**Carolina Alba Marugan Rubio**

**REQ/REP**

– terminal 1) node servidor.js A 9990 2 Hola

– terminal 2) node cliente1.js localhost 9990 Pepe

El programa Cliente1 envía 4 peticiones con el bucle for avisando de ello a consola. Cuando procesa la respuesta, la imprime, y si es la 4 iteración, se despide.

Nota: la función lineOrdenes de tsr.js crea variables globales asociadas a los parámetros pasados por orden en el terminal. En server la variable segundos indica cada cuántos segundos mandas la respuesta

* Comprueba si el orden en que arrancamos los componentes afecta al resultado.

El orden no afecta. Por un lado, el método connect() es asíncrono, no bloqueante. Así, Aunque se haga primero bind() y después connect() o alrevés, el código continúa. En adición a este punto, cito pdf: “No hay restricciones sobre el orden en que se realice bind() y connect(). En los dos posibles órdenes (bind() precede a connect() o connect() precede a bind()), la conexión se establece correctamente. Además, tampoco hay restricciones sobre el tiempo que pueda transcurrir entre las llamadas a esas operaciones.”. Por otro lado, al haber únicamente un Cliente, la única petición que le llegará al servidor será de este, así que el orden de respuesta no variará con respecto a si otro cliente mandara primero la petición, porque el orden de llegada a REP es FIFO, y teniendo en cuenta que se trata de un socket síncrono, hasta que el server no le responda a ese cliente, no atenderá más peticiones.

* ¿Qué ocurre si pasamos un número de argumentos incorrecto? ¿y si están fuera de orden?

Si le pones menos o más argumentos al cliente te dice “parámetros incorrectos” y te indica cual debe ser el uso. Lo mismo hace el servidor

Si están fuera de orden el cliente no puede conectarse porque no encuentra el puerto al que conectarse, pero el servidor funciona solo que no interpreta tiempo de espera para dar respuesta. Claro que depende de qué parámetros cambies.

* En relación con los mensajes multi-segmento:

· ¿De qué forma construye el emisor un mensaje multi-segmento?

Con req.send([nombre,i]) le pasa un array

· ¿Cómo accede el receptor a los distintos segmentos del mensaje?

Cuando recibe un mensaje en el listener, ya recibe cada segmento como un parámetro. De manera que en el callback los distingue por argumentos de la función, solamente tiene que llamar al parámetro.

* El cliente finaliza tras recibir la respuesta a la cuarta petición. ¿Cuando termina el servidor?

El servidor solo termina si desde la terminal haces cntrl+C, llamando a la función adios()

– 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

Nota: el cliente2.js hace 2 operaciones conecta()

* Comprueba si el orden de arranque afecta al resultado. Indica la razón

Afecta en cierta medida. El cliente siempre manda primero la petición al servidor que tiene primero en los parámetros de terminal. Esto es porque se conecta primero al que tiene ipServidor1. Independientemente del orden de arranque, siempre manda mensaje primero al que primero se conecta. Es por esto, que hasta que no reciba la respuesta del primero al que envía el mensaje, por mucho que arranque el otro servidor, hasta que no arranque el primero y le responda, no mandará el próximo mensaje al segundo servidor. También, aunque arranque primero el servidor al que primero envía mensaje y este me responda, hasta que no arranque el segundo, no continuará con los siguientes envíos.

En resumen, afecta tanto el orden de arranque como el orden en el que el cliente se conecta a los servidores.

* ¿Qué ocurre si ambos servidores reciben el mismo número de port?

Sí que puedes conectarte dos veces al mismo puerto, como cliente. Recibes las peticiones exactamente igual, pero viniendo del mismo servidor. Pero si como servidor intentas conectarte a un puerto en el que ya has hecho antes bind(), te sale error “address already in use”.

* ¿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?

Ocurre lo mismo que antes pero la que tiene 3 segundos tarda más. Pero claro que no influye en el orden de llegada de la respuesta, porque al ser un socket req, no puede mandar la siguiente petición hasta que no le ha llegado la respuesta de la petición que ya ha mandado.

* La práctica totalidad del tiempo lo consumen los servidores ¿Conseguimos reducir a la mitad el tiempo de ejecución del cliente al utilizar 2 servidores?

No, porque no has paralelizado ninguna tarea. Hasta que no recibes la respuesta a tu petición, no puedes mandar la siguiente. Por tanto, te da igual si tienes dos servidores, el tiempo de respuesta y petición va a ser el mismo.

* Si queremos usar 3 servidores, ¿hay que modificar el código del cliente?

Sí, porque se tiene que conectar a él, y cuando introduzcamos la orden por el terminal, deberemos poner también la dirección y puerto del tercer servidor. Pero el resto, no cambia.

* Con un número de peticiones par, ¿podemos garantizar que cada servidor atiende la misma cantidad de peticiones?

Si tenemos 2 servidores, número par, y el número de peticiones es un múltiplo de 2, también número par, entonces podemos asegurar que cada servidor atiende la misma cantidad de peticiones. Ya que, el algoritmo que sigue el socket req para enviar peticiones es el Round-Robin, garantizando, en este caso, un reparto igual en la carga.

– 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

* Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón

Sí que afecta, pues el algoritmo que sigue el socket rep para atender peticiones es fair queueing, FIFO. Así, la petición que le llegue antes será la que primero conteste. Puede ocurrir que atienda dos peticiones seguidas del mismo cliente. Y que, habiendo arrancado primero un cliente1 y luego cliente2 y luego el servidor1, me atienda primero al cliente2.

* ¿Podemos asegurar que cada cliente recibe únicamente la respuestas a sus propias peticiones?. Indica la razón

Sí lo puedes asegurar. Ya que, el servidor, no puede atender otra petición hasta que no ha respondido la actual. Por tanto, por muchos mensajes que manden otros clientes, el servidor me responderá, y después recibirá su siguiente petición. De esta manera, otras peticiones no interfieren en la respuesta que me da el servidor.

* En caso de plantear una cantidad distinta de clientes (ej 3), ¿sería necesario modificar el código del cliente o del servidor?

No sería necesario modificar el servidor, él escucha en su puerto y responde peticiones, independientemente de quien vengan. Si creamos otro cliente, habrá que indicarle por orden de terminal un nombre diferente al de los otros, pero el código es el mismo.

* En caso de que uno de los clientes termine antes de tiempo (ctrl-C), ¿el otro sigue recibiendo respuestas?. Indica la razón

Claro que sí. La respuesta del servidor depende de si le llegan los mensajes o no. Si no le llega un mensaje de un cliente, simplemente le llegará otro al que responderá. Este último cliente, si lo desea, podrá mandar otro, pues es independiente del otro cliente. En caso de que un cliente finalice justo antes de recibir la respuesta del servidor, esto no impide al servidor enviar la respuesta. De este modo, no se bloquea la recepción de posteriores peticiones. En caso de que un servidor no pueda responder, entonces sí se bloquearían las siguientes peticiones. Pero esto difícilmente vendrá causado por un cliente, más bien, por problemas del servidor…

**PUSH/PULL (patrón pipeline)**

– terminal 1) node origen1.js A localhost 9000

– terminal 2) node destino.js B 9000

Nota: curiosa la forma en que destino.js maneja lo del filtro

* Comprueba si el orden en que arrancamos los componentes afecta al resultado.

No afecta. El orden de bind() y connect() es igual que en el caso inicial. Es importante destacar que push pull es asíncrono y el algoritmo que sigue push para enviar respuestas es circular, por lo que si hubiera más clientes habría que tener en cuenta que push mandaría la respuesta al siguiente cliente.

* Indica la razón por la que el socket definido en origen.js no define socket.on(‘message’,..)

Pues porque ni el socket pull tiene la capacidad de enviar mensajes, ni el socket push de recibirlos.

– 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

Nota: filtro es el intermediario, con 2 sockets: pull recoge los mensajes de origen, y push los reenvía a destino, añadiéndole su nombre B para indicar que es el filtro. Así, destino recibe 3 parametros y sabe que el mensaje viene del filtro, aunque también conserva el nombre de A

* Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón

No le afecta. Destino simplemente recibe mensajes, lo arranques antes o después no afecta porque hasta que no reciba un mensaje no hará nada. Filtro lo mismo con origen, hasta que origen no le mande nada, no reenviará nada. Destino una vez arrancado, aunque no haya nadie escuchando, el puede mandar mensajes, y si hay, pues los reciben.

* Indica la razón por la que origen.js y destino.js definen un único socket cada uno, pero filtro.js define 2 sockets

Porque si filtro actúa de intermediario, necesita tener la habilidad de recibir mensajes y enviarlos. Algo que con un único socket de tipo pull o push no es posible. Solo es posible si se definen dos sockets, uno de cada tipo. Origen solamente necesita enviar y destino solamente recibir, por lo que con un único socket es suficiente.

* Si origen genera 4 mensajes y filtro retarda 2 segundos, ¿cúanto crees que tarda el último mensaje de origen en llegar a destino?

Tarda dos segundos. La llegada a filtro desde origen es inmediata porque origen no aplica ningún retardo a cada mensaje y filtro trata cada mensaje con un callback que se retarda 2 segundos. Pero el socket pull no necesita, evidentemente, responder a ningún mensaje para poder recibir el siguiente, como sí pasaba con el socket rep. Por ello, filtro recibe los mensajes por orden de llegada y el único retardo que se les aplica es el del callback, 2 segundos. Pero es: llegan, aplica callback, llega, aplica callback… la aplicación del callback es inmediata.

– 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

* Comprueba si el orden en que arrancamos los componentes afecta al resultado. Indica la razón

El orden de arranque sí que afecta porque destino imprime los primeros que le llegan, si arranco primero B y aún no he arrancado C, le llegan primero los de B. Eso habiendo arrancado antes el origen, que, recordemos, utiliza R-R pero manda primero el primer mensaje a quién primero ha establecido la conexión. Si primero origen se conecta a B pero se arranca primero C y a destino le llegan primero los de C, el destino imprimirá la iteración 2 y 4.

* ¿Cómo se reparte la entrega de mensajes a los filtros B y C?

Con el algoritmo Round-Robin, distribuyendo así la carga e independientemente de cuál se haya arrancado primero o de si están arrancados. Es cierto que primero se envía al que primero se ha establecido la conexión, luego se sigue un orden rotatorio.

* ¿Pueden trabajar B y C en paralelo (ej. si se ejecutasen en máquinas distintas)?

En el sentido de que pueden recibir mensajes a la vez (independientemente del nº de clientes) sí, pues cada uno puede también enviar mensajes al mismo destino a la vez (otra cosa es en qué orden se procesen en destino…).

* ¿En qué orden llegan los mensajes a destino?. ¿Cómo afectaría al comportamiento modificar los segundos del filtro C?

Pues el filtro que primero envíe el mensaje, generalmente, será el que primero le llegue al destino, primero en llegar. Depende del orden de arranque también.

Si por ejemplo arrancas primero B pero le pones 10segundos y arrancas después C pero le pones 1 segundo, a destino llegan antes los de C. Así se demuestra que lo que importa es el orden de llegada a destino.

**PUB/SUB (patrón difusión)**

• 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

Nota: ten en cuenta que publicador utiliza la función Math.trunc() para elegir un componente diferente del array y utiliza la i como índice, que va variando. Así, cada vez se hace send() para un tema diferente

* Indica si el orden en que arrancamos los componentes afecta al resultado

Claro que afecta, porque un suscriptor que no está conectado, no va a recibir los mensajes aunque se conecte después. El publicador los envía y los reciben los que en ese momento están conectados. No quedan almacenados los mensajes que no se reciben.

* ¿Qué pasa con los mensajes de Cultura?

Pues que no podemos verlos porque no tenemos ningún suscriptor suscrito a ese tema. Se envían pero no los recibe nadie porque nadie está suscrito.

* ¿Puede recibir el mismo mensaje más de un suscriptor?

Sí, si dos suscriptores están suscritos al mismo tema, ambos recibirán el mismo mensaje.

* ¿Cómo se puede cambiar la cantidad de mensajes que genera el publicador?

El publicador genera mensajes empleando el setTimeout() (que se va llamando cada vez que publica) y una condición del valor de la i. Cuando la i vale 10, ya no se vuelve a llamar al settimeout(), se ha enviado el último mensaje y se despide (entonces se mandan 11 mensajes porque se empieza desde 0). Por tanto, para cambiar la cantidad de mensajes podríamos o bien darle otro valor inicial a la i, o bien cambiar el valor de la i en la condición del if.

* Los suscriptores no terminan. Piensa en una modificación para que terminen tras un tiempo determinado sin recibir mensajes

Es utilizando un TimeOut asignándolo a una variable. Una vez recibes un mensaje, haces clearInterval(nombre variable), y luego vuelves a establecer el setTimeOut()

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

Sí, él va enviando los mensajes. Estos mensajes no son recibidos por nadie.

**APLICACIÓN CHAT**

• 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

Nota: servidor usa un socket pull para recibir mensajes y otro pub para publicarlos, ambos crean puntos de conexión. Cliente usa un socket push para mandar mensajes y otro sub para recibir los publicados, ambos se conectan a un servidor y puerto.

* ¿En qué afecta el orden en que arrancamos los componentes?

Servidor no va a mandar ningún mensaje hasta que no le llegue uno, por cómo está implementado. Lo que sí afecta es en el orden en el que sale que un cliente se ha conectado. Solamente el cliente que ya estaba conectado puede ver que el cliente nuevo se acaba de conectar, los que se acaban de conectar no ven quiénes estaban conectados.

* Indica la razón por la cual ambos puntos de conexión se crean en el servidor

Porque tú para publicar, necesitas indicar en qué lugar vas a crear la conexión, para que la gente se conecte y tú puedas difundir a quien esté conectado a ti. Y para recibir un único mensaje, quien te lo quiera enviar se ha de conectar a ti. En cualquier caso, es el patrón que sigue el pull y el sub.

* El servidor no mantiene una lista de clientes conectados. ¿Por qué razón?

Porque tú como servidor solo te enteras de que ese cliente existe si te manda un mensaje, en ese caso puedes sacarle el Nick e ir elaborando la lista de los que te mandan mensaje. Pero simplemente por conectarse, al menos a nosotros no nos han enseñado una forma de hacerlo.

* Piensa cómo modificar un cliente para que atienda únicamente mensajes de algunos temas concretos

Creo que sería únicamente modificando entrada.subscribe(“”) por entrada.subscribe(“temas concretos”)

**PUBLICADOR ROTATORIO**

Hecho en código

**BROKER/WORKERS**

• 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

Nota: cliente se conecta a brokerRouterRouter y se establece a él mismo una identidad y se la manda, una vez recibe respuesta del bróker se despide. BrokerRR tiene un array para los workers disponibles y otro para los clientes pendientes de respuesta, crea dos puntos de conexión, uno para los clientes y otro para los workers, tiene una función para procesar petición de clientes y otra para procesar petición de workers (porque tanto cliente como worker tienen socket req), la función para procesacliente reenvía petición del cliente a un worker si hay alguno en el array workers y elimina a este del array workers, y si no hay workers disponibles, añade al cliente en array clientes pendientes; la función procesaworker comprueba si hay algun cliente pendiente, lo elimina del array pendientes y le manda a ese worker el mansaje del cliente, si no hubiera clientes pendientes, se añade al worker en el array workers disponibles y cada vez que llega un mensaje del worker, además de asignarle a continuación una petición del cliente que estaba pendiente, también se envía al cliente la respuesta que ha mandado este worker (en caso de que la variable cliente no estuviera vacía). Worker también establece la identidad de su socket req, se conecta al bróker y le manda 3 segmentos vacíos (el router no va a eliminar ninguno, solo va a añadir la id), cuando worker reciba petición solo podremos ver de qué cliente viene la petición y su mensaje (porque el resto de la info que me había mandado el router se ha borrado con el delimitador), cuando el worker responde únicamente añade al final del mensaje (el resto reenvía lo mismo que le llega) su id, una vez pasado un segundo.

Desde el router, cada vez que se le envía una tarea a un worker, se le elimina del array workers disponibles, y solo se le añade (al final del array) cuando este responde con la tarea solucionada, garantizando así un orden rotatorio de procesado de tareas (suponiendo que ambos workers están disponibles).

**Funcionamiento socket router**: cuando le llega un mensaje, el socket no solo añade un identificador, sino que tampoco elimina el delimitador. Por lo que, desde código, puedo acceder tanto al id como al delimitador y al mensaje que originalmente me ha enviado el cliente.

Para enviar, el socket no me añade automáticamente el delimitador, por lo que tengo que ponerlo desde código, y también tengo que especificarle al principio el id de quien quiero enviar (aunque luego el socket consuma el id).

**Funcionamiento de la recepción de mensajes vinientes de router por parte de sockets req**: todo lo que quede delante del primer delimitador, lo eliminará y desde su código no se podrá ver (aunque se lo guarde).

* ¿Cómo afecta al resultado cambiar el orden de arranque de los workers (terminales 2 y 3?. ¿Y de los clientes (terminales 4,5)?

Pues que el primero que se arranque será el que primero envíe el mensaje avisando de que está disponible, y será el primero al que se le asignará la primera petición del cliente. El cliente que primero se arranque será el que primero se reenviará su respuesta. Porque tanto clientes como workers se añaden al final del array del router, garantizando un orden FIFO.

* ¿Qué pasa si arrancamos el broker al final (el 1) pasa al 5))

No tendría por qué pasar nada, le llegarían las peticiones al router y una vez arranque las atiende. Por orden de llegada.

**Estadísticas bróker**

* Indica una estrategia para mantener en el broker estadísticas separadas para cada worker

Pues cuando procesas el mensaje del worker, lo diferencias por el valor del id del worker

* Indica cómo conseguir que se ejecute una acción de forma repetida (periódica)

No sé a qué acción se refiere… Pero supongo que con setInterval…

* 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?

Cuando llegue la respuesta desde w, así lo he implementado yo porque que se la envíes no significa que te vaya a responder.

**Broker1+Broker2**

Hecho en código

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

La cuestión es usar sockets asíncronos para que se comuniquen. Si no, no puedo avisar a broker1 si me llegan dos workers seguidos hasta que broker1 me conteste, no sería practico (usando sockets síncronos). Si usamos sockets pull/push también se puede pero tendríamos que tener dos sockets en cada bróker (como en este caso solamente habría un bróker al otro extremo no pasaría nada). En el caso de pub/sub, habría que añadir el prefijo de suscripción y tener en cuenta el orden en el que ponerlo por el identificador que añade el router (sinceramente tampoco sería muy práctico).

* ¿Es conveniente avisar a broker1 cuando se da de alta un worker?

Conforme lo he implementado yo, sí. Para que aumente el nº de workers disponibles.

* ¿Es conveniente avisar a broker2 cuando llega una petición y no hay workers disponibles?

No es necesario, podemos enviarle peticiones únicamente cuando sabemos que hay workers disponibles.

**Broker tolerante a fallos de workers**

• 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

* ¿Cuantas respuestas se obtienen?. Indica qué trabajadores las han enviado

Se obtienen 2 respuestas. Vienen de w2 y w3b

* ¿Quedan clientes esperando?

No quedan clientes esperando, se termina el programa. Si se cierra un worker abruptamente, el cliente inicial no se queda esperando.

• terminal 1) node broker.js 9000 9001

• el resto igual que en la prueba anterior

Nota: cada vez que se recibe mensaje de cliente, si no hay workers disponibles se asigna a pendiente. Si hay, se asigna nueva tarea al primer worker disponible. En new\_task se inicia el timeout para failure() y se guarda en el array de workers en ese worker, y se manda al worker. Si efectivamente ha habido fallo, se pone a true en la posición del worker del array failure y se vuelve a llamar a dispatch para ver si hay workers libres y demás. Si no ha habido fallo, se recibirá el mensaje del worker y se hace clearTimeout(working[worker]) que es donde teníamos el setTimeout() y además también se borra de working porque ya no está trabajando. Si hay clientes en pendiente, a este worker que acaba de llegar se le asigna nueva tarea y si no se le añade al array de ready porque está listo, además se le enviará al cliente la respuesta de este worker. En el caso de que el worker no hubiera fallado pero hubiera tardado más tiempo en responder que el intervalo establecido, se ignorará su respuesta y se habrá marcado como fallido. Por lo tanto, siempre que a un worker se le haya asignado como fallido, ya nunca se volverá a interactuar con él.

* ¿Quedan clientes esperando?

Si por esperando se refiere a inatendidos, no. Pero una vez atendidos los mensajes que no han fallado, se sale del proceso cliente e inicia otro para responder al cliente que había quedado sin responder, y se le ha asignado uno de los workers que quedaban activos.

* ¿El cierre (caída) del worker es transparente para el cliente?

En principio sí.

* Únicamente se aborda posibles fallos de workers. Indica si se puede aplicar alguna estrategia ante un posible fallo del broker

Se puede hacer algo parecido , con el setTimeout, lo inicias cuando envías una petición de cliente a un worker, y cuando el mismo worker te responde a esa tarea hacer clear timeout (teniendo los mismos arrays que en el brokertoleranteafallos). También se puede controlar desde el cliente, cuando se envía petición el bróker se inicia el timeout y si responde el bróker se hace clear timeout, si no, se vuelve a enviar peticion.