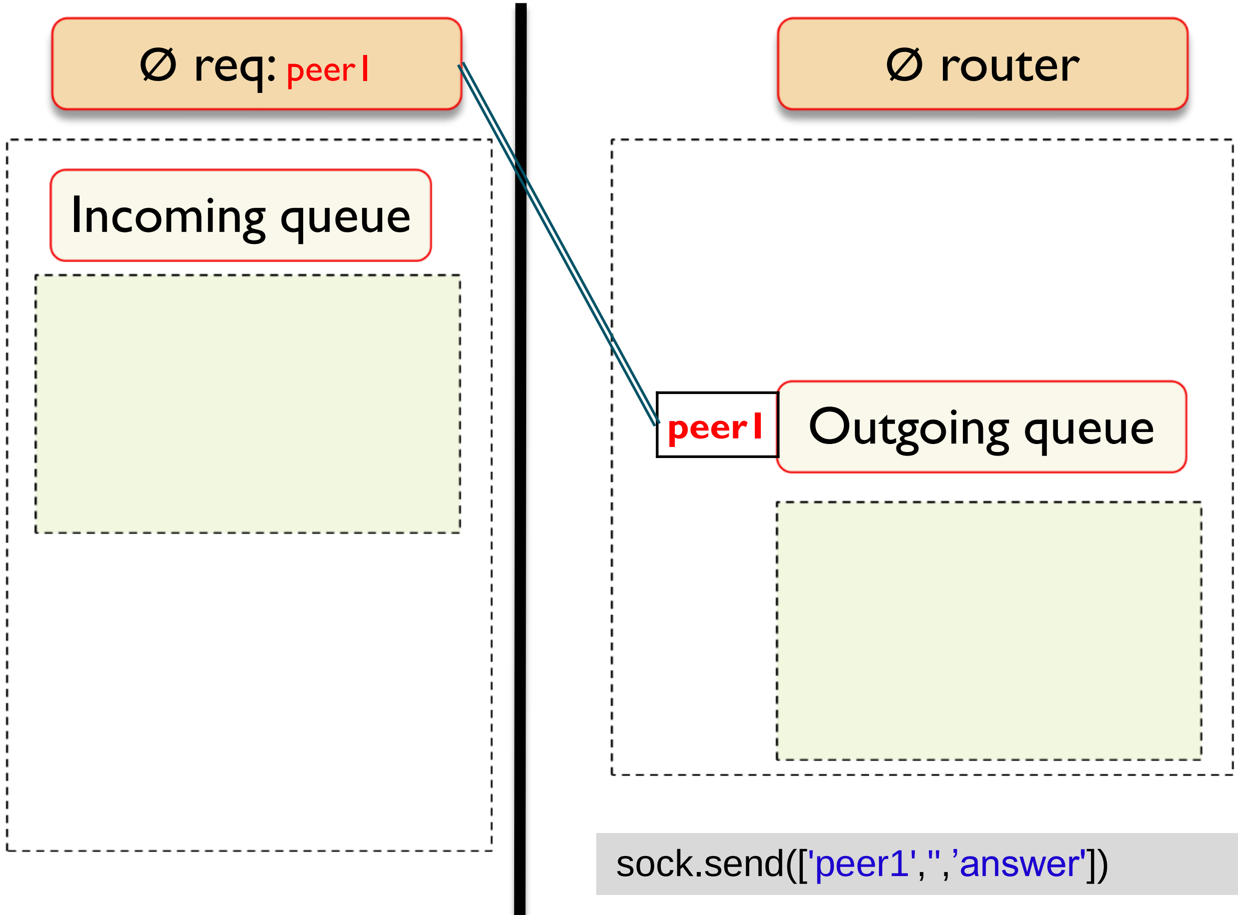
4.4.2. Router sockets: example req peer

- Router app creates an answer, composing the reply message
  - First segment contains identity of peer to receive answer



4.4.2. Router sockets: example req peer

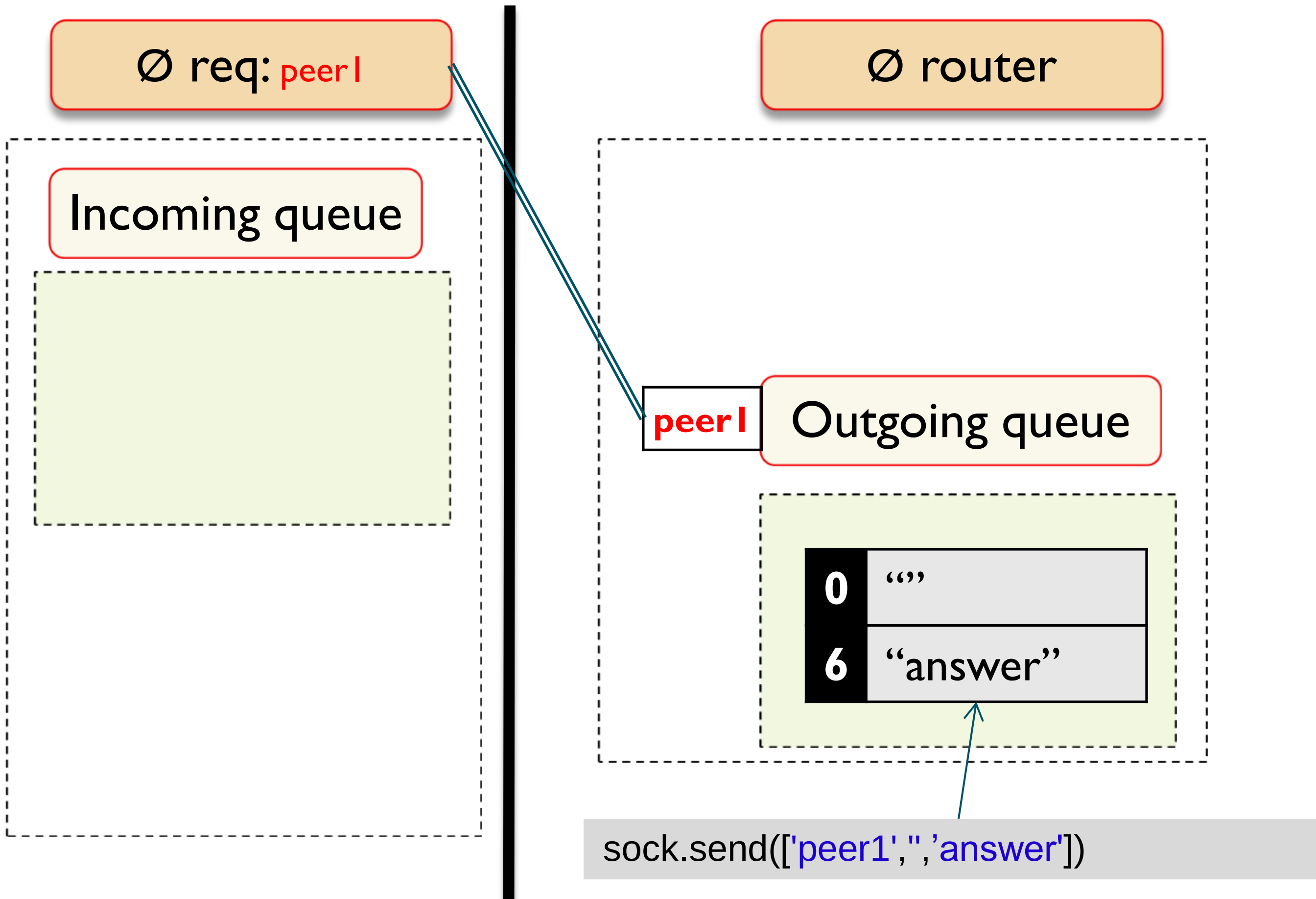
- Router app sends message
  - Router socket selects the output queue based on the identity




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

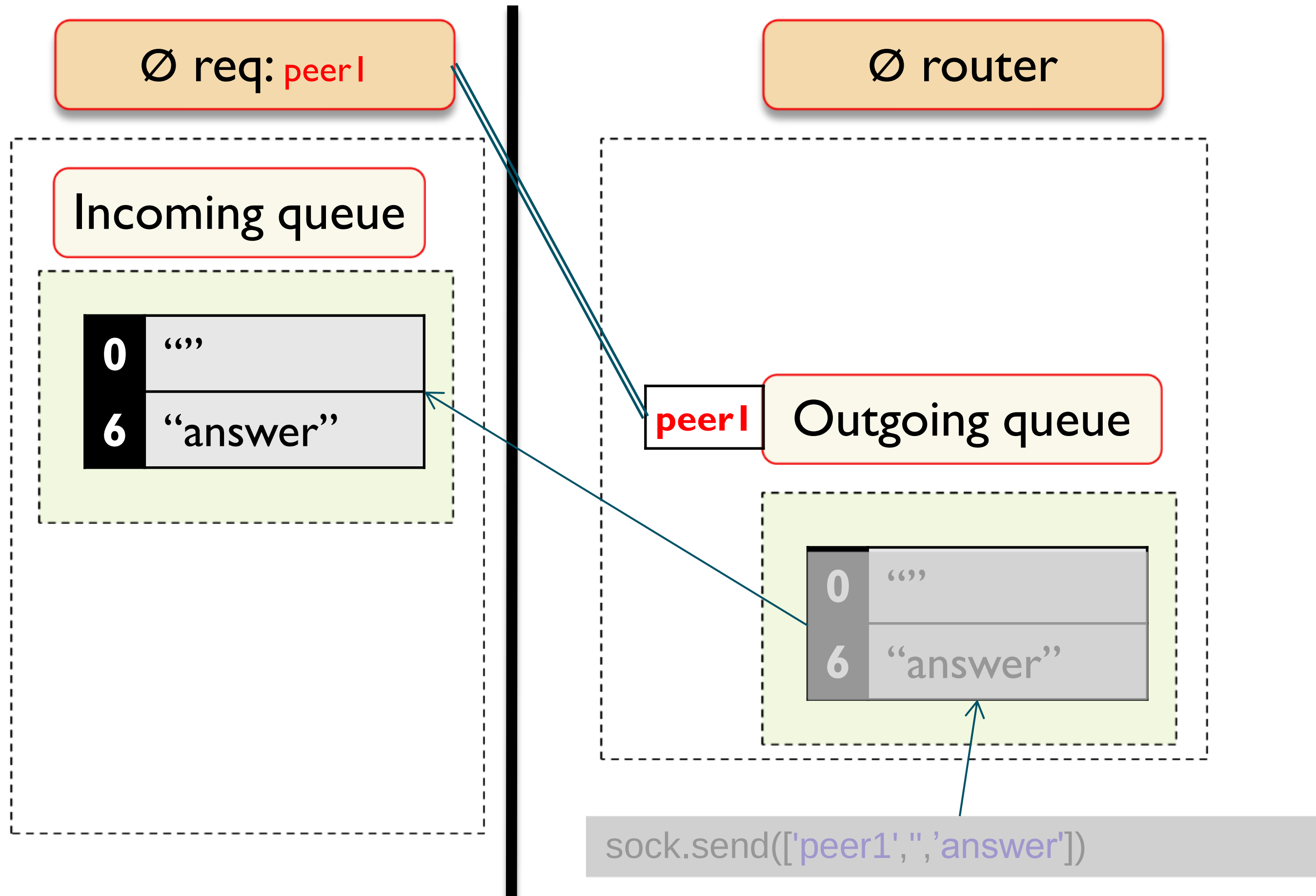
4.4.2. Router sockets: example req peer

- Router socket strips the identity segment
  - Leaves the rest of the message for sending…



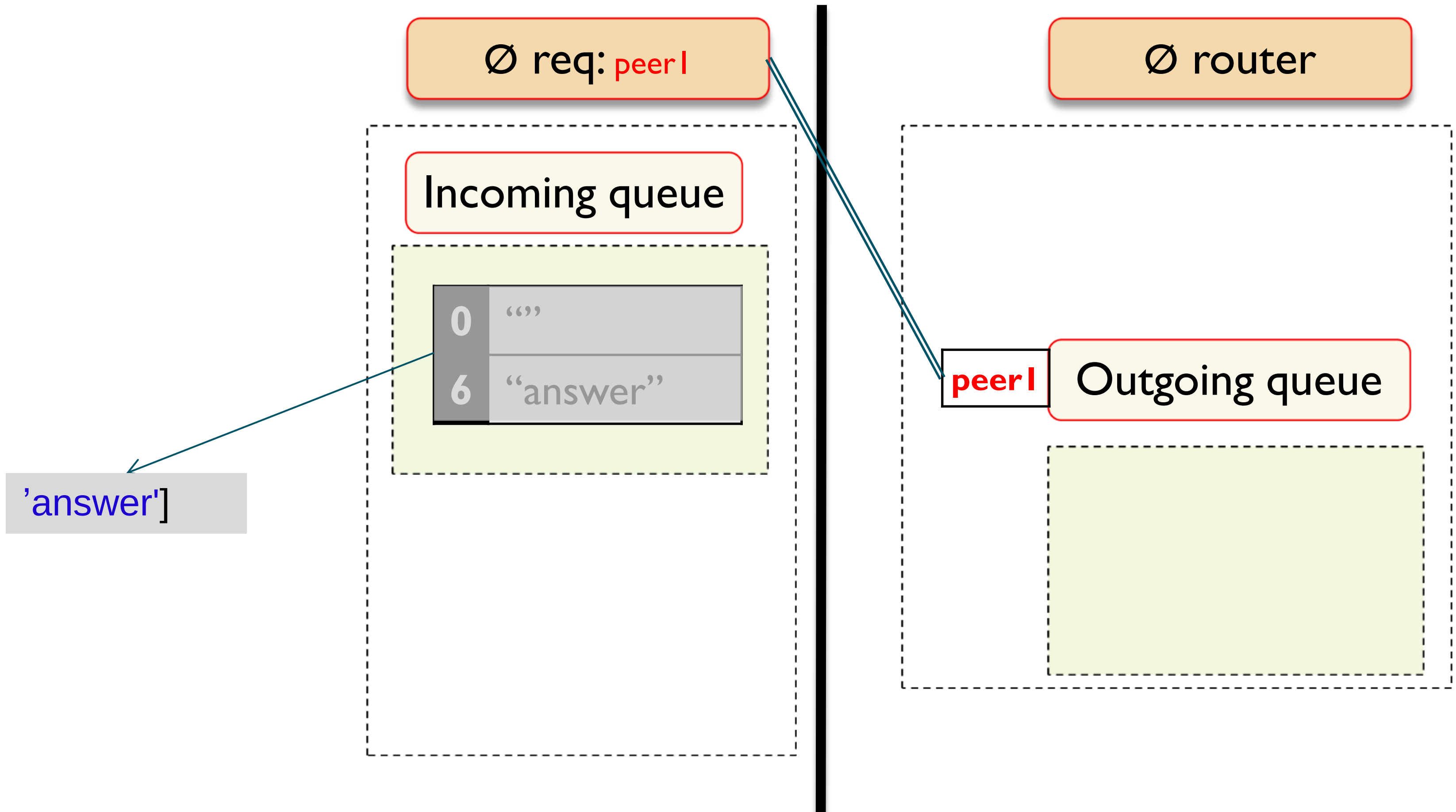
4.4.2. Router sockets: example req peer

- Router socket strips the identity segment
  - Leaves the rest of the message for sending…and sends it



4.4.2. Router sockets: example req peer

- The req socket gives it to its application
  - Eliminating the delimiter segment



Index

1. Producing reliable DS Software
2. Middleware
3. Messaging Systems
4. ZeroMQ
5. Other Middleware
6. Conclusions
7. References

5. Other Middleware

- Event management
  - Often included in messaging systems
    - PUB-SUB pattern
  - E.g. JINI
- Security
  - Authentication
    - A third party warrants the identity of a party
    - E.g. OpenID
  - Authorization
    - A third party authorizes a request
    - E.g. OAuth
  - Integration with other protocols
    - E.g. SSL/TLS and HTTPS
- Transactional support
  - Coordination of distributed atomic state change
    - Fault resilient

Index

6. Conclusions

- DS complexity needs support of code + services
- Standards simplify by making approaches familiar
- Middleware implements common solutions to problems
  - As layers below application code and above communications
- Main middleware target
  - Communication tasks
  - Service request
- Main approaches
  - Messaging
    - Transient/Persistent
    - Broker-based/Brokerless
- Other Middleware
  - Security
  - Transactions

Index

1. Producing reliable DS Software
2. Middleware
3. Messaging Systems
4. ZeroMQ
5. Other Middleware
6. Conclusions
7. References

7. References

- http://zguide.zeromq.org/page:all
  - Can be read online.
  - Has a PDF version
  - The site contains additional information