TSR - Documentació de referència sobre **0MQ**

Curs 2019/20

Contents

1. 0MQ

- L'assignatura CSD va introduir MOM (Message Oriented Middleware)
  - Comunicació indirecta → asincronia, persistència, acoblament feble
  - Es va estudiar JMS com a exemple
- 0MQ és un altre exemple de MOM, però molt diferent a JMS:

  - Sense broker per a encaminar missatges, amb API similar a sockets
    * No cal arrancar cap servidor específic
    * Proporciona sockets per a enviament/recepció
    * Utilitza URLs per a nomenar els endpoints
    * Persistència feble (cues en RAM)
  - Model E/S asincrònica (dirigit per esdeveniments)
  - Àmplia disponibilitat (per a molts SO i llenguatges)
    * Biblioteca gratuïta (codi obert), enllaça amb l'aplicació

1.1. Característiques bàsiques

- Eficient (compromís fiabilitat/eficiència)
  - Persistència feble (cues en RAM)
  - Els sockets tenen cues de missatges associades

cua missatges	descripció	esdeveniments
d'entrada (recepció)	manté missatges que arriben	genera esdeveniment "message"
d'eixida (enviament)	manté missatges a enviar	

- Defineix diferents patrons d'intercanvi de missatges
  - facilita el desenvolupament
- Útil a diversos nivells → mateix codi per a comunicar fils en un procés, processos en una màquina, màquines en xarxa IP
  - Només canvia la configuració del transport en la URL
  - Ens centrem en comunicació entre màquines sobre TCP

1.2. Missatges

- Gestió de buffers transparent
  - Gestiona el flux de missatges entre les cues dels processos i nivell de transport
  - Contingut del missatge transparent
- Els missatges s'entreguen de manera atòmica (tot o res)
  - Poden ser multi-segment (segment = counted string)
    * 1 segment: `send('hola')` → `4 h o l a`
    * 3 seg: `send(['hola','','Ana'])` → `4 h o l a 0 3 A n a`

envie missatge	recepció
`send(['un','dos'])`	`sock.on("message", (a,b)=>{..})` `a` val `'un'`, `b` val `'dos'`
`send(msg)`	`sock.on("message", (...m)=>{..})` segments de `msg` en el vector `m`

1.3. Connexions

- Gestió de connexió/reconnexió entre agents automàtica
  - Un agent executa bind: la resta executa connect

    - En qualsevol ordre
    - Tots els agents coincideixen en algun endpoint
  - Si s'executa bind sobre un port que ja està en ús, apareix un error d'execució

- Connexió/Reconnexió en el transport TCP

  - bind.- La dir IP pertany a una de les interfícies del socket
    - `s.bind('tcp://*:9999')`
  - connect.- ha de conèixer la dir IP del socket que realitze `bind`
    - `s.connect('tcp://127.0.0.1:9999')`

- Quan un agent acaba executa close de forma implícita

- No solament comunicació 1:1

  - n:1 → ex. n clients (cadascun connect), 1 servidor (bind)
  - 1:n → ex. 1 client (n connect, un a cada servidor), n servidors (cadascun bind)

1.4. 0MQ en node

- Instal · lació biblioteca: `npm install zeromq@4`
- Sintaxi

```
const zmq = require('zeromq')         // importa biblioteca
let zsock = zmq.socket('tipusSocket')// creació socket (existeixen diversos tipus)
zsock.bind("tcp://*:5555")            // bind en el port 5555
zsock.connect("tcp://10.0.0.1:5555") // o connect (host 10.0.0.1, port 5555)
zsock.send([..,..])                   // enviament
zsock.on("message", callback)         // recepció
zsock.on("close", callback)           // resposta al tancament de la connexió
```

1.5. Sockets 0MQ

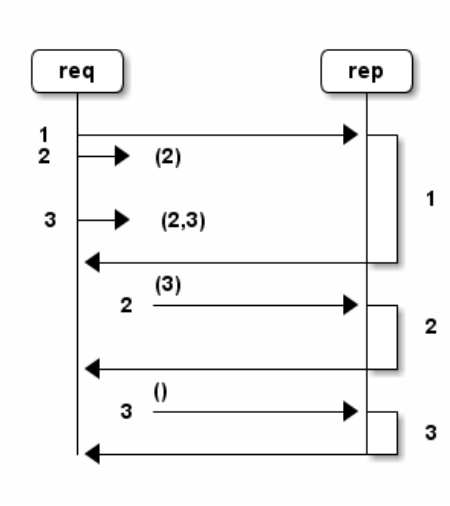
- Existeixen diferents tipus de sockets, per a implementar patrons de disseny (patrons de comunicació) freqüents en Sistemes Distribuïts

  - Cada patró té necessitats diferents → utilitza sockets diferents
  - Molts es resolen amb un parell de sockets específics
  - En altres casos cal combinar diversos tipus de sockets simples

- Quan coneguem els tipus de sockets, 3 passos per a dissenyar aplicació distribuïda:

  1. Decideix quines combinacions de sockets necessites, i en quins agents se situen
  2. Defineix el format dels missatges a intercanviar
  3. Defineix les respostes de cada agent davant els esdeveniments generats pels diferents sockets

2. Patró Client/Servidor: req/rep



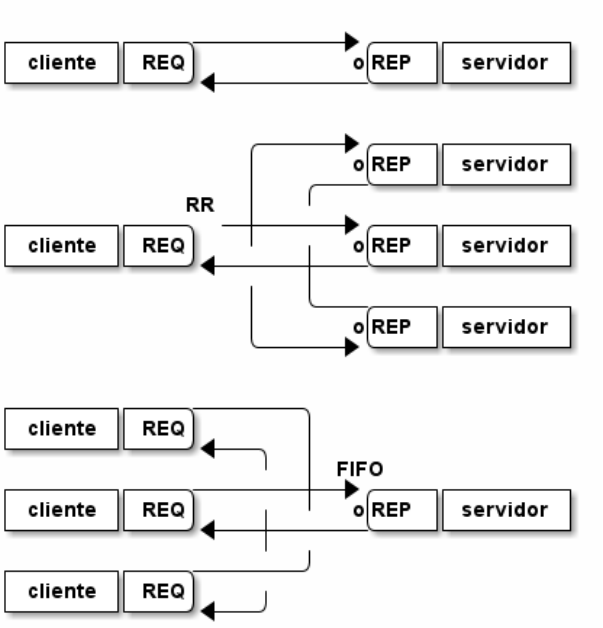
- El servidor usa socket tipus `rep`
  - Executa `bind`
  - Rep `s.on('message', callback)`
- El client usa socket tipus `req`
  - Executa connect
  - Envia amb `s.send(msg)`
- És un **patró de comunicació sincrònic**
  - Si un client envia n peticions, la segona, tercera,.. queden en cua local fins a rebre resposta de la primera
  - Parells pet/resp totalment ordenats

2.1. Connexions

En tots els casos

- Sol · licitud/Resposta
- En les figures el símbol `o` representa `bind`

Són possibles diferents estratègies de connexió (figura).



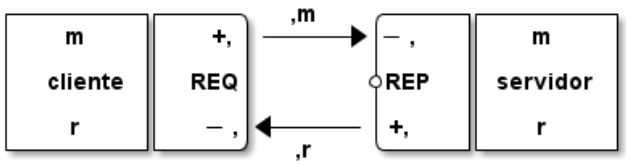
- Connexió `1:1`
- Connexió `1:n`
  - Round-Robin (RR)
  - Sense paral · lelització
- Connexió `n:1`
  - Típica servidor
  - Cola FIFO (encuat equitatiu)
    - Sense inanició clients

2.2. Format missatges

1. L'aplicació client envia un missatge `m`
2. `req` afig un primer segment buit (delimitador) al missatge
   - Delimitador → segment buit que separa embolcall i cos del missatge

  - Els segments fins al primer delimitador es denominen embolcall: permeten indicar metadades associades al missatge (ex.- a qui cal retornar-li la resposta)
  - Els segments següents al primer delimitador es consideren el cos del missatge (dades)
- En la figura representem el delimitador com ,

3. `rep` guarda l'embolcall i passa el cos a l'aplicació
4. Quan `rep` envia la resposta, afig de nou l'embolcall
5. Quan `req` rep la resposta, descarta el delimitador i passa el cos a l'aplicació



2.3. Exemple Client/Servidor 1:1

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()
})
```

Execució

```
> 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)
})
```

Execució

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

2.4. Exemple Client/Servidor 1:n

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. Exemple Client/Servidor n:1

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

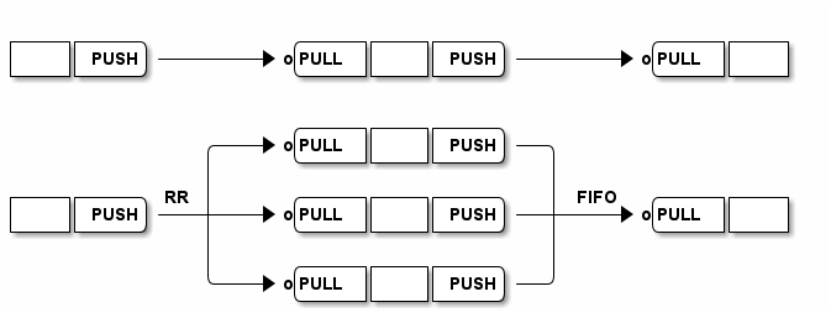
```
const zmq = require('zeromq')
let s = zmq.socket('req')
s.connect('tcp://127.0.0.1:9998')
s.send('dos')
s.on('message', (msg) => {
  console.log('Recibido: '+msg)
  s.close()
})
```

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. Patró Pipeline: push/pull

- Etapes de processament connectades entre sí
  - L'emissor no espera resposta
  - Enviaments concurrents
- Connexions
  - `1:1`, `1:n` (RR), `n:1` (encuat equitatiu)
  - `1:n:1` (map-reduce)
- `push` i `pull` no alteren el format dels missatges



3.1. Exemple Pipeline (1:1)

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. Exemple Pipeline (1:n:1)

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

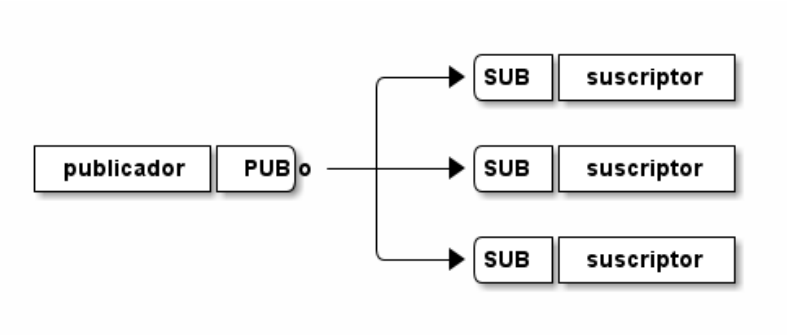
```
const zmq = require('zeromq')
let sin = zmq.socket('pull')
let sout= zmq.socket('push')
sout.connect('tcp://127.0.0.1:9999')
sin.bind('tcp://*:9997')
sin.on('message', n => {sout.send(['2',n])})
```

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. Patró Difusió: pub/sub

- Un publicador (`pub`) difon missatges a n subscriptors (`sub`)
  - El subscriptor ha de subscriure's al tipus de missatges que li interessen `socket.subscribe(tipusMsg)`
    - Pot subscriure's a diversos tipus de missatges
  - `s.subscribe('xx')` → el socket `s` únicament rep els missatges que tinguen `xx` com a prefix del primer segment
    - El prefix `''` (tira buida) concorda amb tots els missatges



- El subscriptor comença a rebre els missatges de `tipusMsg` a partir del moment en què se subscriu (si el publicador ja havia enviat uns altres prèviament, no els rep)
- `pub` i `sub` no modifiquen el format dels missatges

4.1. Exemple Patró pub/sub

publicador.js

```
const zmq = require('zeromq')
let pub = zmq.socket('pub')
let msg = ['un', 'dos', 'tres']
pub.bind('tcp://*:9999')
function emite() {
    let m=msg[0]
    pub.send(m)
    msg.shift(); msg.push(m) //rotatori
}
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('un')
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+'')})
```

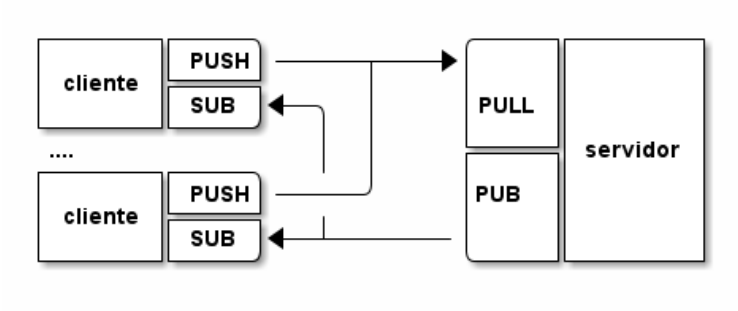
5. Exemple de disseny d'una aplicació completa: Xat

- 1 servidor, n clients
  - Els clients es registren en el servidor (operacions d'alta i baixa)
  - Cada client pot enviar missatges al servidor
  - El servidor difon cada missatge a tots els clients registrats
- Passos per a dissenyar una aplicació distribuïda:
  1. Decideix quines combinacions de sockets necessites, i en quins agents se situen

2. Defineix el format dels missatges a intercanviar
3. Defineix les respostes de cada agent davant els esdeveniments generats pels diferents sockets

5.1. Pas 1: patrons d'interacció

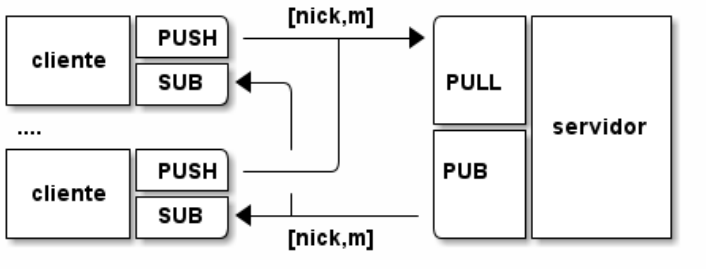
- Un xat combina pipeline i difusió
  - pipeline (client `push`, servidor `pull`)
    - Cada client envia msg al servidor quan l'usuari introdueix una nova frase
  - difusió (client `sub`, servidor `pub`)
    - El servidor difon a tots els clients cada nova frase



NOTA.- no és l'única decisió de disseny possible. Per exemple, podem resoldre la difusió sobre la base de mantenir en el servidor la llista de clients, i remetre missatges a cadascun

5.2. Pas 2: format dels missatges

- Client a servidor.- Indica remitent (nick únic) i text
  - El nick indica l'autor de la frase
- Missatge difós pel servidor a clients
  - Ha d'indicar l'autor de la frase i el text
  - L'autor de la frase pot ser el servidor (nick `server`)
    - Ex.- per a avisar de l'alta o baixa d'un client, etc.
- El client presenta el missatge (interfície text, interfície gràfica, color segons el nick, …)



5.3. Pas 3: esdeveniments

- Client:

  - Inicialment el client executa `push.send([nick,'HI'])`
  - El socket `sub` escolta tots els missatges publicats pel servidor: `sub.subscribe('')`
  - Arriba missatge del servidor al socket `sub`:
    * `sub.on('message', (nick,m)=> {...})`
  - Envia al servidor tota frase escrita per teclat:
    * `process.stdin.on('data',(str)=>{push.send([nick,str])})`
  - Dóna de baixa al client quan finalitza l'aplicació:
    * `process.stdin.on('end',()=>{push.send([nick,'BYE'])})`
- Servidor:
  - Arriba msg: `pull.on('message',(id,m)->{...})`
    * si `m` és `'HI'` → alta de `id` (difon a tots l'avís de l'alta)
    * si `m` és `'BYE'` → baixa d'`id` (difon a tots l'avís de la baixa)
    * Per a un altre `m`, difon `[id,m]`

5.4. Codi xat

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])
    }
})
```

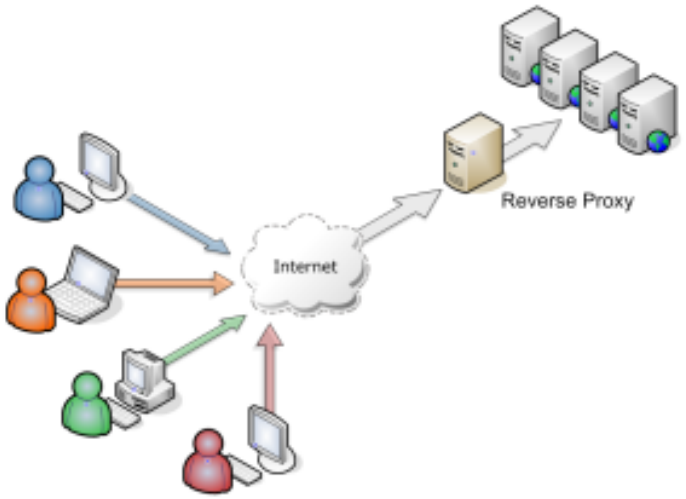
El xat desenvolupat és molt rudimentari (limitat), però pot estendre's fàcilment en diferents sentits:

- Permetre diferents grups d'usuaris (diverses converses). En aquest cas cada client se subscriu a aquells grups en els que està interessat
- Permetre missatges privats (no es difonen, sinó que es remeten de forma directa a destinació)

- Els clients haurien de poder seleccionar/modificar el seu nick, i el servidor verificar que cada nick és únic
- El codi actual no comprova possibles fallades (ex. no verifica l'entrada de l'usuari, de manera que es pot escriure un missatge que coincidisca amb els d'alta/baixa, etc.)
- El servidor no manté estat: un client que no està connectat es perd missatges, i aquests missatges no es reenvien quan connecta el client

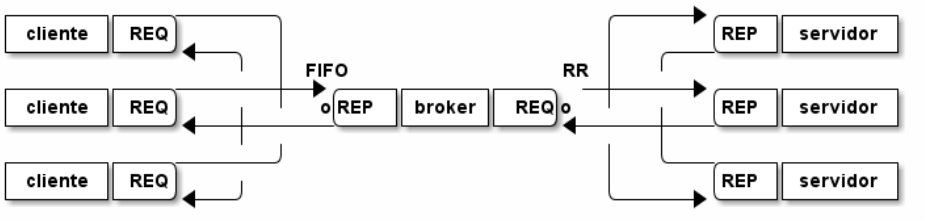
6. Patró broker (proxy invers)

- El component broker (intermediari) s'ocupa de la coordinació/comunicació entre components
  - Els servidors comuniquen al broker les seues característiques
  - Els clients sol · liciten els serveis al broker
  - El broker redirigeix cada sol · licitud al servidor adequat (ex. equilibrant la càrrega)



6.1. Broker `req/rep`

- Utilitzar el patró `req/rep` per a sol · licitud/resposta
- El client demana al broker, i el broker a un worker

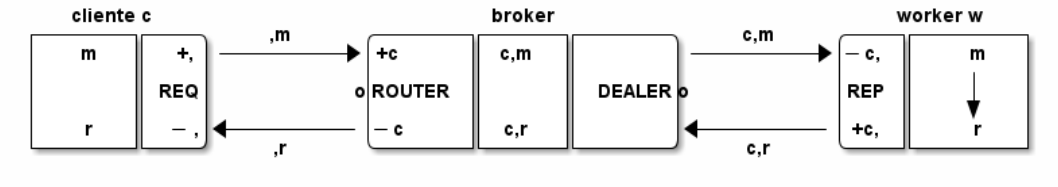


- **Incorrecte**. El broker no atén `n` clients alhora
  - Amb `req/rep` fins que el broker no retorna la resposta al primer client no pot processar la sol · licitud d'un altre client
  - Tenim comportament sincrònic, i ha de ser asincrònic

6.2. Broker `router/dealer`

- Introduïm els dos últims tipus de sockets
- Socket **dealer**
  - És asincrònic, i permet enviament (RR) i recepció (FIFO)
  - No modifica el missatge ni en enviar ni en rebre
- Socket **router**
  - És asincrònic, i permet enviament i recepció (FIFO)
    - Manté una cua per connexió.- quan rep, afig al missatge la identitat de la connexió (ID emissor)
      - `s.identity='xx'` abans d'establir la connexió → en connectar, l'identificador de la connexió és 'xx'
    - En enviar, consumeix el primer segment del missatge, i l'utilitza per a determinar la connexió a través de la qual envia el missatge (pot enrutar)
- Utilitzem la configuració `router/dealer` en el broker

- **Correcte**
  - `router` i `dealer` són asincrònics (atendre la petició d'un client en un worker no impedeix atendre peticions d'uns altres en uns altres workers) → broker asincrònic
  - Peticions servides en ordre d'arribada (FIFO)
  - Repartiment de tasques als workers segons torn rotatori (RR)
- Format missatges
  - Decisió de disseny.- la informació del client viatja al costat del missatge
    - No cal guardar estat de cada client
  - En la figura `m` és el missatge, `r` la resposta, `c` identitat del client, `w` identitat del worker, i `','` un delimitador
    - `req` afig delimitador en enviar, l'elimina en rebre
    - `router` afig identitat de la connexió en rebre, i la consumeix en enviar
    - `rep` guarda l'embolcall en rebre, el reinsereix en enviar



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)})
```

El broker es limita a:

- Reexpedir pel backend (cap a un treballador, segons torn RR) cada petició de client que li arriba pel frontend
- Reexpedir al client adequat a través del frontend cada resposta que li arribe des d'un treballador a través del backend. Per a retornar la resposta al client adequat s'utilitza la informació de client que viatja en el missatge de resposta.

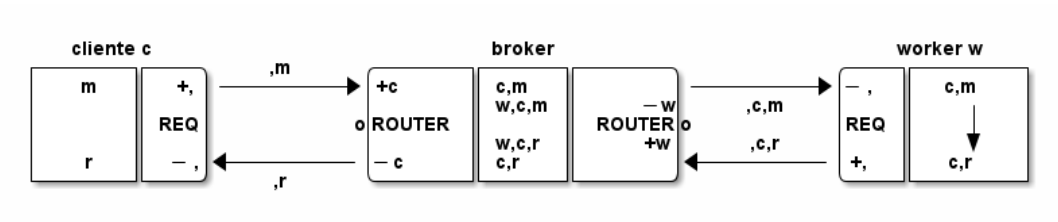
6.3. Broker `router/router`

El segon intent (`router/dealer`) utilitza RR per a repartir el treball. Aqueixa solució només és vàlida quan el cost de processar cada treball és molt similar. En general és convenient poder decidir la destinació (encaminar) → ex. per a aconseguir equilibrat de càrrega.

Per al cas general es proposa un `broker` amb `router/router`

- Cal mantenir en el broker la llista de workers disponibles
  - Afegir un nou worker (alta), eliminar-lo si es dóna de baixa
    - Nous missatges `worker->broker`: alta i baixa
  - Canviem el socket del worker a req: en lloc de respondre a peticions, sol · licita treballs
- Format missatges
  - Decisió de disseny.- la info de client viatja al costat del missatge

    - No cal guardar estat de cada client
- En la figura `m` és el missatge, `r` la resposta, `c` identitat del client, `w` identitat del worker, i `','` un delimitador
    - `req` afig delimitador en enviar, l'elimina en rebre
    - `encaminador` afig identitat de la connexió en rebre, i la consumeix en enviar
    - `rep` guarda l'embolcall en rebre, el reinsereix en enviar



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])
})
```

Tots els missatges que el broker rep des de frontend tenen l'estructura `[c,'',m]`, on `c` correspon a la identitat del client que fa la sol · licitud i `m` al missatge de petició.

- Si en rebre la petició no hi ha treballadors disponibles (`workers.length==0`), anotem al final del vector `cli` el valor de `c` i al final del vector `req` el valor de `m` (representa un petició pendent de processar)
- Si en rebre la petició hi ha treballadors disponibles, se selecciona un d'ells (per simplicitat el primer) i se li envia la petició

Tots els missatges que el broker rep des de backend tenen l'estructura `[w,'',c,'',r]`, on `w` correspon a la identitat del worker que envia la resposta, `c` al client al qual es respon, i `r` representa la pròpia resposta

- El primer missatge que envia un treballador és `['','','']`, que en arribar al broker s'arreplega com `[w,'','','','']` (ja que el socket `req` afig com a prefix un delimitador i el `router` la identitat de l'emissor). Això significa que la identitat del client és `''` (no estem responent a un client), i serveix per a diferenciar el primer missatge d'un treballador de la resposta a una petició de client
- En la resposta a una petició de client (la identitat de client no és `''`)
  - Comprovem si hi ha peticions pendents
    - Si hi ha peticions pendents (`cli.length>0`) extrau la primera petició i la passa a aquest treballador
    - Si no hi havia peticions pendents, afegim el treballador a la llista de treballadors disponibles.
  - Enviem la resposta al client

Analitza el codi proporcionat fins a comprendre el seu funcionament.

7. Possibles millores sobre el broker `router/router`

Podem plantejar diferents modificacions sobre el broker `router/router` per a millorar l'escalabilitat i/o disponibilitat. En cada proposta descrivim aspectes a resoldre i possibles estratègies, encara que no arribem necessàriament a nivell de codi. Una anàlisi detallada de les estratègies i tècniques plantejades facilita abordar altres possibles millores/ampliacions del broker.

7.1. Patró broker tolerant a fallades

Volem implementar tolerància a fallades de workers. Per a fer això hem de detectar la fallada d'un worker, i reconfigurar el sistema per a continuar funcionant sense ell.

Per a detectar la fallada podem plantejar diferents alternatives:

1. Sockets addicionals (tipus `router` en el broker, tipus `req` en cada worker), de manera que el broker envia de forma periòdica missatges denominats 'batec' (heartbeat) als treballadors, i aquests responen per a indicar que continuen vius: si la resposta tarda massa, suposem que el treballador ha caigut

2. Ídem anterior, però per a estalviar missatges només enviem batec als workers dels quals no es té notícia recent (ex. no han retornat recentment respostes a peticions de treballs)

3. Utilitzar únicament les peticions/respostes habituals: un worker es considera caigut si després d'enviar-li una petició tarda massa a retornar la resposta → en enviar una petició a un treballador, el broker utilitza `setTimeout(reconfigura, answerInterval)`: si arriba resposta es cancel · la el timeout, i si venç el timeout considerem que el treballador ha fallat i s'executa `reconfigura`

Per a no complicar el codi, assumim l'alternativa (3)

Volem transparència de fallades → quan el broker detecta la fallada del treballador `w`, reenvia la sol · licitud que estava processant `w` a un altre treballador.

- Hem de registrar en el broker totes les peticions pendents de resposta, i anotar per a cadascuna quin worker l'està processant
- Si detecta la fallada del worker `w`, el broker disposa de la informació sobre la petició que estava processant `w`, i pot reenviar-la a un altre

ftbroker.js (broker que tolera fallades 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 requests,
      sendToW(w,who.shift(),req.shift()) // process first pending req
   else
      workers.push(w)         // add as available worker
})
```

7.2. Equilibrat de càrrega

El broker basat en router/router permet remetre cada petició al treballador que vulguem. Per simplicitat, el codi proporcionat es limita a aplicar un torn rotatiu, però seria desitjable triar el treballador amb menor càrrega efectiva (equilibrat de càrrega).

En un sistema en producció cada treballador s'executa en un node diferent, i pot obtenir la càrrega del seu node per a comunicar-la al broker. Per la seua part el broker ha de rebre i mantenir informació sobre la càrrega de cadascun dels treballadors, i tenir en compte aqueixos valors a l'hora de seleccionar quin treballador gestiona una petició.

Hem de definir una estratègia per a actualitzar la informació de càrrega de cada treballador. Observem que:

- Els treballadors no necessiten informar mentre processen peticions
- Un treballador que espera peticions no modifica la seua càrrega mentre està en espera

Amb aquestes premisses, un treballador ha d'informar de la seua càrrega cada vegada que passarà a espera (ex.- després de donar-se d'alta o després de respondre a una petició):

- Un treballador notifica la seua càrrega en tot missatge 'convencional' que envia al broker → alta o resposta a una petició. No necessitem missatges específics per a comunicar la càrrega
- Quan el broker rep una petició, la redirigeix al treballador que està esperant sol · licituds i ha comunicat menor càrrega

Per a representar als treballadors en espera i la seua càrrega, tenim diferents alternatives:

1. Mantenir array desordenat (ex. inserir al final). Per a triar el de mínima càrrega cal recórrer tot el vector (cost lineal)
2. Mantenir array ordenat (càrrega creixent): complica la inserció (cost lineal), però l'elecció de mínim és trivial (el primer del vector)
3. Utilitzar una estructura de dades més elaborada (ex. min-heap), amb costos logarítmics de cerca i inserció

7.3. Tipus de treballs

Fins ara hem assumit:

- Un únic tipus de petició per part dels clients
- Workers homogenis

En conclusió els workers són equivalents → qualsevol worker pot acceptar qualsevol petició

Però en la pràctica:

- Podem tenir diferents tipus de sol · licituds
- Diferents workers poden tenir capacitats específiques

de manera que segons el tipus de sol · licitud és preferible dirigir-lo a un tipus de worker o un altre

Per a implementar aquesta especialització dels workers, assumim que:

- Els clients indiquen en cada sol · licitud el tipus de petició. En arrancar un client li passarem com a argument el tipus de peticions que realitza (ex. suposem tipus A,B,C)
- Quan un treballador es dóna d'alta, indica el tipus de peticions que pot atendre. En arrancar el treballador, li passarem com a argument el tipus de peticions que pot atendre (ex. A, B o C)

El broker classificarà els treballadors segons el seu tipus, i dirigeix cada petició a un dels treballadors que poden atendre aqueix tipus de petició. Una possibilitat és substituir la llista de treballadors per diverses llistes, una per tipus de treballador: quan arriba un treball, se cerca el treballador en la llista del tipus corresponent.