TSR - **0MQ** Reference documentation

Course 2019/20

Contents

1. 0MQ

- CSD introduced MOM (Message Oriented Middleware)
  - Indirect communication → asynchrony, persistence, weak coupling
  - JMS was studied as an example
- 0MQ is another example of MOM, however it is quite different from JMS:

  - No message routing broker, with a socket-like API
    * No need to set up any servers
    * Offers sockets for message send/receive
    * Endpoints identified with URLs
    * Weak persistence (RAM-based queues)
  - Async I/O (event-driven)
  - Widely available (many OSs and language runtimes)
    * Free library (open source), to be linked to application

1.1. Basic properties

- Efficient (tradeoff reliability/efficiency)
  - Weak persistence (RAM-based queues)
  - 0MQ sockets have associated message queues

message queue	description	events
incoming (receive)	keeps arriving messages	generates "message" event
outgoing (send)	keeps messages to be sent	

- Defines several message exchange patterns
  - eases development
- Useful on several levels → same code to communicate threads within a process, processes within a machine, machines in an IP network.
  - By only changing the transport specification in the URL
  - We place the focus on inter-machine communications over TCP

1.2. Messages

- Transparent buffer management
  - Manages the message flow between the queues attached to the 0MQ sockets and the transport level.
  - Message content transparent to their management
- Atomic message delivery (all or nothing)
  - Can be structured in multiple segments (segment = counted byte buffer)
    * 1 segment: `send('hola')` → `4 h o l a`
    * 3 segments: `send(['hola','','Ana'])` → `4 h o l a 0 3 A n a`

sending messages	receiving messages
`send(['uno','dos'])`	`sock.on("message", (a,b)=>{..})` `a` holds `'uno'`, `b` holds `'dos'`
`send(msg)`	`sock.on("message", (...m)=>{..})` segments of `msg` in vector `m`

1.3. Connections

- Automatic connection/reconnection management

  - One agent `bind`s, the rest `connect`.
    * In any order
    * All agents must meet at some endpoint
  - Binding to an already in-use port raises a run-time exception

- Connection/Reconnection over TCP
  - bind.- The IP address must belong to one of the machine's interfaces
    * `s.bind('tcp://*:9999')`
  - connect.- must know the IP address of the `bind`socket.
    * `s.connect('tcp://127.0.0.1:9999')`
    * Can use domain names instead of IP addresses

- `close` is implicitly called when an agent terminates.

- Not only 1:1 communications
  - n:1 → eg. n clients (each calls one `connect`), 1 server (`bind`)
  - 1:n → eg. 1 client (calls n `connect`, one to each server), n servers (each one calls `bind`)

1.4. 0MQ in nodejs

- Module installation: `npm install zeromq@4`
- Syntax

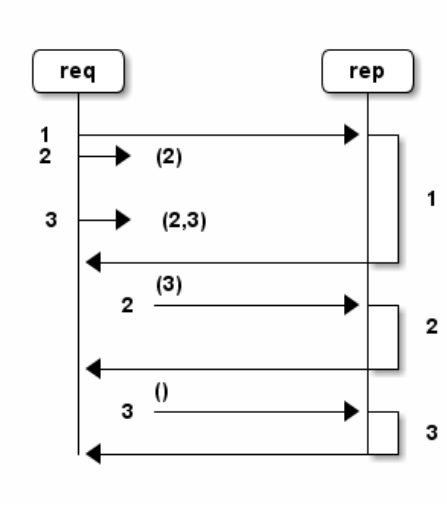
```
const zmq = require('zeromq')              // imports module
let zsock = zmq.socket('tipoSocket') // creates socket (several types available)
zsock.bind("tcp://*:5555")           // bind to port 5555
zsock.connect("tcp://10.0.0.1:5555") // or connect (host 10.0.0.1, port 5555)
zsock.send([..,..])                  // send
zsock.on("message", callback)        // event-driven, async, receive
zsock.on("close", callback)          // handling connection closing
```

1.5. 0MQ Sockets

- There are several kinds of sockets to implement common communication patterns in Distributed Systems.
  - Each pattern with its own requirements → requires concrete socket types
  - Many use specific socket pair kinds
  - Some other patterns allow mixing several simple socket kinds

- When the pattern to use is known, 3 steps to design a distributed application:
  1. Decide what socket combination is needed and on which agents to place them.
  2. Define message formats
  3. Define how agents react to each received message

2. Client/Server pattern: req/rep



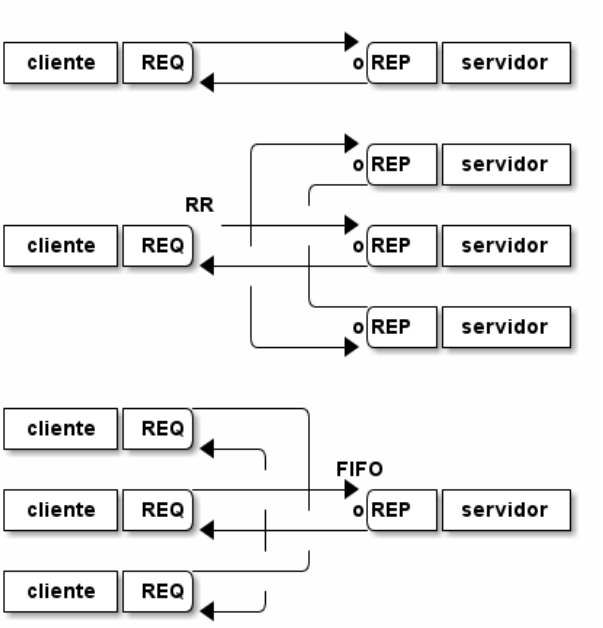
- The server uses `rep` socket
  - Calls `bind`
  - Receives with `s.on('message', callback)`
- Client uses `req` socket
  - Calls `connect`
  - Send with `s.send(msg)`
- It is a **synchronous communication pattern**
  - If a client sends n requests, the second, third,... stay in their local queue until a response to the first request is received
  - Totally ordered request/response pairs

2.1. Connections

Always

- Request/Reply
- In the figures, symbol `o` stands for a `bind`

Several connection strategies are possible (figure).



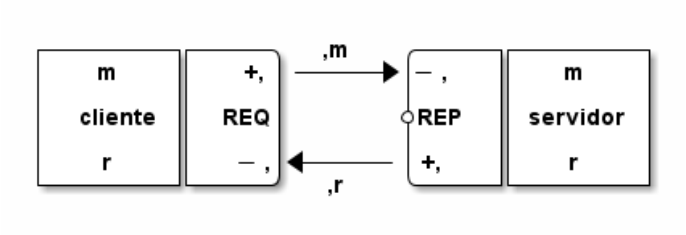
- Connection `1:1`
- Connection `1:n`
  - Round-Robin (RR)
  - No parallelism
- Connection `n:1`
  - Typical server
  - FIFO queue (fair queue)
    - No client starvation

2.2. Message format

1. Client application sends message `m`
2. `req` adds a first empty segment (delimiter) to the message
   - Delimiter → empty segment, separating the 'envelope' from the body of the message

  - Al segment up till the first delimiter are referred to as the 'envelope': can be used to associate metadata to the message (e.g., who must receive the reply)
  - The segments following the first delimiter constitute the body of the message (data)
- In the figure, the delimiter is represented as a ,

3. `rep` socket stores the envelope, delivering the message body to the application
4. When `rep` sends back the reply, it prefixes the envelope back.
5. When `req` receives the reply message, it drops the delimiter, delivering the body to the application



2.3. 1:1 Client/Server Example

cliente.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9999')
s.send('Alex')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

Execution

```
> node cliente.js
Recibido: Hola, Alex
```

servidor.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9999')
s.on('message', (nom) => {
  console.log('Nombre: '+nom)
  s.send('Hola, '+nom)
})
```

Execution

```
> node servidor.js
Nombre: Alex
```

2.4. 1:n Client/Server example

cliente.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.connect('tcp://127.0.0.1:9999')
s.send('Alex1')
s.send('Alex2')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
})
```

servidor1.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9998')
s.on('message', (nom) => {
  console.log('Serv1, '+nom)
  s.send('Hola, '+nom)
})
```

servidor2.js

```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9999')
s.on('message', (nom) => {
  console.log('Serv2, '+nom)
  s.send('Hola, '+nom)
})
```

2.5. n:1 Client/Server example

client1.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.send('uno')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

client2.js

```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.send('dos')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

server.js

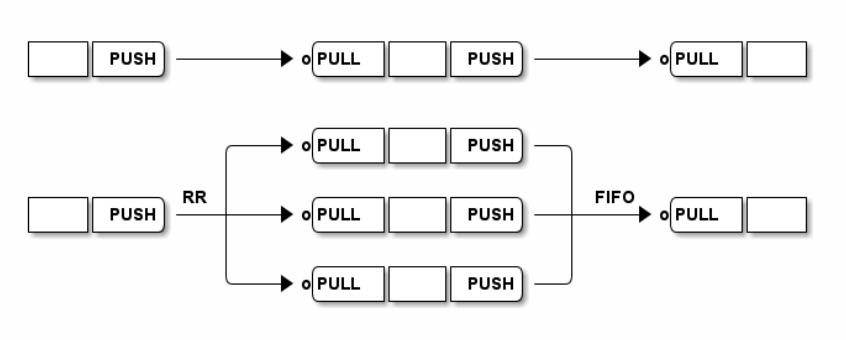
```
const zmq = require('zeromq')
let s = zmq.socket('rep')
s.bind('tcp://*:9998')
s.on('message', (n) => {
  console.log('Serv1, '+n)
  switch (n) {
      case 'uno': s.send('one'); break
      case 'dos': s.send('two'); break
      default: s.send('mmmmm.. no se')
  }
})
```

3. Pipeline pattern: push/pull

- Interconnected processing stages
  - Emitter does not expect any answer
  - Concurrent sends
- Connections
  - `1:1`, `1:n` (RR), `n:1` (fair queuing)

  - `1:n:1` (map-reduce)

- `push` y `pull` do not alter message formats



3.1. Pipeline (1:1) example

emisor.js

```
const zmq = require('zeromq')
let s = zmq.socket('push')
s.connect('tcp://127.0.0.1:9999')
s.send(['ejemplo','multisegmento'])
```

receptor.js

```
const zmq = require('zeromq')
let s = zmq.socket('pull')
s.bind('tcp://*:9999')
s.on('message', (tipo,txt) => {
    console.log('tipo '+tipo +' texto '+txt)
})
```

3.2. Pipeline (1:n:1) example

fan.js

```
const zmq = require('zeromq')
let s = zmq.socket('push')
s.connect('tcp://127.0.0.1:9996')
s.connect('tcp://127.0.0.1:9997')
s.connect('tcp://127.0.0.1:9998')
for (let i=0; i<8; i++)
    s.send(''+i)
```

sink.js

```
const zmq = require('zeromq')
let s = zmq.socket('pull')
s.bind('tcp://*:9999')
s.on('message', (w,n) => {
    console.log('worker '+w +' resp '+n)
})
```

worker1.js

```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9996')
sin.on('message', n => {sout.send(['1',n])})
```

worker2.js

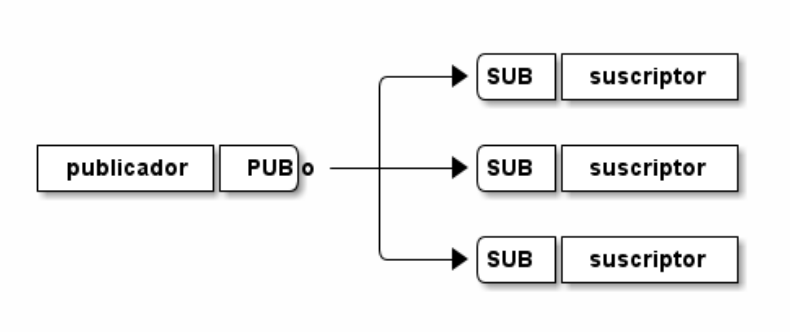
```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9997')
sin.on('message', n => {sout.send(['2',n])})
```

worker3.js

```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9998')
sin.on('message', n => {sout.send(['3',n])})
```

4. Broadcast pattern: pub/sub

- A publisher (`pub`) broadcasts messages to n subscribers (`sub`)
  - A subscriber must subscribe to the messages it is interest in `socket.subscribe(tipoMsg)`
    - Can subscribe to several types of messages
  - `s.subscribe('xx')` → socket `s` only receives messages starting with `xx`
    - Prefix `''` (empty string) selects all messages



- The subscriber starts receiving messages starting with `tipoMsg` from the moment it subscribes (messages previously sent by the publisher are not received)
- `pub` and `sub` preserve message formats

4.1. pub/sub pattern example

publicador.js

```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let msg = ['uno', 'dos', 'tres']
pub.bind('tcp://*:9999')
function emite() {
    let m=msg[0]
    pub.send(m)
    msg.shift(); msg.push(m) //rotatorio
}
setInterval(emite,1000) // every second
```

suscriptor1.js

```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('uno')
sub.on('message', (m) =>
    {console.log('1',m+'')})
```

suscriptor2.js

```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('dos')
sub.on('message', (m) =>
    {console.log('2',m+'')})
```

suscriptor3.js

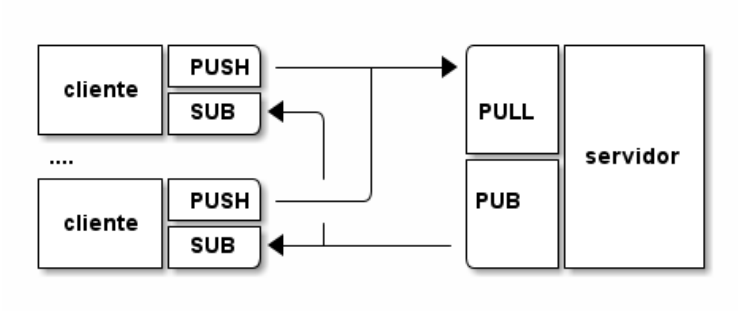
```
const zmq = require('zeromq')
let sub = zmq.socket('sub')
sub.connect('tcp://127.0.0.1:9999')
sub.subscribe('tres')
sub.on('message', (m) =>
    {console.log('3',m+'')})
```

5. Complete app example: Chat

- 1 server, n clients
  - Clients register with the server (register/unregister operations)
  - Each client can send messages to the server
  - The server broadcasts each message to all registered clients
- Steps to design the distributed applications
  1. Decide what socket combinations are needed, and in which agents to place them.
  2. Define message formats
  3. Define how messages are handled at each agent

5.1. Step 1: interaction patterns

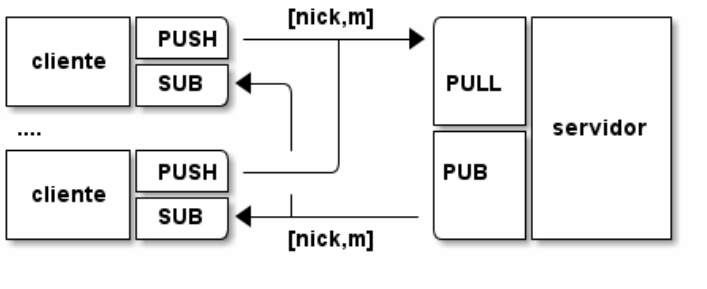
- A chat combines pipeline and broadcast
  - pipeline (client `push`, server `pull`)
    - Each client sends msg to the server when its user writes a new sentence
  - broadcast (client `sub`, server `pub`)
    - The server broadcasts each sentence to each client



NOTE.- this is not the only possible way to design the application. An alternative would make the server keep a list of the clients, and send each sentence to each one of them

5.2. Step 2: message formats

- Client to server.- Sender's nickname + text
  - Nickname identifies the text's author
- Message broadcasted by the server
  - Nickname + text
  - The server can also originate text (nick `server`)
    - E.g.- to notify clients joining/leaving, etc.
- The client shows the received message (console interface, graphic interface, nick-based color, …)



5.3. Step 3: message handling events

- Client:
  - Initially, client runs `push.send([nick,'HI'])`
  - The `sub` socket receives all messages published by the server: `sub.subscribe('')`
  - To receive messages sent to `sub`:
    * `sub.on('message', (nick,m)=> {...})`
  - It sends all text written by the user with the keyboard:
    * `process.stdin.on('data',(str)=>{push.send([nick,str])})`
  - Unregisters the client when done:
    * `process.stdin.on('end',()=>{push.send([nick,'BYE'])})`
- Server:
  - msg arrival: `pull.on('message',(id,m)->{..})`
    * if `m` is `'HI'` → register `id` (broadcasts notification)
    * if `m` is `'BYE'` → unregistration of `id` (broadcasts notification)
    * For any other `[id,m]`, broadcasts `[id,m]`

5.4. Code for chat

client.js

```
const zmq = require('zeromq')
const nick='Ana'
let sub = zmq.socket('sub')
let psh = zmq.socket('push')
sub.connect('tcp://127.0.0.1:9998')
psh.connect('tcp://127.0.0.1:9999')
sub.subscribe('')
sub.on('message', (nick,m) => {
    console.log('['+nick+']'+m)
})
process.stdin.resume()
process.stdin.setEncoding('utf8')
process.stdin.on('data'  ,(str)=> {
    psh.send([nick, str.slice(0,-1)])
})
process.stdin.on('end',()=> {
    psh.send([nick, 'BYE'])
    sub.close(); psh.close()
})
process.on('SIGINT',()=> {
    process.stdin.end()
})
psh.send([nick,'HI'])
```

server.js

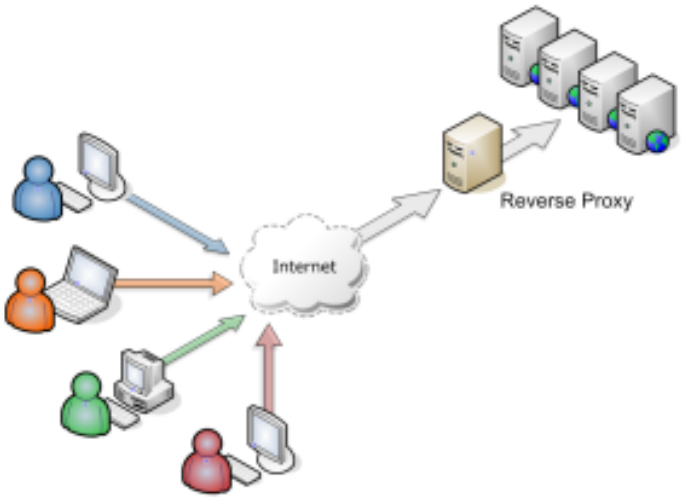
```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let pull= zmq.socket('pull')

pub.bind ('tcp://*:9998')
pull.bind('tcp://*:9999')

pull.on('message', (id,txt) => {
    switch (txt.toString()) {
    case 'HI':
        pub.send(['server',id+' connected'])
        break
    case 'BYE':
        pub.send(['server',id+' disconnected'])
        break
    default:
        pub.send([id,txt])
    }
})
```

The chat presented above is extremely simple, but it can be extended along different lines:

- Support groups of users (each with its own conversation). Each client can subscribe to several conversations.
- Support private messaging (not broadcasted, sent directly to a concrete user)
- Support modification/selection of nicknames by clients, checking they are unique.
- In the above code, no effort is made to check for possible errors like repeated registration of users, etc…
- The server is currently stateless: Messages are only sent to a client if it is connected, otherwise they are lost to that client.

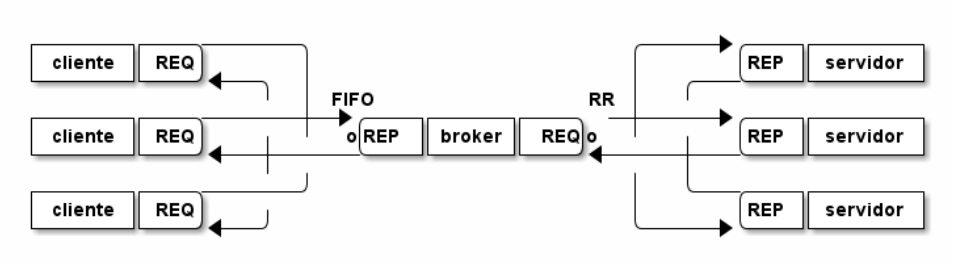
6. Broker pattern (reverse proxy)

- A broker agent (a proxy) takes care of coordinating communications among other agents
  - Servers tell broker about their properties
  - Clients request services of the broker
  - The broker redirects client requests to the proper server being able to perform several utility functions like load balancing, failure detection (of servers), etc…



6.1. Broker `req/rep`

- Using pattern `req/rep` for request/reply
- Client requests from broker, broker requests from worker server



- **Incorrect**. The broker cannot concurrently serve `n` clients
  - With `req/rep` until the broker replies the first client's request it cannot process the second client's request
  - We have a synchronous behavior, where we need it to be asynchronous

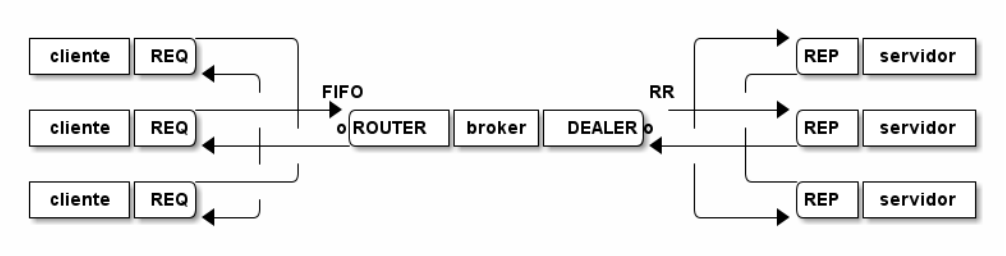
6.2. `router/dealer` broker

- In this scenario we use our last two kinds of 0MQ sockets
- Socket **dealer**
  - Asynchronous, allows sending (RR) and receiving (FIFO)
  - Does not alter messages
- Socket **router**
  - Asynchronous, allows sending and receiving (FIFO)
    * Keeps a separate queue per connection.- when receiving, it prepends the identity of the sender to the message before delivering it to the application.

· `s.identity='xx'` before connecting to a `router` → makes connection id be 'xx' at that `router`

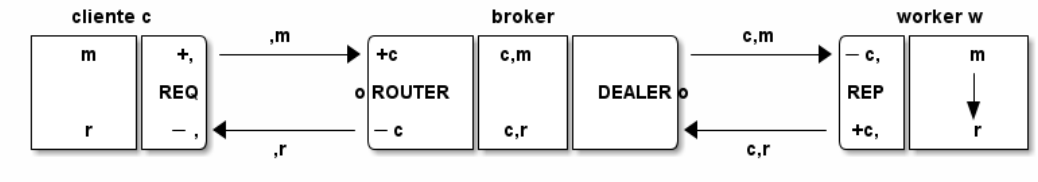
* When sending, the `router` consumes the first segment of the message, using it to select the connection through which to send the rest of the message

- Using a configuration `router/dealer` at the broker



- **Correct**
  - `router` and `dealer` work asynchronously (while handling a request they can accept new requests) → asynchronous broker
  - Requests served in arriving order (FIFO)
  - We can assign tasks to workers following a round robin (RR) schedule

- Message format
  - Design decision.- client information is part of the message
    * Avoids storing client's data at the broker
  - In figure `m` is the message, `r` the reply, `c` the client's id, `w` the worker's id, and `','` a delimiter segment
    * `req` adds a delimiter on send, removes it on receive
    * `router` adds client id when delivering a received message to the app, consumes it when the app sends a message
    * `rep` stores the envelope on receive, re-inserts the saved envelope on send of a reply



client.js

```
const zmq = require('zeromq')
let req = zmq.socket('req');
req.connect('tcp://localhost:9998')
req.on('message', (msg)=> {
    console.log('resp: '+msg)
    process.exit(0);
})
req.send('Hola')
```

worker.js

```
const zmq = require('zeromq')
let rep = zmq.socket('rep');
rep.connect('tcp://localhost:9999')
rep.on('message', (msg)=> {
    setTimeout(()=> {
        rep.send('resp')
    }, 1000)
})
```

broker.js

```
const zmq = require('zeromq')
let sc = zmq.socket('router') // frontend
let sw = zmq.socket('dealer') // backed
sc.bind('tcp://*:9998')
sw.bind('tcp://*:9999')
sc.on('message',(...m)=> {sw.send(m)})
sw.on('message',(...m)=> {sc.send(m)})
```

The broker app actions get reduced to:

- Resend through the backend socket (towards a worker, according to RR policy) each client request it receives through the frontend socket
- Resend through the frontend socket the answer it receives from the backend socket. Given that the client information travels as the envelope of that answer message, the frontend router socket will correctly select the client that must receive that answer.

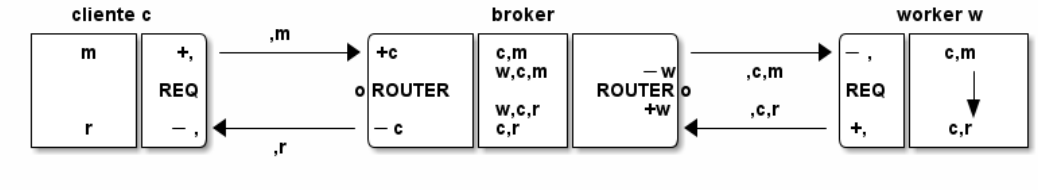
6.3. `router/router` broker

Our previous design (`router/dealer`) used RR to hand out the request to a worker. This is only good if processing costs for each job and worker are similar. Generally speaking, it is more convenient to be able to decide the precise worker that must handle a particular request → eg. to attain proper load balancing.

To get a more general approach for routing requests to workers we propose a `broker` with a `router/router` structure

- Requires the broker to explicitly maintain the list of available workers
  - Must support adding new workers (register) and remove them when no longer available (unregister)
    - Need to introduce new `worker->broker` messages: register/unregister
  - We change the worker socket to a `req`: instead of replying to requests, it asks for work to be done.
- Message format
  - Design decision.- client info piggybacked to the message sent to a worker
    - Avoids storing client state in the broker
  - In figure `m` is the message, `r` the reply, `c` client id, `w` worker id, and `','` a delimiter segment

* `req` adds delimiter on send, removes it on receive
* `router` adds client id on reception delivery, consumes it on app send
* `rep` saves the envelope on receive, piggybacks it on reply send



client.js

```
const zmq = require('zeromq')
let req = zmq.socket('req');
req.connect('tcp://localhost:9998')
req.on('message', (msg)=> {
    console.log('resp: '+msg)
    process.exit(0);
})
req.send('Hola')
```

worker.js

```
const zmq = require('zeromq')
let req = zmq.socket('req')
req.identity='Worker1'+process.pid
req.connect('tcp://localhost:9999')
req.on('message', (c,sep,msg)=> {
    setTimeout(()=> {
        req.send([c,'','resp'])
    }, 1000)
})
req.send(['','',''])
```

broker.js

```
const zmq = require('zeromq')
let cli=[], req=[], workers=[]
let sc = zmq.socket('router') // frontend
let sw = zmq.socket('router') // backend
sc.bind('tcp://*:9998')
sw.bind('tcp://*:9999')
sc.on('message',(c,sep,m)=> {
    if (workers.length==0) {
        cli.push(c); req.push(m)
    } else {
        sw.send([workers.shift(),'',c,'',m])
    }
})
sw.on('message',(w,sep,c,sep2,r)=> {
    if (c=='') {workers.push(w); return}
    if (cli.length>0) {
        sw.send([w,'',
            cli.shift(),'',req.shift()])
    } else {
        workers.push(w)
    }
    sc.send([c,'',r])
})
```

All messages delivered to the broker application through the frontend socket are structured like this `[c,'',m]`, where `c` is the id of the requesting client and `m` the actual request message.

- If no worker is available when a request is received (`workers.length==0`), the broker appends `c`to vector `cli` and `m` to vector `req` (pending request representation)
- If, on the contrary, workers are available when a request arrives, one of them is chosen (for simplicity in the code, the first one) and the `[c,'',m]` message is sent to it. To properly select the broker, given that the backend is a `router`socket, the actual message sent is `[w,'',c,'',m]`, where `w`is the selected worker's id.

All messages delivered to the broker through the backend are expected to have the structure `[w,'',c,'',r]`, where `w` is the id of the worker sending the reply message, `c` the client that should receive it, and `r` the actual reply.

- The first message sent by worker `w`is `['','','']`, delivered to the broker as `[w,'','','','']` (as `w`'s `req` socket prepends a delimiter while the broker's backend `router`socket adds the

worker's id (`w`). Instead of a client id we find `''`, which the broker interprets as directed to itself, allowing the broker to distinguish it as the first message from that worker, not as the answer to a client's request.

- In a proper answer to a client's request
  - Broker looks for pending requests
    * If pending requests (`cli.length>0`) extract the first one, pass it to that worker
    * If no pending requests, add the worker to the list of available workers.
  - Send the worker's answer message through the frontend socket, which will route it to the correct client.

Analyze the provided code till you understand how it works.

7. Possible improvements on the `router/router` broker

Several modifications of the basic `router/router` broker are possible to improve scalability and/or availability. Next we offer some of those modifications, pointing out problems to solve and possible approaches to address them, although we may not actually provide code. A detailed analysis of the provided approaches and techniques will help think of other broker improvements.

7.1. Fault tolerant broker template

To tolerate worker failures we need to detect a worker's failure, and reconfigure the system to continue working without the failed worker.

We have several alternatives to detect a worker's failure:

1. Use additional sockets (`router` in the broker, `req` in each worker), so that the broker sends periodic 'heartbeats' to the workers, and they reply to indicate they are still alive: if the answer takes too long to arrive to the broker, the worker is assumed to have failed.

2. As before, but heartbeats are sent only to those workers who have not yet returned their answers to the last requests they were sent to handle.

3. Use only the existing messages: A worker is considered failed if it does not return a reply within a given time interval → when a request is relayed to a worker, the broker runs `setTimeout(reconfigura, answerInterval)`: if the answer is received before the timeout is raised, the timeout is cleared, else, the timeout is raised, the worker is assumed to have failed, and the function `reconfigura` is run.

To simplify the code we assume alternative (3)

If we want failure transparency → when worker `w` is assumed failed, the broker resends the request it was supposed to process to another worker.

- The broker must keep track of all requests pending to be answered, together with the id of the worker that is handling it.
- If worker `w` is declared failed, the broker has the information of the request `w` was handling, and can relay it to another worker

ftbroker.js (broker que tolera fallos de workers)

```
const zmq = require('zeromq')
const ansInterval = 2000     // answer timeout. If exceeded, worker failed
```

```
let who=[], req=[]           // pending request (client,message)
let workers=[], failed={}    // available & failed workers
let tout={}                  // timeouts for attended requests

let sc = zmq.socket('router') // frontend
let sw = zmq.socket('router') // backend
sc.bind("tcp://*:9998", (err)=>{
   console.log(err?"sc binding error":"accepting client requests")
})
sw.bind("tcp://*:9999", (err)=>{
   console.log(err?"sw binding error":"accepting worker requests")
})

function dispatch(c,m) {
    if (workers.length)             // if available workers,
      sendToW(workers.shift(),c,m)  //    send request to first worker
    else {                          // no available workers
      who.push(c); req.push(m)      //    set as pending
  }
}
function resend(w,c,m) {
    return function() {  // ansInterval finished and not response
        failed[w]=true   // Worker w has failed
        dispatch(c,m)
    }
}
function sendToW(w,c,m) {
    sw.send([w,'',c,'',m])
    tout[w]=setTimeout(resend(w,c,m), ansInterval) }

sc.on('message', (c,sep,m) => dispatch(c,m))

sw.on('message', (w,sep,c,sep2,r) => {
     if (failed[w]) return             // ignore msg from failed worker
     if (tout[w]) {                    // ans received in-time
         clearTimeout(tout[w])         //    cancel timeout
         delete tout[w]
     }
     if (c) sc.send([c,'',r])          // If it was a response, send resp to client
     if (who.length)                        // If there are pending request,
         sendToW(w,who.shift(),req.shift()) //    process first pending req
     else
         workers.push(w)                    // add as available worker
})
```

7.2. Load balancing

The `router/router`broker can select the worker that will handle each request. Our simple code uses a round robin policy to select that worker, but we could also select the least loaded worker

instead, to better balance the load.

In a production system, each worker is likely to run on a different computer. A worker can get the load of the computer it is running in, and inform the broker about it. Then, the broker should store this load information, and use it to select the worker that will handle a request.

Let us define the approach to update the load info for each worker. We note that:

- A worker does not need to inform while it is processing a request.
- A worker waiting for a request will not modify its information while waiting to be given a request to handle.

With these assumptions in mind, we conclude that a worker must provide load information only when it is either registering or replying to a request, thus:

- Each worker can piggyback load info within each 'conventional' message it sends to the broker → No specific messages are needed to inform about the load
- When the broker receives a client request, it relays it to the waiting worker with the least load

We have several options to represent waiting workers and their loads:

1. Keep a non-ordered array (eg. append). To find the least loaded worker, we need to traverse this array (linear cost for each request)
2. Keep an ordered array (in increasing load): makes insertion more expensive (linear cost per reply), constant time to find the worker (the first one of the vector)
3. Use a more sophisticated data structure (eg. min-heap), with logarithmic costs on both insertion and deletion

7.3. Job types

Until now we have assumed the following:

- A unique type of client request (Job)
- Homogeneous workers

All workers are equivalent → any worker can handle any request

However, in a more realistic setting:

- We may actually have different types of requests (jobs)
- Workers may have specific capabilities

So, depending on the type of Job, we may need to direct it to specific kinds of workers

To implement his worker specialization idea, we assume that:

- Each client request carries with it the type of job. When a client is started, we configure it with the job type it requests (eg we assume types A,B,C)
- When a worker registers, it informs about the types of jobs it can handle. When starting a worker, we configure with the list of job types it can handle (eg A,B o C)

The broker can thus classify each worker according with the job types it can handle, relaying each request to one of the workers that can handle its job type. UA possibility is to replace the worker list by a set of lists, one per job type, where workers wait. Then, on request arrival, the list corresponding to the job type of the request is searched.