Lab 2: Introduction to 0MQ

Networked Information Systems Technologies. Academic Year 2025-2026

Goals

- Consolidate the theoretical concepts introduced in Unit 3.
- Experiment with different design patterns (basic communication patterns) and socket types from the ZeroMQ library on Node.js

Working method

- For simplicity, we launch the different components of each application on the same machine (IP = localhost)
  - But the components can also be distributed over different machines.
- It is not necessary to modify the provided code.
  - Programs require command-line arguments
  - These arguments allow us to propose different scenarios.
  - You can modify the "port" numbers in the examples (ask your teacher)
- In the different tasks, questions are raised that the student must answer.
  - They allow you to verify that the basic concepts have been understood
  - You must find out the answer and understand the justification for that answer.
- The file "fuentes.zip" contains the code to perform the different tests. That code uses identifiers, function names and comments written in Spanish. Those elements have not been translated.

Sessions

1.- Test of the basic patterns

2.- Chat application and rotating publisher

SESSION 1.- Testing of basic patterns

- All components used in these tests use the “tsr.js” library, which is included below for reference.
- We'll work with three patterns: REP-REQ, PULL-PUSH, and PUB-SUB, explained in the following sections. The different components of a pattern are gathered in a single directory.
- To perform tests on a pattern we must open several terminals (between 2 and 4 depending on the test to be performed), place ourselves in each terminal on the directory corresponding to the pattern, and execute the command indicated for each terminal.

tsr.js library

```
const zmq  = require('zeromq')
const path = require('node:path')
module.exports = {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion, conecta}


function error(msg) {console.error(`\x1b[31m ${msg} \x1b[0m `); process.exit(1)}


function lineaOrdenes(params) { //Comprobacion de parametros en linea de órdenes. Crea variables asociadas
   var args = params.split(" "), prog = path.win32.basename(process.argv[1])
   if( process.argv.length != (args.length+2) )
      error(`Parámetros incorrectos. Uso: node ${prog} ${names}`)
   args.forEach((param,i) => {global[param] = process.argv[2+i]})
   console.log(`Arranca el programa ${prog} con los siguientes parámetros en línea de órdenes:`)
   for (let a in args) {console.log(`\t${args[a]}\t|${global[args[a]]}|`)}
}
function traza(f,names,value) { //muestra los argumentos al invocar la función f
   console.log(`funcion ${f}`)
   var args = names.split(" ")
   for (let a in args) console.log(`\t${args[a]}\t|${value[a]}|`)
}
function adios(sockets, despedida) {      //cierra la conexión y el proceso
   return ()=>{
      console.log(`\x1b[33m ${despedida} \x1b[0m`)
      for (let s in sockets) sockets[s].close()
      process.exit(0)
   }
}
function creaPuntoConexion(socket,port) {
   console.log(`Creando punto conexión en port ${port} ...`)
   socket.bind(`tcp://*:${port}`, (err)=>{
      if (err) error(err)
      else console.log(`Escuchando en el port ${port}`)
   })
}
function conecta(socket,ip,port) {
   console.log(`Conectando con ip ${ip} port ${port}`)
   socket.connect(`tcp://${ip}:${port}`)
}
```

1.1- Testing the basic patterns: REQ-REP

- We open three independent terminals and in all of them we go to the req-rep directory
- In this directory you will find the files *cliente1.js, cliente2.js and servidor.js*
- To practice this pattern we will perform 3 different tests, plus all the tests that the student wishes to practice:
  - A client and a server
  - One client and two servers
  - Two clients and one server

cliente1.js

```
const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
lineaOrdenes("ipServidor portServidor nombre")


let req = zmq.socket('req')
conecta(req, ipServidor, portServidor)


function procesaRespuesta(idServidor, saludo, nombre, iteracion) {
 traza('procesaRespuesta','idServidor saludo nombre iteracion',[idServidor, saludo, nombre, iteracion]) //traza
  if (iteracion==4) adios([req], "He terminado")()
}
req.on('message', procesaRespuesta)
req.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([req],"abortado con CTRL-C"))


for (let i=1; i<=4; i++) {
  console.log(`enviando mensaje: [${nombre},${i}]`)
  req.send([nombre,i])
}
```

cliente2.js

```
const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
lineaOrdenes("ipServidor1 portServidor1 ipServidor2 portServidor2 nombre")


let req = zmq.socket('req')
conecta(req, ipServidor1, portServidor1)
conecta(req, ipServidor2, portServidor2)


function procesaRespuesta(idServidor, saludo, nombre, iteracion) {
 traza('procesaRespuesta', 'idServidor saludo nombre iteracion', [idServidor,saludo,nombre,iteracion])
  if (iteracion==4) adios([req], "He terminado")()
}
req.on('message', procesaRespuesta)
req.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([req],"abortado con CTRL-C"))


for (let i=1; i<=4; i++) {
  console.log(`enviando mensaje: [${nombre},${i}]`)
  req.send([nombre,i])
}
```

servidor.js

```
const {zmq, error, lineaOrdenes, adios, traza, creaPuntoConexion} = require('../tsr')
lineaOrdenes("identidad port segundos saludo")


var rep = zmq.socket('rep')
creaPuntoConexion(rep, port)


function procesaPeticion(nombre, iteracion) {
  traza('procesaPeticion','nombre iteracion',[nombre,iteracion])
  setTimeout(()=>{rep.send([identidad, saludo, nombre, iteracion])}, parseInt(segundos)*1000) //calcula y envía respuesta
}


rep.on('message', procesaPeticion)
rep.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([rep],"abortado con CTRL-C"))
```

REQ-REP: A client and a server

- terminal 1) `node servidor.js A 9990 2 Hello`
- terminal 2) `node cliente1.js localhost 9990 Pepe`
- Questions
  - What happens if we pass a wrong number of arguments? What if they're out of order? An error is raised
  - Check if the order in which we start the components affects the result. State the reason.
  - Regarding multi-segment messages:
    - How does the sender construct a multi-segment message?
    - How does the receiver access the different segments of the message?
  - The client terminates after receiving the response to the fourth request. When does the server terminate?

2- No, the order doesn't affect the execution. This is due to the asynchronous behaviour of the REQ-REP pattern. The messages are stored in the Outgoing queue if the server is not turned on

3-1 The sender is constructing multi-segment messages by using an array
3-2 The server access the elements the multi-segment messages individually by addressing them with the array

4- The server only terminates when reciving a SIGINT event, which corresponds to CTRL-C

1- No, as each message is destined to an specific server, so the order is always preserved. However, this provokes that as long as one of the servers is not on, the client won't ever end, as its outgoing queue is still waiting for the other server to connect

3- No, the order wouldn't change, as the REQ will use Round-Robin to forward the messages if it is connected to more than 1 server. Therefore, the client will always send the second and forth message to the second server, no matter what the order or delays.

REQ-REP: one client and two servers

- terminal 1) `node servidor.js A 9990 2 Hello`
- terminal 2) `node servidor.js B 9991 2 Hello`
- terminal 3) `node cliente2.js localhost 9990 localhost 9991 Pepe`
- Questions
  - Check if the start order affects the result. State the reason.
  - What happens if both servers receive the same port number?
  - What happens if the two servers receive different seconds (e.g., 1 and 3 respectively)? Does this affect the order in which the client is responded to?
  - Almost all the time is consumed by the servers. Can we reduce the client execution time by half by using two servers?
  - If we want to use 3 servers, do we need to modify the client code?
  - With an even number of requests, can we guarantee that each server serves the same number of requests?

An error: address already used is raised

Yes, in order to include the third server's IP and port

5- Yes, as we use RoundRobin. Each msg will go for one of the servers

4- As we are using a REQ-REP socket, the REQ socket will have to receive a response to the first request before being able to transmit the next request. Therefore, the time of the client cannot be reduced, as it will always show the same behaviour. Using a intermmediate proxy wouldn't improve the execution time.

REQ-REP: Two clients and one server

- terminal 1) `node servidor.js 9990 Sports Football Culture`
- terminal 2) `node cliente1.js localhost 9990 Pepe`
- terminal 3) `node cliente1.js localhost 9990 Ana`
- Questions
  - Check if the order in which we start the components affects the result. State the reason.
  - Can we ensure that each client receives only the responses to their own requests? State the reason.
  - If a different number of clients is proposed (e.g. 3), would it be necessary to modify the client or server code?
  - If one of the clients terminates prematurely (Ctrl-C), does the other continue to receive responses? State the reason.

1- If we start the clients before the server, the order of messages will always be the same. However, if we start first the server, depending on the delay between the initiation of the clients the order of reception in the server will be different. Nevertheless, once both clients are on, the server will alternate between client1 and client2. Both clients will always receive their messages in the correct order.
This is due to the RR policy when processing and answering requests.

2- Yes. Each turn, the server processes one of the messages, IN ORDER. While a message is being processed by the server, any incoming message is stored in the incoming queue. The message will stay in that queue as long as the server doesn't finish the communication with the client. Therefore, each request is attended in order, sequentially, always ensuring that the message is correctly responded

3- No, neither of the codes should be modified.

4- Yes, it does continue. The reason is that when the server receives the message, it process its and returns the message, passing then to the next message.

1.2. Testing the basic patterns: PUSH-PULL

- We open three independent terminals and in all of them we go to the push-pull directory
- In this directory you will find the files origen1.js, origen2.js, filtro.js and destino.js; respectively: origin/source (able to handle 1 or 2 filters), filter and destination.
- We will do the following tests:
  - One origin and one destination
  - One source, one filter, and one destination
  - One source, two filters and one destination

origen1.js

```
const {zmq, lineaOrdenes, error, adios, conecta} = require('../tsr')
lineaOrdenes("nombre hostSig portSig")

let salida = zmq.socket('push')
conecta(salida, hostSig, portSig)

salida.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([salida],"abortado con CTRL-C"))

for (let i=1; i<=4; i++) {
   console.log(`enviando mensaje: [${nombre},${i}]`)
   salida.send([nombre,i])
}
```

origen2.js

```
const {zmq, lineaOrdenes, error, adios, conecta} = require('../tsr')
lineaOrdenes("nombre hostSig1 portSig1 hostSig2 portSig2")

let salida = zmq.socket('push')
conecta(salida, hostSig1, portSig1)
conecta(salida, hostSig2, portSig2)

salida.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([salida],"abortado con CTRL-C"))

for (let i=1; i<=4; i++) {
   console.log(`enviando mensaje: [${nombre},${i}]`)
   salida.send([nombre,i])
}
```

filtro.js

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion, conecta} = require('../tsr')
lineaOrdenes("nombre port hostSig portSig segundos")

let entrada = zmq.socket('pull')
let salida  = zmq.socket('push')

creaPuntoConexion(entrada, port)
conecta(salida, hostSig, portSig)

function procesaEntrada(emisor, iteracion) {
   traza('procesaEntrada','emisor iteracion',[emisor,iteracion])
   setTimeout(()=>{
      console.log(`Reenviado: [${nombre}, ${emisor}, ${iteracion}]`)
      salida.send([nombre, emisor, iteracion])
   }, parseInt(segundos)*1000)
}

entrada.on('message', procesaEntrada)
entrada.on('error'  , (msg) => {error(`${msg}`)})
 salida.on('error'  , (msg) => {error(`${msg}`)})
process.on('SIGINT' , adios([entrada,salida],"abortado con CTRL-C"))
```

destino.js

```
const {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion} = require('../tsr')
lineaOrdenes("nombre port")

var entrada = zmq.socket('pull')
creaPuntoConexion(entrada, port)

function procesaMensaje(filtro, nombre, iteracion) {
   traza('procesaMensaje','filtro nombre iteracion',[filtro, nombre, iteracion])
}
entrada.on('message', procesaMensaje)
entrada.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([entrada],"abortado con CTRL-C"))
```

PUSH-PULL: One source and one destination {AB}

- terminal 1) `node origen1.js A localhost 9000`
- terminal 2) `node destino.js B 9000`
- Questions
  - Check whether the order in which we start the components affects the result. State the reason.
  - Explain why the socket defined in origen1.js does not use socket.on('message',..)

1.Thanks to persistence the order in which the components turn on does not matter

2. It does not use it because the socket is a PUSH socket, so it won't and cannot receive a message: it only sends

PUSH-PULL: one source, one filter, and one destination {ABC}

- terminal 1) `node origen1.js A localhost 9000`
- terminal 2) `node filtro.js B 9000 localhost 9001 2`
- terminal 3) `node destino.js C 9001`
- Questions
  - Check whether the order in which we start the components affects the result. State the reason. No, due to the persistence of messages, all of them are send and received in order
  - Explain why origen1.js and destino.js define a single socket each, but filtro.js defines 2 sockets Because filtro needs to obtain the message from origen1 and send the message to destino, which requires 2 sockets
  - If origen1 generates 4 messages and filtro delays 2 seconds, how long do you think it takes for the last message from source to reach destination?

Around 2 seconds. Push does not wait for replies, so it sends one message after the other, practically instantaneous. Then, filter takes 2 seconds, and afterwards the messages are sent to destino without waiting for obtaining a response

PUSH-PULL: One source, two filters and one destination: { A-(B,C)-D }

Note that we use "origen2"

- terminal 1) `node origen2.js A localhost 9000 localhost 9001`
- terminal 2) `node filtro.js B 9000 localhost 9002 2`
- terminal 3) `node filtro.js C 9001 localhost 9002 3`
- terminal 4) `node destino.js D 9002`
- Questions
  - Check whether the order in which we start the components affects the result. State the reason.
  - How is message delivery distributed to filters B and C?
  - Can B and C work in parallel (e.g. if they were running on different machines)?
  - In what order do messages arrive at their destination? How would changing the filter C seconds affect that behavior?

1. The order in which components are started DOES affect the order of the messages in the final destination.
If we connect first one filter, its messages that will arrive will be the even / odd, but never a mix. This is due to the ROund Robin strategy followed in this type of patterns.

2. The frst and third messages are enqueued and sent through the channel connected to B, and the other 2 are sent to C (Round Robin)

3.Yes, as both filters are independent from one another

4. The messages arrive in the order they where sent. If the messages from B arrive first, they will be displayed first. If we varied the delays in filters, then the order of messages would also be affected

1.3.- Testing the basic patterns: PUB-SUB

- We open 3 independent terminals, and in all of them we open the pub-sub directory
- In this directory you will find the files publicador.js and suscriptor.js (i.e., publisher and subscriber, respectively)
- We will test with one publisher and several subscribers.

publicador.js

```
const {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion} = require('../tsr')
lineaOrdenes("port tema1 tema2 tema3")
let temas = [tema1,tema2,tema3]
let pub = zmq.socket('pub')
creaPuntoConexion(pub, port)


function envia(tema, numMensaje, ronda) {
   traza('envia','tema numMensaje ronda',[tema, numMensaje, ronda])
   pub.send([tema, numMensaje, ronda])
}
function publica(i) {
   return () => {
      envia(temas[i%3], i, Math.trunc(i/3))
      if (i==10) adios([pub],"No me queda nada que publicar. Adios")()
      else setTimeout(publica(i+1),1000)
   }
}
setTimeout(publica(0), 1000)
pub.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([pub],"abortado con CTRL-C"))
```

suscriptor.js

```
const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
lineaOrdenes("identidad ipPublicador portPublicador tema")


let sub = zmq.socket('sub')
sub.subscribe(tema)
conecta(sub, ipPublicador, portPublicador)


function recibeMensaje(tema, numero, ronda) {
   traza('recibeMensaje','tema numero ronda',[tema, numero, ronda])
}


   sub.on('message', recibeMensaje)
   sub.on('error'  , (msg) => {error(`${msg}`)})
process.on('SIGINT' , adios([sub],"abortado con CTRL-C"))
```

PUB-SUB: one publisher, several subscribers

- terminal 1) `node publicador.js 9990 Economy Sports Culture`
- terminal 2) `node suscriptor.js A localhost 9990 Economy`
- terminal 3) `node suscriptor.js B localhost 9990 Sports`
- terminal 4) `node suscriptor.js C localhost 9990 Economy`
- Questions
  - Check whether the order in which we start the components affects the result
  - What about Culture messages?
  - Can more than one subscriber receive the same message?
  - How can I change the number of messages the publisher generates?
  - Subscribers don't end. Consider a change to have them terminated after a certain period without receiving messages.
  - It's possible that the publisher generates some messages before it has yet to process subscribers' connections. What happens to those messages?

1. The order will affect the result: as the PUB/SUB is not persistent, every message that a subscriber looses will not be recovered. Therefore, if a subscriber connects after the publisher has sent everything, then no message will be received.

2. Same as with Economy. All the messages, if received, will be displayed in the same order. HOWEVER, those not received will not be recovered.

3.Yes. Subscribers receive the same messages published over one topic.

4.By changing the function publica in publicador, by increasing the max limit stablished (10), we can make that the publisher sends more messages

5.We could have that after connecting for the first time, the suscriber uses setTimeout() to launch after X seconds an exit(1). Therefore, whenever a message is received, the setTimeout would be deleted, and after processing the message a new setTimeout would be initiated.

6. They are lost

SESSION 2.- Chat application and rotating publisher

During this session, we'll practice using different sockets using a small chat application and make a small modification to the publisher/subscriber code to convert it into a rotating publisher.

2.1.- Chat application

- We open three independent terminals and in all of them, go to the pub-sub directory.
- In this directory you will find the files clienteChat.js and servidorChat.js

clienteChat.js

```
const {zmq, params, error, adios, conecta} = require('./tsr')
params(nick hostServidor portDifusion portPipeline)


let entrada = zmq.socket('sub')
let salida  = zmq.socket('push')
connecta(salida, `tcp://${hostServidor}:${portPipeline}`)
connecta(entrada,`tcp://${hostServidor}:${portDifusion}`)
entrada.subscribe('')
entrada.on('message', (nick,m) => {console.log('['+nick+']'+m)})


process.stdin.resume()
process.stdin.setEncoding('utf8')
process.stdin.on('data' ,(str)=> {salida.send([nick, str.slice(0,-1)])})
process.stdin.on('end',()=> {salida.send([nick, 'BYE']); adios([entrada.salida],"Adios")()})


salida.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([entrada,salida],"abortado con CTRL-C"))
salida.send([nick,'HI'])
```

servidorChat.js

```
const {zmq, params, error, adios, creaPuntoConexion} = require('./tsr')
params(portDifusion portEntrada)

let entrada = zmq.socket('pull')
let salida  = zmq.socket('pub')
creaPuntoConexion(salida, portDifusion)
creaPuntoConexion(entrada,portEntrada)

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

salida.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([entrada,salida],"abortado con CTRL-C"))
```

Testing the application

- terminal 1) `node servidorChat.js 9000 9001`
- terminal 2) `node clienteChat.js Pepe localhost 9000 9001`
- terminal 3) `node clienteChat.js Ana localhost 9000 9001`
- Questions
  - How does the order in which we start the components affect?
  - Explain why both endpoints are created on the server
  - The server doesn't maintain a list of connected clients. Why?
  - Our code interprets the text of certain messages (e.g., "HI" and "BYE"). Consider a strategy to gather additional information that allows you to separate the message type from the text.
  - Assuming we have solved the previous point, think about how to modify a client so that it only handles messages from certain specific topics.

1.The order does not matter, if we are considering that messages are not sent. Any message sent without the other client being active will be lost

2.Because the clients are not sending the messages to one another directly, but indirectly through the server. Therefore, whenever a message is sent, the

3.Because it doesn't need it. The PUB doesn't store a list of connected SUB, but rather it broadcasts the information to all subscribers, who depending on their subscription will accept and process the message or not. It is the SUB socket who perfroms the filtering

4.

5.

2.2 – Implementing a rotating publisher

- Using the pub-sub pattern and using the code provided in the previous session as a base, develop a publisher.js program that:
  - It is invoked as

    ```
    node publisher port numMessages topic1 topic2 topic3 ...
    ```
  - port = the port to which subscribers should connect (host is localhost)
  - numMessages = number of messages to be issued, after which the publisher terminates
  - topic1 topic2 topic3 … = variable number of topics (a priori we do not know how many)
  - It must generate one message per second, each with the following structure
    - topic (in circular order), message number, round (first iteration on the topics, second, etc.)

- Example:

```
$ node publisher 9000 5 Politics Football Culture
Politics 1 1
Football 2 1
Culture 3 1
Politics 4 2
Football 5 2
```