TSR - Práctica 2: **0MQ**

Curso 2022/23

Contents

1. Introducción

1.1. Objetivos

- Afianzar los conceptos teóricos introducidos en el tema 3
- Experimentar con distintos patrones de diseño (patrones básicos de comunicación) y tipos de sockets
- Profundizar en el patrón broker (proxy inverso)

1.2. Propuesta para la organización del tiempo

- Sesión 1.- Prueba de los patrones básicos y la aplicación Chat
- Sesión 2.- Publicador rotatorio, prueba del patrón broker y estadísticas broker
- Sesión 3.- Broker para clientes + broker para workers
- Sesión 4.- Prueba del patrón broker tolerante a fallos

1.3. Método de trabajo

- Por simplicidad, lanzamos los distintos componentes de cada aplicación en la misma máquina (IP = localhost)

  - Pero también se pueden repartir los componentes sobre máquinas distintas

- No debes modificar el código proporcionado ni su estructura de directorios

  - Los programas requieren argumentos en línea de órdenes
  - Dichos argumentos permiten plantear distintos escenarios
  - Puedes modificar los números de port de los ejemplos (pregunta al profesor)

- Se recomienda utilizar el fichero `refZMQ.pdf` como referencia
- En las distintas tareas se plantean cuestiones que el alumno debe responder

  - Permiten verificar que se han entendido los conceptos básicos
  - Debes averiguar la respuesta y comprender la justificación de dicha respuesta

- El fichero `fuentes.zip` contiene el código para realizar las distintas pruebas

2. Tareas

2.1. Prueba de los patrones básicos y la aplicación chat

- Todos los componentes utilizados en estas pruebas utilizan la biblioteca `tsr.js`, que se describe en el subapartado correspondiente
- En cada sub-apartado se detalla uno de los patrones básicos. Los distintos componentes de un patrón están reunidos en un mismo directorio
- Para realizar las pruebas sobre un patrón debemos abrir varios terminales (entre 2 y 5 según la prueba a realizar), situarnos en cada terminal sobre el directorio correspondiente al patrón, y ejecutar la orden que se indica para cada terminal

2.1.1. Biblioteca `tsr.js`

- Para simplificar las pruebas de los distintos patrones y el código de cada componente suponemos definida la siguiente biblioteca `tsr.js`, que importa la biblioteca `zeromq` y exporta las funciones:

  - `error(msg)` Muestra el mensaje de error y finaliza la ejecución del programa
  - `lineaOrdenes(params)` Comprueba que la línea de órdenes contiene los parámetros indicados, y crea las variables correspondientes
  - `traza(f,nombres,valores)` Muestra el valor de los argumentos cuando se invoca la función f
  - `adios(sockets,despedida)` Cierra los sockets indicados. muestra el mensaje de despedida, y finaliza el programa
  - `creaPuntoConexion(socket,port)` Aplica la operación `bind` sobre el port indicado, y verifica el resultado
  - `conecta(socket,ip,port)` Aplica la operación `connect` sobre `tcp://ip:port`

2.1.2. Prueba del patrón cliente/servidor (`req/rep`)

Nos situamos en el directorio `req-rep`, donde encontramos los ficheros `cliente1.js`, `cliente2.js` y `servidor.js`

- Un cliente y un servidor

  - terminal 1) `node servidor.js A 9990 2 Hola`

  - terminal 2) `node cliente1.js localhost 9990 Pepe`
  - Preguntas
    * ¿Qué ocurre si pasamos un número de argumentos incorrecto? ¿y si están fuera de orden?
    * Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón
    * En relación con los mensajes multi-segmento:
      · ¿De qué forma construye el emisor un mensaje multi-segmento?
      · ¿Cómo accede el receptor a los distintos segmentos del mensaje?
    * El cliente finaliza tras recibir la respuesta a la cuarta petición. ¿Cuando termina el servidor?

- Un cliente y dos servidores
  - terminal 1) `node servidor.js A 9990 2 Hola`
  - terminal 2) `node servidor.js B 9991 2 Hello`
  - terminal 3) `node cliente2.js localhost 9990 localhost 9991 Pepe`
  - Preguntas
    * Comprueba si el orden de arranque afecta al resultado. Indica la razón
    * ¿Qué ocurre si ambos servidores reciben el mismo número de port?
    * ¿Qué ocurre si los dos servidores reciben un valor de segundos distinto (ej 1 y 3 respectivamente)?. ¿Afecta al orden en que se responde al cliente?
    * La práctica totalidad del tiempo lo consumen los servidores ¿Conseguimos reducir a la mitad el tiempo de ejecución del cliente al utilizar 2 servidores?
    * Si queremos usar 3 servidores, ¿hay que modificar el código del cliente?
    * Con un número de peticiones par, ¿podemos garantizar que cada servidor atiende la misma cantidad de peticiones?

- Dos clientes y un servidor
  - terminal 1) `node servidor.js A 9990 2 Hola`
  - terminal 2) `node cliente1.js localhost 9990 Pepe`
  - terminal 3) `node cliente1.js localhost 9990 Ana`
  - Preguntas
    * Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón
    * ¿Podemos asegurar que cada cliente recibe únicamente la respuestas a sus propias peticiones?. Indica la razón
    * En caso de plantear una cantidad distinta de clientes (ej 3), ¿sería necesario modificar el código del cliente o del servidor?
    * En caso de que uno de los clientes termine antes de tiempo (ctrl-C), ¿el otro sigue recibiendo respuestas?. Indica la razón

2.1.3. Prueba del patrón pipeline (push/pull)

- Nos situamos en el directorio `push-pull`, donde encontrarás los ficheros `origen1.js`, `origen2.js`, `filtro.js` y `destino.js`
- origen1 -> destino A-B

  - terminal 1) `node origen1.js A localhost 9000`
  - terminal 2) `node destino.js B 9000`
  - Preguntas
    * Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón
    * Indica la razón por la que el socket definido en origen.js no define socket.on('message',..)

- origen1 -> filtro -> destino A-B-C

  - 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`
  - Preguntas
    * Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón
    * Indica la razón por la que origen.js y destino.js definen un único socket cada uno, pero filtro.js define 2 sockets
    * Si origen genera 4 mensajes y filtro retarda 2 segundos, ¿cúanto crees que tarda el último mensaje de origen en llegar a destino?

- origen2 -> [filtro, filtro] -> destino A-(B,C)-D

  - 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`
  - Preguntas
    * Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón
    * ¿Cómo se reparte la entrega de mensajes a los filtros B y C?
    * ¿Pueden trabajar B y C en paralelo (ej. si se ejecutasen en máquinas distintas)?
    * ¿En qué orden llegan los mensajes a destino?. ¿Cómo afectaría al comportamiento modificar los segundos del filtro C?

2.1.4. Prueba del patrón difusión (pub/sub)

Nos situamos en el directorio `pub-sub`, donde encontramos los ficheros publicador.js y suscriptor.js

- terminal 1) `node publicador.js 9990 Economia Deportes Cultura`
- terminal 2) `node suscriptor.js A localhost 9990 Economia`
- terminal 3) `node suscriptor.js B localhost 9990 Deportes`
- terminal 4) `node suscriptor.js C localhost 9990 Economia`
- Preguntas
  - Indica si el orden en que arrancamos los componentes afecta al resultado
  - ¿Qué pasa con los mensajes de Cultura?
  - ¿Puede recibir el mismo mensaje más de un suscriptor?
  - ¿Cómo se puede cambiar la cantidad de mensajes que genera el publicador?
  - Los suscriptores no terminan. Piensa en una modificación para que terminen tras un tiempo determinado sin recibir mensajes

- Es posible que el publicador genere algún mensaje cuando todavía no ha procesado las conexiones de los suscriptores. ¿Qué pasa con esos mensajes?

2.2. Prueba aplicación chat

En el directorio `chat` encontrarás la implementación de una aplicación chat rudimentaria. Analiza el código y comprueba su funcionamiento.

- 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`
- Preguntas
  - ¿En qué afecta el orden en que arrancamos los componentes?
  - Indica la razón por la cual ambos puntos de conexión se crean en el servidor
  - El servidor no mantiene una lista de clientes conectados. ¿Por qué razón?
  - Piensa cómo modificar un cliente para que atienda únicamente mensajes de algunos temas concretos

2.3. Publicador rotatorio

Utilizando el patrón `pub/sub`, desarrolla un programa `publicador.js` que se invoca como

```
node publicador.js port numMensajes tema1 tema2 tema3 ...
```

donde:

- `port` = el port al que deben conectarse los suscriptores (el host es `localhost`)
- `numMensajes` = número de mensajes a emitir, tras lo cual el publicador termina
- `tema1 tema2 tema3 ...` = número variable de temas (a priori no sabemos cuántos)

Debe generar un mensaje por segundo con la estructura `[tema, numMensaje, numRonda]`

- tema (en orden circular)
- número de mensaje
- ronda (primera iteración sobre los temas, segunda, etc)

Ejemplo-

```
node publicador 8888 5 Politica Futbol Cultura
```

debe emitir (cada mensaje en un segundo):

```
Politica 1 1
Futbol 2 1
Cultura 3 1
Politica 4 2
Futbol 5 2
```

2.4. Tareas sobre el patrón broker/Workers

2.4.1. Prueba del patrón broker/workers

En el directorio `broker-worker` encontrarás varios ficheros, pero para esta práctica únicamente utilizamos `cliente.js`, `brokerRouterRouter.js`, y `workerReq.js` Analiza el código y comprueba su funcionamiento:

- terminal 1) `node brokerRouterRouter 9000 9001`
- terminal 2) `node workerReq w1 localhost 9001`
- terminal 3) `node workerReq w2 localhost 9001`
- terminal 4) `node cliente A localhost 9000`
- terminal 5) `node cliente B localhost 9000`
- Preguntas:
  - ¿Cómo afecta al resultado cambiar el orden de arranque de los workers (terminales 2 y 3?. ¿Y de los clientes (terminales 4,5)?
  - ¿Qué pasa si arrancamos el broker al final (el 1) pasa al 5))

2.4.2. Estadísticas broker

Modifica el código de `brokerRouterRouter.js` para mantener el total de peticiones atendidas y el número de peticiones atendidas por cada worker

- El broker debe mostrar dicha información en pantalla cada 5 segundos
- Preguntas
  - Indica una estrategia para mantener en el broker estadísticas separadas para cada worker
  - Indica cómo conseguir que se ejecute una acción de forma repetida (periódica)
  - Si llega una petición y se la pasamos al worker w, ¿debemos incrementar el número de peticiones atendidas por w (y el total) en ese momento, o cuando llegue la respuesta desde w?

2.4.3. Broker para clientes + Broker para workers

Tomamos como punto de partida el fichero `brokerRouterRouter.js` disponible en `fuentes.zip` directorio `broker-workers`

- Reescribe el código para tener dos brokers interconectados entre sí a los que llamamos `broker1.js` y `broker2.js`
- La solución elegida debe mantener las mismas características externas (o sea, frente a clientes y frente a workers) que el código original

Concretamente:

- `broker1` conoce a los clientes, y mantiene la cola de peticiones pendientes
- `broker2` conoce a los brokers, y se responsabiliza de mantener los workers disponibles y repartir la carga
- Todo cliente envía su petición a `broker1`, que la guarda en la cola de peticiones pendientes o la pasa a `broker2`
  - NOTA.- `broker1` no sabe qué workers están disponibles, pero podemos organizar el código para que sepa cuántos están disponibles
- Cuando llega una petición a `broker2`, éste la envía al worker correspondiente
- La respuesta del worker llega a `broker2`, que la pasa a `broker1`, y éste al cliente
- El alta de un worker llega a `broker2` (y si se considera necesario se puede informar a `broker1`)

Debes decidir cómo se comunican broker1 y broker2 (hay más de una alternativa)

- Preguntas
  - ¿Qué alternativas podemos usar para comunicar entre sí los dos brokers?

  - ¿Es conveniente avisar a broker1 cuando se da de alta un worker?
  - ¿Es conveniente avisar a broker2 cuando llega una petición y no hay workers disponibles?

2.4.4. Broker tolerante a fallos de workers

En la carpeta `brokerToleranteFallos` encontramos una nueva versión del fichero `broker.js` que es capaz de gestionar algunas situaciones de fallo.

Probamos el broker 'normal'. Nos situamos en el directorio broker-worker y lanzamos

- terminal 1) `node brokerRouterRouter 9000 9001`
- terminal 2) `node workerReq w1 localhost 9001`
- terminal 3) `node workerReq w2 localhost 9001`
- terminal 4) `node workerReq w3 localhost 9001`
- terminal 2) `ctrl-C`
- terminal 5) `node cliente A localhost 9000 & node cliente B localhost 9000 & node cliente C localhost 9000`
- Preguntas
  - ¿Cuantas respuestas se obtienen?. Indica qué trabajadores las han enviado
  - ¿Quedan clientes esperando?

Repite la prueba, pero esta vez nos situamos en el directorio `brokerToleranteFallos`

- terminal 1) `node broker.js 9000 9001`
- el resto igual que en la prueba anterior
- Preguntas
  - ¿Quedan clientes esperando?
  - ¿El cierre (caída) del worker es transparente para el cliente?
  - Únicamente se aborda posibles fallos de workers. Indica si se puede aplicar alguna estrategia ante un posible fallo del broker