Tema 3 – Middleware. ZeroMQ

Guía del alumno

1. Introducción

En todo sistema distribuido de gran escala habrá un alto número de componentes que tendrán que colaborar entre sí para desarrollar cierta funcionalidad. Algunos de estos componentes podrán reaprovecharse de otras aplicaciones distribuidas desarrolladas anteriormente. Por tanto, no será raro que en la implantación de un determinado sistema distribuido participen múltiples equipos de programadores, cada uno especializado en unos pocos componentes del sistema. Esto implica que tanto la especificación como el desarrollo de cada componente de un sistema distribuido podrían realizarse de manera independiente, tanto temporal (pues los componentes se habrán desarrollado durante diferentes plazos) como física (pues cada equipo de desarrollo habrá trabajado en una empresa o lugar distinto).

Una vez esos componentes hayan sido implantados se tendrá que pasar a desplegarlos sobre cierto conjunto de ordenadores. El número de instancias a desplegar de cada componente dependerá de la carga prevista por el administrador del sistema. Esa carga, a su vez, depende del tipo de servicio ofrecido y del éxito que llegue a tener. Por ejemplo, la Wikipedia es actualmente un servicio altamente escalable, utilizado concurrentemente por miles de usuarios. Su despliegue requiere un número muy alto de instancias. Por ello, resulta necesario automatizar su propio despliegue mediante el uso de herramientas especializadas.

Tanto en un aspecto (desarrollo) como en el otro (despliegue) interesará que las interfaces de los componentes permitan una interacción sencilla y fiable. Para ello no bastará con conocer esas interfaces; también se necesitará utilizar un mismo protocolo que permita esa interacción. Aquí ya existe una primera dificultad: el protocolo a utilizar debe ser común para todos los componentes que deban interconectarse. Eso implica que no se podrá improvisar un protocolo de interacción, sino que habrá que adoptar alguno ya estandarizado. Si se adopta un protocolo estándar con una especificación clara resultará sencillo implantar los componentes que le den soporte y lo utilicen. Es más, con algo de suerte se podrá utilizar alguna implementación existente de los módulos que lo soporten. Así se simplificará el desarrollo de los componentes de esa nueva aplicación distribuida de gran escala.

El hecho de que haya muchos componentes en una aplicación de este tipo y que estos deban interactuar entre sí aconseja modelar estas aplicaciones como conjuntos de agentes. Se dice que son *agentes* porque no son entidades pasivas, sino activas, y cada uno de estos agentes interactúa con otros (es decir, con otros agentes o componentes del sistema) estableciendo dependencias. En una dependencia determinada cada agente ejercerá un rol. Por ejemplo, en una interacción en la que se transmita información de un componente a otro, el agente que genere la información actuaría como "*productor*" mientras que el agente que la reciba actuaría como "*consumidor*". Existen múltiples escenarios de interacción posibles. Algunos de ellos son ya escenarios clásicos en la computación concurrente, con restricciones bien conocidas que regulan el comportamiento correcto de los agentes. Por ejemplo, una interacción productor-consumidor con un *buffer* de capacidad limitada [1] establece unas normas de sincronización que ya fueron descritas en otras asignaturas (CSD).

Aunque la escalabilidad es un objetivo frecuente en las aplicaciones distribuidas, en algunos casos encontraremos aplicaciones compuestas por unos pocos módulos. Incluso en esos casos habrá que ser cuidadoso para que ese conjunto de módulos esté implantado de una manera fiable, sin errores. Debe recordarse que la depuración de aplicaciones distribuidas suele ser más compleja que la de aplicaciones diseñadas para un solo ordenador (generalmente compuestas por un solo ejecutable). Existen varios motivos: son aplicaciones concurrentes que exigen algún tipo de sincronización a la hora de acceder a recursos compartidos, se necesita comunicación a través de la red introduciendo retardos en las interacciones, la coordinación entre actividades resulta más compleja, debería soportarse el fallo de cualquiera de los

componentes... Una aplicación distribuida fiable debería comportarse bien en todas esas situaciones.

Revisando todo lo que se ha dicho hasta el momento comprobamos que:

- Las aplicaciones distribuidas no pueden implantarse mediante un único programa de pequeño tamaño. Se necesitan múltiples componentes, normalmente desarrollados por varios equipos de programadores.

- Deben utilizarse protocolos de interacción entre los componentes.

- Los programadores deberán completar el desarrollo respetando ciertos plazos.

La conclusión es que no conviene reimplantar desde cero las nuevas aplicaciones distribuidas. Habría que preocuparse por una infinidad de detalles que en ocasiones anteriores ya se habrán resuelto y comprobado. Un diseño modular permitirá reutilizar esos componentes fiables, así como sus protocolos de interacción.

Como ejemplos de esos detalles de bajo nivel que deben resolverse en la mayoría de las aplicaciones distribuidas cabría citar los siguientes:

- Cómo podrá encontrar un proceso cliente al proceso servidor que atienda sus peticiones. Generalmente el proceso cliente conoce el identificador o nombre del servicio que le interesa. Ese nombre debería traducirse a una dirección. Esa es una tarea que podrá resolver un servidor de nombres distribuido. Sin embargo, puede que el servicio sea escalable y esté implantado por un servidor replicado. En ese caso habría múltiples réplicas servidoras, cada una ubicada en una máquina con una dirección distinta. De algún modo debe decidirse a qué máquina acceder. También debería conocerse el puerto al que dirigir las peticiones. Por tanto, no resulta trivial la gestión de este problema.

- Cómo especificar la funcionalidad de un servicio. Generalmente los servicios distribuidos son altamente disponibles. Es decir, garantizan la continuidad de su servicio, por lo que no debería admitirse que se detengan, ni siquiera cuando falle alguna de sus réplicas o cuando resulte necesario reconfigurarlos. Sin embargo, los requisitos de estos servicios suelen evolucionar con el tiempo. Por ello, de vez en cuando habrá que modificar su API y extender o adaptar su funcionalidad; es decir, actualizar dinámicamente [2] el servicio. Esto obligará a que clientes y servidores utilicen algún mecanismo para comunicar la versión del API que esperan y proporcionan, respectivamente. Habrá que comprobar que esas versiones sean compatibles.

- Debe utilizarse un formato conocido por clientes y servidores para el contenido del mensaje. Esto guarda relación con el protocolo de interacción entre los agentes emisores y receptores de los mensajes. Ese protocolo suele ubicarse en el nivel de aplicación (o, al menos, está por encima del nivel de transporte). En el protocolo se establecerá el formato de los mensajes a utilizar. Ese protocolo debe facilitar también que los agentes que participen en la interacción puedan ser desplegados sobre diferentes plataformas. En este contexto, el término "**plataforma**" [3] se refiere a *la combinación entre el hardware de la máquina y el sistema operativo y bibliotecas auxiliares que deban utilizarse en cada nodo*.

- Decidir el grado de sincronización entre los agentes. Las interacciones entre agentes suelen estar basadas en un mensaje de petición y un mensaje de respuesta [4]. De esta

manera se implanta una interacción sincrónica: el cliente se bloquea tras haber emitido la petición y permanecerá así hasta que llegue la respuesta del servidor. En los sistemas escalables se intenta minimizar la sincronización. Si en lugar de esa interacción sincrónica se utiliza alguna variante que no introduzca bloqueos, deberá diseñarse algún mecanismo para que el servidor comunique su respuesta y el cliente pueda determinar a qué petición previa corresponde.

- Decidir los mecanismos de seguridad a emplear. En algunos casos interesa que los potenciales clientes se registren en el sistema. Posteriormente, antes de iniciar una sesión de uso del servicio tendrán que autenticarse y mantener cierto contexto en cada una de las peticiones efectuadas. Así los servidores podrán asegurar que sólo los usuarios registrados y autorizados están utilizando el servicio. Existen múltiples mecanismos que permiten llevar a cabo este tipo de gestión. Hay que decidir cuál es el que se utiliza.

- Gestionar las situaciones de fallo. No siempre resultará sencillo identificar cuándo uno de los nodos del sistema ha fallado. Por ejemplo, si no recibimos un mensaje de respuesta a una petición previa puede deberse a que: (i) haya problemas en el canal de comunicación utilizado, (ii) el servidor esté saturado sirviendo peticiones previas de otros clientes o (iii) realmente haya fallado. Se necesita algún mecanismo para detectar cuándo ha habido fallos en un sistema distribuido [5]. No resultará trivial implantar ese mecanismo. En cualquier caso, detectar un fallo no resuelve completamente el problema. Posteriormente habrá que reconfigurar aquel servicio donde haya ocurrido ese fallo. En el mejor de los casos estará replicado y bastará con una reconfiguración sencilla. En la Unidad 5 analizaremos este problema en profundidad.

Debería buscarse una solución común para todos estos problemas de bajo nivel. No resulta complicado encontrarla: basta con recurrir a los estándares. Para que un estándar técnico sea publicado se necesita que algunos especialistas de las empresas más importantes de ese sector formen una comisión. Esa comisión evaluará las soluciones existentes en función de su calidad, posibles inconvenientes y esfuerzo necesario para aplicarlas (por lo que respecta a recursos y soporte a utilizar para implantarlas). Una vez se hayan evaluado las diferentes alternativas se prepara una propuesta de especificación que sufre diferentes revisiones hasta que finalmente es admitida y publicada. Este proceso garantiza que cada estándar publicado podrá ser adoptado sin excesivas dificultades y que la solución aportada ha sido verificada tanto desde un punto de vista teórico como práctico.

Utilizar soluciones estandarizadas también conlleva otra ventaja importante: la interoperabilidad. Múltiples empresas utilizarán un mismo estándar para implantar sus soluciones y estos estándares, para el caso particular de la computación distribuida, también especifican el protocolo necesario para que los diferentes componentes que participen puedan intercomunicarse sin problemas.

En el contexto que nos ocupa, las soluciones basadas en estándares utilizadas para resolver los detalles de bajo nivel que se han listado previamente reciben el nombre de "*middleware*". Veamos qué es un *middleware*.

2. Middleware

Uno de los primeros trabajos que presentó el concepto de *middleware*, describiéndolo y comentando las ventajas que aportaría en el diseño, desarrollo y despliegue de aplicaciones distribuidas fue escrito por Philip A. Bernstein [3]. En ese artículo se considera que el

"*middleware*" es aquella capa de *software,* ubicada entre el sistema operativo de cada uno de los nodos del sistema y los componentes de las aplicaciones distribuidas, que cumple estos requisitos:

- Proporciona una interfaz de programación (API) estándar.
- Utiliza protocolos de interacción estándar.
- Garantiza la interoperabilidad de componentes desplegados sobre distintas plataformas.
    - Esta propiedad puede considerarse una consecuencia directa de las dos anteriores: API y protocolos estándar.
    - También implica que el middleware deberá estar implantado sobre múltiples plataformas. Por tanto, no estará ligado a una única arquitectura o a un solo sistema operativo.
- Proporciona un conjunto de servicios de interés general.

Estos requisitos conllevan que el middleware sea una pieza fundamental para garantizar que el sistema distribuido donde se utilice sea un *sistema abierto*. Un sistema abierto es aquel que garantiza la interoperabilidad, recurriendo a estándares para ello.

La arquitectura de un sistema distribuido basado en *middleware* se muestra en la Ilustración 1.

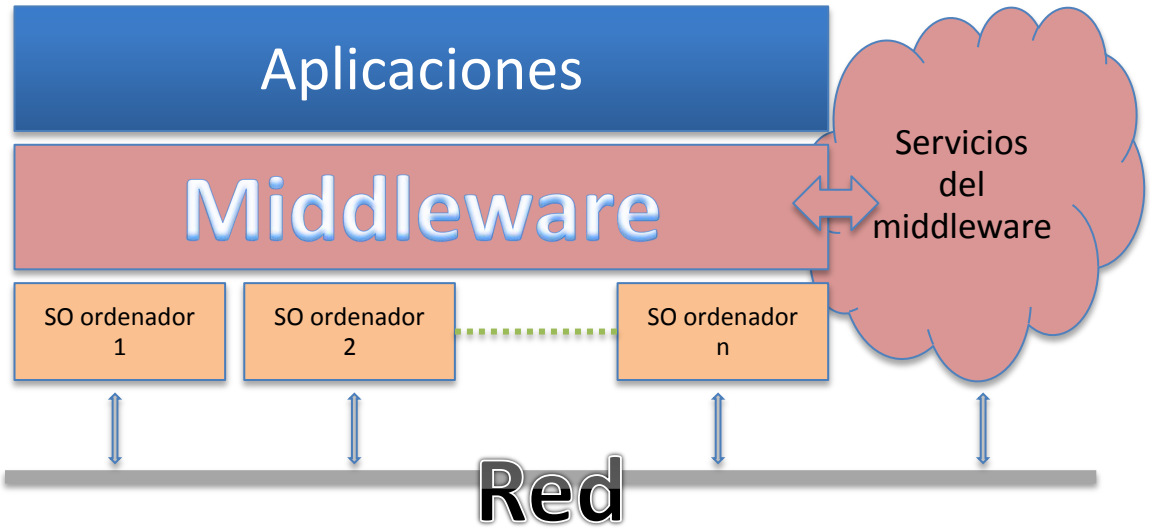

Ilustración 1. Arquitectura basada en middleware.

Según se comenta en [6], los *middleware* introducen varios tipos de transparencia en un sistema distribuido, en función de los servicios que aporten. Por ejemplo, el mecanismo de *llamada a procedimiento remoto* (RPC) [4] o de *invocación a objeto remoto* (ROI) puede proporcionarse en un middleware de comunicaciones y facilitaría transparencia de ubicación. Ofrecer transparencia en un sistema distribuido implica que se está proporcionando cierta interfaz fácilmente utilizable por el programador en la que se ocultan los detalles y la gestión necesarios para proporcionar alguna propiedad interesante para el sistema. Por ejemplo, en la transparencia de ubicación el mecanismo utilizado para proporcionarla facilitará una interfaz de llamada que será idéntica para las entidades locales y remotas. Así el programador no tiene por qué conocer dónde se ubican los objetos que vayan a invocar los distintos componentes de la aplicación distribuida. Es decir, se oculta el uso "proxies" clientes y "esqueletos" servidores

que internamente utilizarán mensajes para hacer llegar las peticiones a la ubicación real de cada objeto invocado, así como retornar las respuestas al agente invocador.

Como consecuencia de su ubicación en la arquitectura y de los servicios que suele ofrecer, un *middleware* facilita las siguientes ventajas:

- Para el programador:
    - Los componentes desarrollados utilizando los servicios proporcionados por el *middleware* tendrán una implantación sencilla. Se utilizarán conceptos claros y bien definidos, que simplificará el proceso de diseño y desarrollo de los componentes. Los problemas más complejos relacionados con el servicio ofrecido ya estarán resueltos por el middleware, empleando una solución eficiente y fiable. De esta manera se reduce la probabilidad de que haya errores en el código implantado.
    - La funcionalidad proporcionada por el middleware está basada en soluciones estándar que han superado un riguroso proceso de evaluación y revisión. Generalmente, su API estará bien documentada y su funcionalidad habrá sido comprobada en un buen número de proyectos anteriores.
    - Se garantiza un mantenimiento sencillo. Si el estándar evoluciona, la nueva API suele ser compatible con la anterior, incorporando nuevas operaciones sin invalidar las previamente existentes. De esta manera no se exige que haya grandes cambios en los componentes de una aplicación si se decidiera migrarla a la nueva versión de algún estándar.
- Para el administrador:
    - Como los *middleware* proporcionan funcionalidad e interfaces estándar, el administrador encontrará varios productos desarrollados por diferentes empresas proveedoras para seleccionar el *middleware* a utilizar. Cada proveedor se esforzará en generar un producto lo más eficiente y fácilmente administrable posible. Por ello, las tareas de instalación, configuración y actualización de un *middleware* no deberían ser complejas.
    - Por estar basados en estándares, los *middleware* deben ser interoperables. Resultará sencillo combinar diferentes *middleware* proporcionados por proveedores distintos para construir una buena base sobre la que desarrollar o desplegar aplicaciones distribuidas eficientes. Las aplicaciones distribuidas de gran escala podrán ser desplegadas en múltiples centros de datos y no será raro que en cada centro sus administradores hayan seleccionado un proveedor distinto para una misma clase de middleware.
    - En otros casos puede que interese utilizar múltiples productos *middleware* especializados en servicios diferentes: invocación remota, soporte transaccional, seguridad... No tiene por qué haber restricciones para instalar, configurar y utilizar esas combinaciones.

3. Sistemas de mensajería

Para que un sistema pueda ser escalable tendrá que diseñarse de manera que jamás se bloquee su actividad. Para ello todas las interacciones deberían ser asincrónicas y la toma de decisiones debería ser inmediata (tomando únicamente la información local disponible para

este fin). Los middleware basados en RPC o en ROI no cumplen estas restricciones. En ambos casos se están realizando interacciones basadas en un patrón petición-respuesta que bloquea al proceso que actúe como cliente mientras no se reciba la respuesta. Sería conveniente encontrar otro patrón de interacción.

Los sistemas de mensajería fueron diseñados con ese objetivo. El emisor de un mensaje no se suspende esperando una respuesta o una confirmación por parte del receptor. Por tanto, en estos sistemas se utiliza *comunicación asincrónica* [6]. Por otra parte, si el canal de comunicación puede almacenar los mensajes tendremos *comunicación persistente*. En ese caso no será necesario que el proceso receptor exista cuando el emisor envíe el mensaje. El canal guardará el mensaje hasta que el receptor esté preparado para recibirlo.

Otra ventaja de los *middleware* de mensajería (también conocidos como "*Message-Oriented Middleware*" o MOM) es que no necesitan la imagen de espacio compartido entre procesos ofrecida en los sistemas de objetos distribuidos. Cuando se emplea mensajería cada proceso es independiente de los demás y no comparte ningún recurso con el resto de procesos. Esto elimina algunas de las restricciones que habíamos visto en la sección anterior para desarrollar sistemas escalables. Al no haber un espacio compartido se evitarán los problemas que podría conllevar la ejecución de múltiples actividades concurrentes: no hay recursos compartidos que exijan complejos protocolos de protección. En los sistemas basados en mensajería, la concurrencia no supondrá ningún problema pues no se necesitarán mecanismos adicionales de sincronización.

La misión del *middleware* de mensajería es transmitir piezas de información entre los diferentes agentes del sistema. Para ello suelen emplearse colas para implantar cada canal de comunicación y se garantiza la entrega atómica del mensaje (es decir, se entrega o no se entrega). Suele darse cierta libertad para fijar el tamaño de los mensajes: dependerá de las necesidades de la aplicación.

Existen estándares para este tipo de middleware. Un ejemplo es AMQP (*Advanced Message Queueing Protocol*) [7], estandarizado por OASIS en octubre de 2012 y por ISO/IEC en abril de 2014. Sin embargo, aunque este estándar cumple todos los requisitos para generar sistemas middleware que lo implanten, su interfaz es excesivamente amplia. Su configuración también resulta compleja. Su descripción no sería abordable en esta asignatura. Tampoco resulta sencillo para un programador aprender toda su funcionalidad y considerar todos los detalles de su interfaz a la hora de desarrollar una aplicación que lo utilice. Todo esto explica que AMQP no haya tenido una gran difusión.

Por el contrario, otros sistemas de mensajería como JMS (especificación estándar JSR-914) o ØMQ (http://zeromq.org/) [8] se han preocupado por proporcionar una interfaz sencilla e intuitiva. ØMQ, además, ofrece un rendimiento excelente. Por ello, ØMQ será el sistema de mensajería que se utilizará en las prácticas de laboratorio.

Como ya se ha comentado anteriormente los sistemas de mensajería proporcionan un sistema asincrónico de comunicación. Eso facilitará el desarrollo de aplicaciones distribuidas escalables al eliminar una fuente potencial de bloqueos. Existen dos tipos principales de sistemas de mensajería en función de su grado de persistencia:

- En los sistemas de mensajería no persistentes tanto el agente emisor como el agente receptor deben existir y estar pendientes del canal de comunicación para que esa comunicación sea posible. El protocolo UDP sería un ejemplo de comunicación de este tipo, aunque los sistemas de mensajería utilizan sus propios mecanismos y protocolos a nivel de aplicación.

- Por su parte, los sistemas de mensajería persistentes utilizarán canales (o conectores) capaces de mantener los mensajes. De esa manera no es necesario que haya ningún receptor activo cuando se envíe un mensaje: el canal lo mantendrá hasta que el receptor esté listo para recibirlo. En esta variante se suele implantar esa capacidad del canal mediante una cola de mensajes. Podemos distinguir dos tipos de alternativas entre los sistemas persistentes:

    - Sistemas basados en gestor (*broker-based*): Existirá al menos un proceso gestor que mantendrá las colas, guardando los mensajes en almacenamiento secundario en algunos casos. Esta variante proporciona garantías fuertes para entregar (no se perderán mensajes) y persistir los mensajes, sacrificando el rendimiento (se debe invertir cierto tiempo para escribir el contenido de los mensajes en disco y recuperarlo desde allí cuando sea necesario). AMQP y JMS son ejemplos de sistema de este tipo.

    - Sistemas sin gestor (*brokerless*): Las colas de mensajes no son mantenidas por ningún proceso externo ni tampoco en almacenamiento secundario. Esto obliga a que la persistencia sea débil (en memoria principal) y que esté parcialmente manejada por emisores y receptores. Presenta la ventaja de ofrecer un excelente rendimiento a costa de reducir las garantías de persistencia. Los mensajes no se pueden mantener a perpetuidad: si los receptores no están disponibles en un plazo relativamente breve, los mensajes se descartarán. ØMQ es un ejemplo de sistema de este tipo.

4. ØMQ

4.1. Introducción

ØMQ (ZeroMQ) es un middleware de comunicaciones. Puede decirse que es “simple” en el sentido de ser fácil de utilizar y configurar. Su configuración solo necesita utilizar URL para especificar los “endpoints” de sus operaciones bind() y connect(). Su uso es también sencillo, pues su API es similar a la tradicional para sockets en los sistemas BSD.

Aparte de su facilidad de uso y configuración, sus objetivos principales son:

- Amplia disponibilidad: ØMQ se ha implantado como una biblioteca escrita en C++. Puede usarse en muchos lenguajes de programación y se ha migrado a varios sistemas operativos (incluyendo Windows, Linux y MacOS).
- Soporte para múltiples patrones de comunicación: ØMQ soporta varios tipos de sockets. Con ellos, el programador puede utilizar diferentes patrones de comunicación entre procesos, sin que ello suponga ningún esfuerzo.
- Buen rendimiento: Casi todos los patrones de comunicación de ØMQ son asincrónicos y esto mejora el rendimiento y escalabilidad de los servicios implantados con ØMQ. La asincronía implica que cada proceso pueda continuar su ejecución tan pronto como invoque una operación sobre un socket, pues no debe esperar ningún reconocimiento o respuesta por parte de otros procesos. Siempre continúa inmediatamente.
- Adaptabilidad a diferentes tipos de despliegue: Los sockets ØMQ pueden usarse para comunicar procesos ubicados en un mismo ordenador, en ordenadores diferentes o incluso para comunicar los hilos de un mismo proceso. Un mismo conjunto de programas podrá desplegarse en todos esos escenarios sin requerir apenas cambios. El pro-

gramador solo debe elegir un URL adecuado para realizar el bind() o connect() de cada socket, dependiendo del escenario.

ØMQ sigue un modelo de comunicación débilmente persistente. Un sistema de comunicaciones basado en mensajes es persistente cuando mantiene los mensajes en tránsito hasta que el proceso receptor esté listo para aceptarlos. Habitualmente, esa persistencia se implanta manteniendo los mensajes en procesos intermediarios que gestionan la comunicación (esto es, en los "brokers"). Sin embargo, ØMQ no utiliza "brokers". Por ello, la persistencia débil está soportada por cada instancia de la biblioteca ØMQ, en cada proceso. La biblioteca mantiene los mensajes en colas internas (en memoria principal) hasta que puedan enviarse al proceso receptor (en el proceso emisor) o puedan entregarse (en el proceso receptor).

La tecnología de comunicación que realmente se utiliza en un middleware de mensajería depende principalmente del protocolo de transporte. ØMQ soporta múltiples transportes (por ejemplo, "pgm" para realizar difusiones, "tcp" para comunicaciones punto a punto, "inproc" para intercomunicar a los hilos de un mismo proceso...) pero algunos de ellos no tienen sentido en NodeJS (por ejemplo, "inproc", pues JavaScript no soporta múltiples hilos en un mismo proceso). Por tanto, en NodeJS solo se usarán dos de esos transportes: (1) "tcp" para comunicación entre procesos ubicados en uno o varios ordenadores y (2) "ipc" para algunos casos en que todos los procesos estén ubicados en un mismo ordenador. El transporte "ipc" está basado en sockets UNIX soportados por ficheros. El transporte a utilizar debe especificarse en el primer componente del URL.

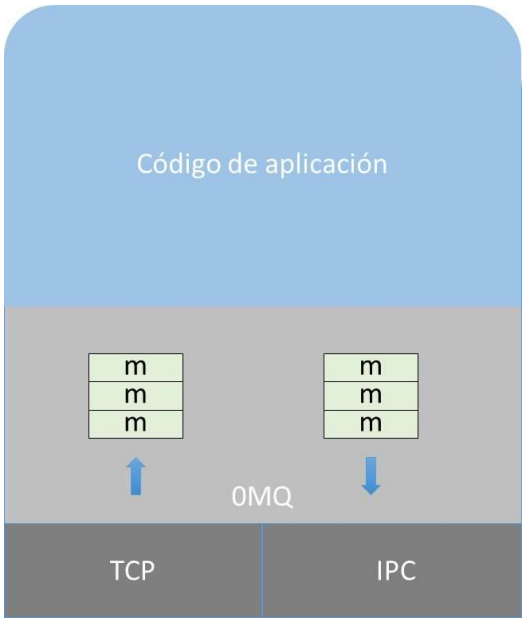

Ilustración 2. Arquitectura de un proceso ØMQ.

La Ilustración 2 muestra la arquitectura de un proceso ØMQ. Su parte superior representa el código y datos de la aplicación. La zona central representa la biblioteca ØMQ. En los lenguajes interpretados, como JavaScript, la biblioteca se importa (por ejemplo, utilizando la sentencia "require()" en NodeJS). Sin embargo, en los lenguajes compilados (como C o C++), la biblioteca se enlaza con el resto del programa.

Hemos mencionado previamente que ØMQ proporciona persistencia débil. Para ello, necesita mantener los mensajes en colas. La Ilustración 2 muestra las colas de recepción (a la izquierda) y de envío (a la derecha). En la parte inferior de la figura podemos ver los módulos que implantan los transportes correspondientes (TCP o IPC para el caso de un proceso NodeJS).

4.2. Mensajes

Los mensajes son los elementos que los procesos intercambian entre sí para comunicarse en un middleware como ØMQ. ØMQ ofrece estas características en su gestión de los mensajes:

- Los mensajes pueden ser "multi-parte". Esto significa que los mensajes pueden estructurarse en una secuencia de partes o segmentos. Para ello, el proceso emisor debe organizar el mensaje como un vector, con un segmento en cada una de sus componentes. El vector debe pasarse como primer argumento en las llamadas a la operación send(), como muestra este ejemplo:

```
mySocket.send(["This is", "an example", "of message"])
```

Este mensaje tiene tres segmentos. Cada segmento es una cadena de caracteres. Cada segmento será recibido en un parámetro distinto en el proceso receptor, como muestra el siguiente ejemplo, que complementa al que acabamos de ver:

```
otherSocket.on("message", function (seg1, seg2, seg3) {
  console.log("Segment 1 is: ", seg1+'')
  console.log("Segment 2 is: ", seg2+'')
  console.log("Segment 3 is: ", seg3+'')
})
```

Obsérvese que los mensajes se entregan cuando un evento "message" ocurre en el proceso receptor. Para ello, debemos escribir un "callback" que gestione esa recepción. En este ejemplo, el "callback" debe tener tantos parámetros como segmentos tenía el mensaje enviado. El código del "callback" muestra el contenido de cada segmento.

Podríamos haber hecho lo mismo utilizando este código alternativo:

```
otherSocket.on("message", function (...segs) {
  for (let i in segs)
    console.log("Segment %d is: %s ", i, segs[i]+'')
})
```

En este segundo caso, al emplear los puntos suspensivos conseguimos que todos los argumentos se reciban en un vector "segs". La salida mostrada en estos dos últimos ejemplos es idéntica. En ambos casos estamos convirtiendo cada segmento en una cadena (concatenando para ello la cadena vacía tras cada segmento), pues los segmentos se pasan por omisión como objetos de tipo Buffer. Si visualizamos un objeto Buffer (por ejemplo, utilizando console.log, como en estos ejemplos), NodeJS muestra una secuencia de octetos en hexadecimal, difícilmente legible.

Otra alternativa consiste en dejar la lista de parámetros del "callback" vacía. En ese caso, habrá que utilizar el pseudovector "arguments". Podemos convertir "arguments" en un vector convencional utilizando la operación Array.from(), como se muestra seguidamente:

```
otherSocket.on("message", function () {
  let segments = Array.from(arguments)
  ...
})
```

De entre las tres alternativas que acabamos de mencionar, la primera se recomienda en aquellos casos en que todos los mensajes a recibir tienen un número concreto y conocido de segmentos. Por su parte, la segunda resulta especialmente útil cuando los mensajes que deba recibir ese agente puedan ser de diferentes tipos y cada tipo utilice un número distinto de segmentos. La tercera se utilizó principalmente en versiones anteriores del lenguaje JavaScript, para gestionar mensajes de diferentes tipos en un mismo agente. En aquellas versiones todavía no era posible utilizar parámetros "...arg", por lo que se necesitaba llamar a Array.from(), u otros métodos similares, para convertir esos argumentos recibidos en un vector.

- Los mensajes se entregan atómicamente. A pesar de poder estructurar los mensajes en múltiples segmentos, su recepción siempre se realiza en una única acción. No podemos tener una recepción por cada segmento. Los ejemplos mostrados arriba ya ilustran esta situación. Por tanto, o se entregan todas las partes de un mensaje a la vez o, si faltara alguna de ellas, no se entrega nada todavía.
- Las operaciones de recepción y envío son asincrónicas. El proceso emisor deja su mensaje en la cola de salida, sin suspender su ejecución. Por ello, puede continuar de inmediato con la instrucción que siga a send().

  La recepción de un mensaje está asociada al evento "message" del socket receptor. Por tanto, tampoco requiere que el proceso receptor se suspenda. Cuando el mensaje llegue, ese evento "message" será generado. Eso implica que el contexto del "callback" asociado al evento, incluyendo los argumentos que representan el mensaje recibido, será ubicado al final de la cola de turnos de ese proceso receptor. Una vez el receptor finalice su turno actual (y otros que pudiera haber en la cola de turnos antes de la llegada del mensaje), procederá con la gestión del mensaje entregado.

- Gestión automática de las conexiones. Las conexiones entre procesos se establecen cuando uno de ellos utiliza bind() y el otro connect() sobre un mismo URL. La gestión de la conexión la realiza la biblioteca ØMQ y resulta transparente para el programador. Esa transparencia implica lo siguiente:
  - 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.
  - Cuando se utilicen sockets asincrónicos (es decir, todos excepto REQ y REP), si dos procesos A y B ya están conectados y uno de ellos falla, el otro no necesita hacer nada para reconectarse con aquel que reemplace al que ha fallado. Así, si B fallase, podríamos iniciar un nuevo proceso C que sustituya a B. C realizaría el bind() o connect() correspondiente, pero A no necesitaría repetir ninguna de esas operaciones. Su conexión se restablece automáticamente.

El contenido de los mensajes resulta irrelevante para la biblioteca ØMQ. ØMQ se limita a propagar los mensajes entre los procesos, asumiendo que cada segmento será un Buffer. Ya

hemos visto cómo podemos convertir un Buffer en una cadena (concatenando la cadena vacía). Por tanto, el programador es el responsable de dar estructura a los mensajes de la manera más adecuada. Hay múltiples alternativas para ello: (1) como una sola cadena, (2) como objetos en formato JSON, (3) como mensajes multi-segmento... La primera alternativa es la más sencilla para mensajes cortos, mientras que tanto la segunda como la tercera serán adecuadas para mensajes compuestos.

4.3. API

Veamos cuáles son las principales características de los sockets ØMQ (cómo se crean, operaciones de envío y recepción, transportes a utilizar, tipos de sockets...) para poder explicar posteriormente sus patrones de comunicación.

4.3.1. Sockets

Los sockets deben crearse utilizando la operación socket() del módulo NodeJS "zmq". Esa operación necesita un argumento: el tipo de socket a crear. Por tanto, un fragmento de código para crear un socket de tipo Req podría ser el siguiente:

```
const zmq = require('zmq')
let so = zmq.socket('req')
```

En ØMQ hay once tipos de socket. Son los siguientes:

- Req: Socket bidireccional sincrónico. Necesario para enviar peticiones desde un cliente a un servidor y recibir las respuestas correspondientes.
- Rep: Socket bidireccional sincrónico. Necesario para recibir las peticiones enviadas por los clientes utilizando sockets Req y enviar las respuestas asociadas a esas peticiones.
- Push: Socket unidireccional asincrónico. Solo puede enviar mensajes.
- Pull: Socket unidireccional asincrónico. Solo puede recibir mensajes.
- Pub: Socket unidireccional asincrónico. Publicador. Difunde mensajes a múltiples sockets Sub.
- Sub: Socket unidireccional asincrónico. Suscriptor. Recibe los mensajes difundidos por sockets Pub.
- Dealer: Socket bidireccional asincrónico. Puede considerarse una variante asincrónica del socket Req.
- Router: Socket bidireccional asincrónico. Puede considerarse una variante asincrónica del socket Rep.
- Xpub: Cuando haya múltiples publicadores y múltiples suscriptores interesados en los mensajes difundidos por esos publicadores, se puede implantar un gestor estático al que todos esos procesos se conecten. En ese caso, el gestor utilizará un socket Xpub para redifundir los mensajes publicados hacia todos los suscriptores conectados a este gestor. No utilizaremos sockets Xpub en esta asignatura.
- Xsub: Cuando haya múltiples publicadores y múltiples suscriptores interesados en los mensajes difundidos por esos publicadores, se puede implantar un gestor estático al que todos esos procesos se conecten. En ese caso, el gestor utilizará un socket Xsub para recibir todos los mensajes publicados y redifundirlos hacia los procesos suscritos. No utilizaremos sockets Xpub en esta asignatura.

- Pair: Estos sockets se usan para conectar hilos en un mismo proceso. Como JavaScript es un lenguaje de programación que no soporta múltiples hilos de ejecución, no usaremos este tipo de socket en esta asignatura.

Una vez se haya creado un socket, el programador debe especificar su "endpoint", es decir, el URL del que escuchará o al que se conectará. Hay dos operaciones para ello: bind() y connect(). La operación bind() es asincrónica, con un "callback" para gestionar posibles errores, pero también existe una variante sincrónica: bindSync(). Como los procesos no pueden utilizar un socket hasta que no haya finalizado su bind() correspondiente, resulta recomendable utilizar la variante sincrónica. La operación connect() solo tiene una variante que retorna el control inmediatamente, pero no facilita ningún "callback" para averiguar el resultado de ese intento de conexión. Como ya se ha dicho, las conexiones son gestionadas de manera transparente por la biblioteca. Así, se puede enviar mensajes por esa conexión tan pronto como connect() devuelva el control. En algún momento la conexión será establecida y entonces todos aquellos mensajes que se hayan enviado por ese socket serán transmitidos y entregados a su destino. El programador debe ser cuidadoso a la hora de pasar el argumento (URL) a connect(). Si hubiera algún error, el proceso no podría reaccionar.

Ya se ha comentado que la biblioteca ØMQ realiza automáticamente la gestión de las conexiones. Por tanto, el programador no debe preocuparse por el orden en que se ejecutarán bindSync() y connect(). Sin embargo, hay otros aspectos de esas operaciones que merece la pena destacar:

- Conviene emplear bindSync() en aquel agente cuyo "endpoint" deba ser conocido por todos los demás agentes. Por ejemplo, cuando utilicemos un patrón de comunicación cliente-servidor, el servidor debería ser conocido por todos los clientes. En ese caso, el servidor realizará un bindSync() para su socket Rep y todos los sockets Req de los clientes utilizarán connect().
- Cuando tengamos una relación 1:N entre los roles interpretados por los procesos, el rol con cardinalidad 1 utilizará bindSync() mientras que el rol con cardinalidad múltiple utilizará connect(). En general, esta segunda recomendación guarda relación con lo dicho en el párrafo anterior (es decir, normalmente el rol con cardinalidad 1 suele tener un "endpoint" estático y conocido por los demás procesos).

Cuando un proceso ya haya enviado o recibido todos los mensajes que pretendía gestionar por un socket S, podrá finalizar esa sesión de comunicaciones utilizando la operación close() sobre S. Tras ello, ese socket ya no podrá procesar ningún otro mensaje.

Como ya se ha mencionado al explicar la Ilustración 2, los sockets ØMQ utilizan colas de recepción y envío para facilitar persistencia débil. La cantidad y tipo de colas manejadas por cada socket dependen de su tipo. Así, un socket tiene ambos tipos de cola cuando permita comunicación bidireccional (como en los tipos Req, Rep, Dealer y Router), pero solo un tipo de cola cuando solo soporte comunicación unidireccional. Por ejemplo, los sockets Pub y Push solo tienen colas de envío, mientras que Sub y Pull las tienen de recepción.

Entre los sockets bidireccionales, los Router son especiales pues pueden gestionar múltiples conexiones con otros procesos de manera que se pueda seleccionar a qué conexión específica

se desea enviar cada mensaje. Por ello, necesita un par de colas de envío/recepción por cada conexión establecida, como veremos en algunos ejemplos más tarde. Todos los demás sockets bidireccionales solo tienen una cola de envío y una cola de recepción, compartidas entre todas las conexiones existentes. Por ello, cuando los mensajes se dejan en la cola de envío, se entregarán en sus correspondientes procesos receptores siguiendo una política de turno circular (o Round-Robin). Por otra parte, los mensajes que se vayan recibiendo se dejan en la cola de recepción en orden de llegada (es decir, con una política FIFO).

Ya hemos mencionado que en NodeJS solo podremos utilizar dos transportes en ØMQ: TCP e IPC. Describamos ahora cómo se debe especificar un URL en cada transporte.

En ambos casos, el URL tendrá al menos dos partes. La primera contiene el nombre del transporte seguida por el carácter dos puntos (":") y dos barras, generando así el prefijo "tcp://" o "ipc://". La segunda parte es la dirección, que depende del protocolo:

- En TCP esa dirección tiene dos elementos, separados por un carácter dos puntos: (a) la dirección IP y (b) el número de puerto.
- En IPC, la dirección es el nombre de ruta absoluto del fichero de tipo socket UNIX. Ese fichero debe tener permiso de lectura y escritura. Por ejemplo:

```
so.bindSync("ipc:///tmp/mySocket")
```

Por lo que respecta al transporte TCP, hay varias formas de especificar su dirección:

- De manera general, se utilizará una dirección IP convencional, como esta:

```
so.bindSync("tcp://192.168.1.3:8000")
```

- Podemos utilizar el asterisco en caso de usar la operación bind() o bindSync() (pero no para connect()), indicando que se van a utilizar todas las direcciones locales:

```
so.bindSync("tcp://*:8000")
```

- Por último, podemos hacer referencia a todas las direcciones asociadas a una interfaz de red concreta. De nuevo, esto es solo útil en las operaciones bind() y bindSync(). Por ejemplo, esta sentencia se refiere a todas las direcciones de la primera tarjeta de red Ethernet del equipo:

```
so.bindSync("tcp://eth0:8000");
```

4.3.2. Patrones de comunicación básicos

Hay tres patrones de comunicación básicos en ØMQ:

- REQ/REP: Patrón petición/respuesta sincrónico bidireccional. Se utilizará cuando uno o más clientes (con sockets Req) interactúen con un servidor que use un socket Rep.
- PUSH/PULL: Patrón asincrónico unidireccional. Los mensajes se envían con un socket Push y son recibidos por otro proceso con un socket Pull.
- PUB/SUB: Patrón de publicación/suscripción asincrónico unidireccional. Un proceso publicador utiliza un socket Pub para difundir mensajes a uno o más procesos suscriptores (con sockets Sub).

Esos patrones se explican en profundidad en los próximos apartados.

Patrón de comunicaciones REQ/REP

Este patrón permite que múltiples procesos clientes, utilizando sockets Req, interactúen con un proceso servidor que utilice un socket Rep. Los nombres de estos sockets corresponden a la abreviatura en inglés de los términos "petición" (Request) y "respuesta" (Reply).

Generalmente, los sockets ØMQ son asincrónicos. Sin embargo, Req y Rep no lo son. Veamos por qué. Supongamos tres procesos clientes A, B y C que envían mensajes a un servidor S. De momento solo han enviado un mensaje cada uno: mA, mB y mC, respectivamente. Esos tres mensajes se han entregado asincrónicamente a S, creando sus correspondientes turnos de ejecución en S. Más pronto o más tarde, S iniciará esos turnos. Sin embargo, cuando cada una de esas peticiones sea procesada, podrá solicitar otras operaciones asincrónicas en S. Por ejemplo, leer el contenido de un fichero local, una o más veces en cada petición. Si eso ocurriera, se crearían más turnos de ejecución en S para proseguir con la ejecución de cada petición. Al final, no se podría predecir en qué orden finalizarían esas peticiones.

Hemos dicho que cada socket ØMQ tiene, por omisión, una cola de envío y otra de recepción. Sus políticas habituales de entrega de mensajes siguen un turno circular. Eso implica que el socket Rep utilizado en S estaría gestionando al menos tres conexiones: una por cada cliente conectado (A, B y C). Si se siguiera esa política circular, la primera respuesta debería enviarse a A, la segunda a B, la tercera a C y así sucesivamente. Eso no tendría por qué coincidir con el orden de llegada de sus peticiones, ni tampoco con su orden de finalización. Por tanto, esas respuestas podrían enviarse a un proceso equivocado, por culpa de esa gestión asincrónica.

Se necesita alguna solución para este problema. En lugar de esa gestión general (que permitiría cierto grado de concurrencia en la gestión de las peticiones, a pesar de solo tener un hilo en el proceso servidor), el patrón REQ/REP utiliza una gestión sincrónica. Para ello se establecen ciertos criterios para gestionar los mensajes en cada socket:

- Regla de envío: Cuando se envíe un mensaje "m1" por un socket Req R1, todo intento posterior de envío de un nuevo mensaje "m2" deja "m2" en un estado bloqueado. La gestión de "m2" solo se reanudará cuando una respuesta para "m1" sea recibida en R1. Esto evita que un mismo cliente pueda tener más de una petición pendiente, incluso cuando se conecte a múltiples servidores.
- Regla de recepción: Cuando una petición "m1" es recibida en un socket Rep R2, todo intento posterior de recepción de un nuevo mensaje "m2" deja "m2" en un estado bloqueado. La gestión de "m2" solo se reanudará cuando la respuesta para "m1" (y para todos los demás mensajes que puedan preceder a "m2" en la cola de recepción) sea enviada por R2. Esto evita que un mismo servidor pueda tener más de una petición en servicio simultáneamente, incluso cuando esas peticiones hayan sido enviadas por diferentes clientes.

Además, los sockets Rep no pueden enviar una respuesta de antemano (pues ese intento de envío es bloqueado) mientras no se reciba su petición asociada.

La regla de recepción resuelve directamente el problema presentado anteriormente. Solo puede haber una respuesta pendiente en cada socket Rep. Esa respuesta solo puede corresponder a la petición actualmente en servicio. No hay ambigüedad posible. Solo se podrá

utilizar una conexión en cada momento. En nuestro ejemplo, si las peticiones hubieran llegado en orden mA, mB y mC, podríamos asegurar que se atenderá primero mA y se le dará respuesta. Tras ello se atenderá mB y se responderá esa segunda petición. Finalmente, se recibirá mC y se contestará a C. El servicio de esas tres peticiones se realiza de manera secuencial, sin ningún tipo de concurrencia.

Ese comportamiento sincrónico puede tener consecuencias imprevistas en caso de fallo. Por ejemplo, consideremos la ejecución de la Ilustración 3. En esa ejecución, un cliente interactúa con un servicio replicado que tiene dos réplicas. En este caso, el sistema no facilita transparencia de replicación. Por tanto, cada cliente conoce las direcciones de todas las réplicas. En la Ilustración 3, la segunda réplica falla tras haber recibido una petición, justo antes de responderla. Por ello, el socket Req del cliente está esperando una respuesta, pero esa respuesta no llegará jamás. Los intentos de envío de sucesivas peticiones quedarán bloqueados en su cola de envío, debido a la regla de envío descrita previamente.

Para solucionar esa situación, el cliente debe fijar un "timeout" para cada respuesta. Si ese plazo se agota, el socket Req debe ser: (1) cerrado para descartar cualquier intento posterior de envío de peticiones, (2) creado de nuevo y (3) reconectado por cada réplica servidora. Esa reconexión se realiza implícitamente en cada réplica servidora que permanezca activa si se reutiliza el mismo URL. Tras esos tres pasos, el cliente podrá reanudar su actividad, reenviando cada una de las peticiones que habían quedado bloqueadas debido al fallo anterior.

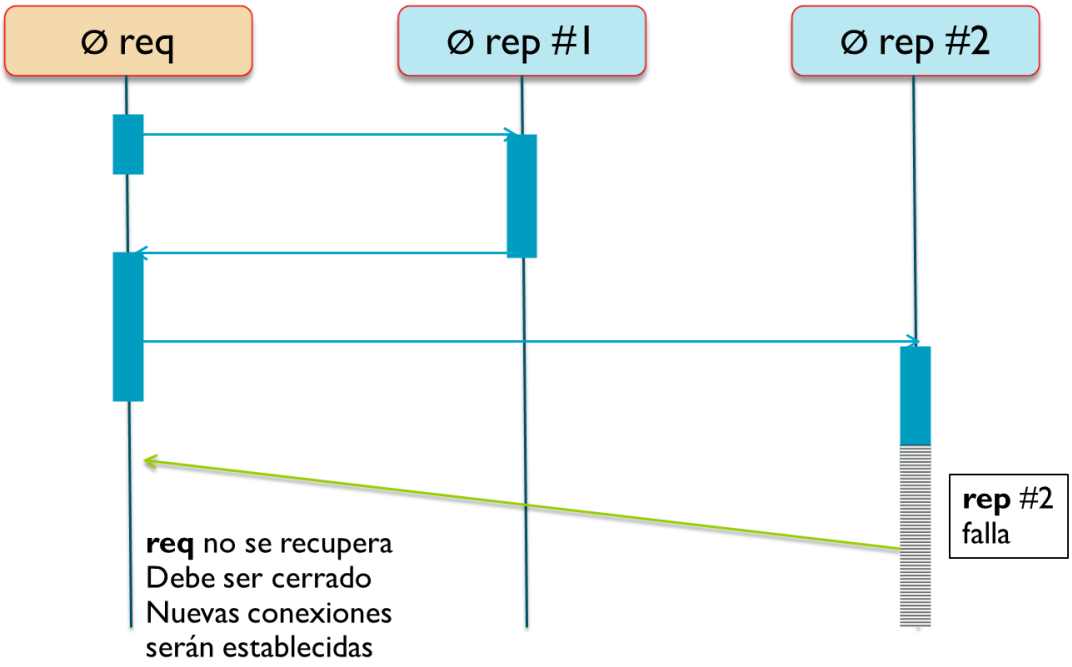

Ilustración 3. Fallo de un servidor en el patrón REQ/REP.

Los sockets Req añaden automáticamente un primer segmento vacío a cada mensaje enviado. Ese segmento se llama "delimitador". Los delimitadores son eliminados automáticamente por los sockets Rep al entregar los mensajes al proceso servidor. Por tanto, el uso de delimitadores es transparente para el programador de aplicaciones. Cuando el proceso servidor responde al

cliente, el socket Rep vuelve a añadir el delimitador a esa respuesta. Por último, el socket Req también eliminará el delimitador antes de entregar la respuesta al cliente.

Esta gestión de los delimitadores se necesita para gestionar "envoltorios" (equivalentes a un sobre en el que meter el mensaje) cuando un socket Rep interactúa con otros tipos de sockets (diferentes al Req). En esos casos, el "envoltorio" es la secuencia de segmentos que precede e incluye al delimitador. Toda esa secuencia la mantiene el socket Rep cuando entrega el mensaje al servidor, para incluirla de nuevo como prefijo en la respuesta emitida por el servidor.

Patrón de comunicaciones PUSH/PULL

Los dos sockets que componen este patrón son asincrónicos y unidireccionales. Por tanto, el patrón resultante es asincrónico y la información solo puede enviarse desde el socket Push al Pull.

En el caso más sencillo, hay un solo proceso emisor con un socket Push y un solo receptor con un socket Pull. Todos los mensajes enviados son recibidos por el mismo proceso.

Como ya se ha mencionado, un mismo socket puede conectarse con varios sockets. En ese caso, se podría definir un canal de múltiples agentes de procesamiento como el mostrado en la Ilustración 4. En él, tenemos un agente A inicial con un socket Push que pasa su información a dos agentes B y C en la segunda etapa de procesamiento. Ambos tienen sockets Pull y Push. Utilizarán sus Push para propagar el resultado de su proceso a una tercera etapa de procesamiento, en la que el agente D con un socket Pull recibe todos los resultados.

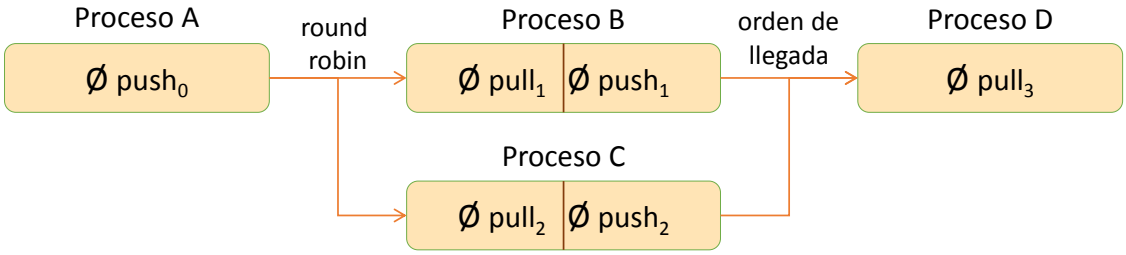

Ilustración 4. Canal de proceso ("pipeline").

Obsérvese que el único socket Push del agente A está conectado con dos sockets Pull, pertenecientes a los agentes B y C. Como ya se ha explicado, los mensajes enviados son entregados en orden circular a B y C, recibiendo cada uno de ellos la mitad de esos mensajes (distribuyendo así su carga de procesamiento de manera equitativa). Posteriormente, los mensajes que envíen ambos procesos hacia el agente D se guardan temporalmente, si es necesario, en orden FIFO en la cola de recepción de su socket Pull.

Patrón de comunicaciones PUB/SUB

Al igual que en el patrón anterior, este patrón utiliza sockets asincrónicos unidireccionales. Por tanto, los mensajes siempre se propagarán del socket Pub al Sub. Sin embargo, hay una diferencia con el patrón anterior: si el socket Pub se conecta a varios sockets Sub, entonces cada mensaje se propaga a todos los sockets Sub. Así, en vez de un turno circular de entrega

sobre todos los sockets conectados, este patrón sigue una política en la que se entrega a todos los destinatarios disponibles. Con ello, este patrón implanta difusiones.

Hay dos características de este patrón que conviene explicar:

- Los mensajes solo se entregan a los suscriptores que estén actualmente disponibles. Si hubiera algún suscriptor temporalmente inalcanzable o que no hubiera sido iniciado todavía, los mensajes dirigidos hacia ellos se perderían. Conviene recordar que en otros tipos de socket, las conexiones se gestionan automáticamente y que los mensajes en tránsito son mantenidos en las colas de envío para entregarlos cuando sea posible. Ese comportamiento no se sigue en el patrón PUB/SUB.
- Los procesos suscriptores deben especificar qué mensajes les interesa recibir. Para ello, el programador debe especificar, mediante el método subscribe(), los prefijos de interés. A partir de ese momento, el socket Sub solo entregará aquellos mensajes publicados que contengan alguno de esos prefijos, descartando los demás mensajes. Si un suscriptor está interesado en todos los mensajes, debe especificar una cadena vacía como argumento en su correspondiente llamada a subscribe().
  Si no se llega a utilizar subscribe(), entonces todos los mensajes publicados son descartados por ese proceso suscriptor. Eso significaría que no está interesado en ningún prefijo. Por ello, no se entrega nada a ese socket Sub.

4.4. Tipos de socket avanzados

ØMQ facilita dos tipos avanzados de sockets: Dealer y Router. Pueden considerarse variantes asincrónicas para los sockets Req (Dealer) y Rep (Router). La asincronía introduce algunos inconvenientes serios en la gestión de las interacciones cliente/servidor. Si hay múltiples clientes interactuando con un único servidor y las peticiones y respuestas son asincrónicas, el servidor debe recordar qué conexión cliente está asociada a cada petición. En el patrón REQ/REP, el socket usado por los servidores es de tipo Rep. Por tanto, la gestión de conexiones debe introducirse en el socket avanzado que reemplace al Rep. Ese socket es el Router. Por ello, los sockets Router deben admitir que se especifique (o gestionar directamente) a qué conexión cliente debe dirigirse cada respuesta.

El resto de este apartado describe en primer lugar los sockets Dealer, seguidos por los Router y un ejemplo de uso de ambos tipos de socket.

4.4.1. Sockets Dealer

Estos sockets son bidireccionales y asincrónicos. Son utilizados por los clientes cuando implantan interacciones cliente/servidor asincrónicas. Por tanto, los procesos clientes envían peticiones y reciben respuestas mediante este tipo de socket.

La diferencia principal entre sockets Req y Dealer es su nivel de sincronía. Los sockets Req son sincrónicos (bloquean envíos de más peticiones mientras no se reciba la contestación para la petición actual) y los sockets Dealer son asincrónicos (se pueden enviar tantas peticiones como se quiera, sin incurrir en ningún bloqueo, independientemente de cuántas respuestas se hayan recibido). Pero también existe una segunda diferencia: los sockets Req introducen automáticamente un delimitador que precede a los segmentos normales de la petición (y

también lo esperan y eliminan en las respuestas) mientras que los sockets Dealer no introducen ni gestionan delimitadores.

La segunda diferencia introduce problemas cuando un socket Dealer debe interactuar con un socket Rep. El socket Rep asume que las peticiones entrantes incluyen el delimitador para separar el "envoltorio" del resto del mensaje. Si no hay delimitador, no entregan el mensaje, que es rechazado. Por tanto, si se usa un Dealer, el programador debe añadir manualmente el delimitador necesario.

Veamos un ejemplo de programas cliente y servidor que utilizan sockets Rep en el servidor y Dealer en el cliente. Se han implantado dos réplicas del servidor que comparten un mismo código, pero que escuchan en diferentes puertos, como sigue:

```
// Server 1
const zmq = require('zmq')
let rep = zmq.socket('rep')
rep.bindSync('tcp://*:8888')
rep.on('message', function (intro, count) {
    rep.send(['World ', count])
})
```

```
// Server 2
const zmq = require('zmq')
let rep = zmq.socket('rep')
rep.bindSync('tcp://*:8889')
rep.on('message', function (intro, count) {
    rep.send(['World ', count])
});
```

Este es el programa cliente:

```
// Client
const zmq  = require('zmq')
let dealer = zmq.socket('dealer')
let msg = ["", "Hello ", 0]
let host = "tcp://127.0.0.1:888"

dealer.connect(host + 8)
dealer.connect(host + 9)

setInterval(function() {
  dealer.send(msg)
  msg[2]++
}, 1000)

dealer.on('message', function(del, seg1, seg2) {
    console.log('Response: ' + seg1 + seg2)
})
```

Como podemos ver, los tres procesos se ejecutan en el mismo ordenador. Hay dos servidores, escuchando en los puertos 8888 y 8889, respectivamente, utilizando un socket Rep. Por otra parte, el cliente usa un socket Dealer conectado a ambos puertos. Eso implica que las peticiones se repartirán circularmente entre cada servidor: los mensajes con número impar al puerto 8888 y los mensajes con número par al 8889.

El programa cliente ofrece dos diferencias cuando su código se compara con un programa cliente equivalente que usara un socket Req. Primero, el vector "msg" contiene ahora tres segmentos. El primero es un delimitador que no llegará a entregarse a los procesos servidores y que no se necesitaría en un socket Req. De hecho, el callback que gestiona el evento "message" en los programas servidores solo tiene dos parámetros (el primero, "intro", recibe la cadena "Hello " mientras que el segundo, "count", recibe el número de mensaje). Recuérdese que los delimitadores son añadidos automáticamente por los sockets Req durante el envío y son extraídos y guardados automáticamente por los sockets Rep durante la recepción. Segundo, el callback para el evento "message" en el programa cliente tiene tres parámetros mientras que las respuestas enviadas por los programas servidores solo tienen dos segmentos (cadena "World " y el número de mensaje, respectivamente). De hecho, el primer parámetro del callback ("del") se utiliza para recibir el delimitador que añadió automáticamente durante el envío el socket Rep servidor. Ese primer parámetro no llega a utilizarse en el código del callback. Realmente, este primer parámetro sirve para descartar el delimitador.

La figura 5 muestra cómo la primera petición se ha transmitido del cliente al servidor. En estas figuras, cada etapa muestra al proceso cliente en la mitad izquierda y al proceso servidor en la mitad derecha. Los números ubicados a la izquierda del contenido de cada segmento muestran el tamaño del segmento (en octetos). Esa es la forma habitual de empaquetar los mensajes en ØMQ. En el primer paso, el cliente envía la petición, que incluye tres segmentos. Ese mensaje es mantenido temporalmente en la cola de envío del socket Dealer, hasta que la biblioteca llega a enviarlo al servidor. El paso 2 muestra su llegada al proceso servidor. En ese momento es ubicado en la cola de recepción del socket Rep. En esa cola, la biblioteca identifica el segmento delimitador y lo separa del resto del mensaje. Finalmente, en el paso 3, el cuerpo del mensaje (sus dos segmentos principales) se entrega a la aplicación servidora. El delimitador se guarda en los buffers de la biblioteca para reincluirlo después en la respuesta.

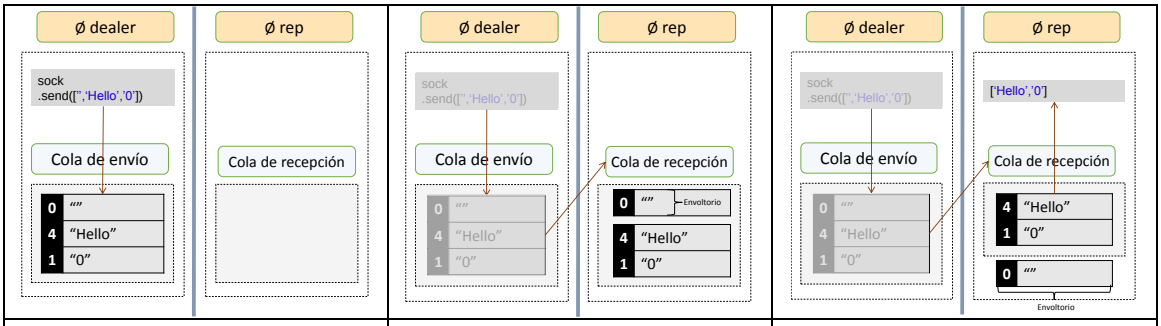

Ilustración 5. Pasos en la transmisión de una petición del cliente al servidor.

Posteriormente, el servidor procesa el mensaje y envía la respuesta correspondiente. Los pasos en la transmisión de esa respuesta se muestran en la Ilustración 6. Primero, el servidor construye la respuesta con dos segmentos y utiliza la operación send(). La biblioteca guarda el mensaje en la cola de envío, añadiéndole el envoltorio correspondiente que en este caso solo contiene el delimitador. En el paso 2, el mensaje se transmite al cliente. En el paso 3, como los sockets Dealer no gestionan delimitadores, la respuesta entera se entrega al proceso cliente. En el callback asociado al evento "message", el delimitador es descartado, como ya hemos explicado anteriormente.

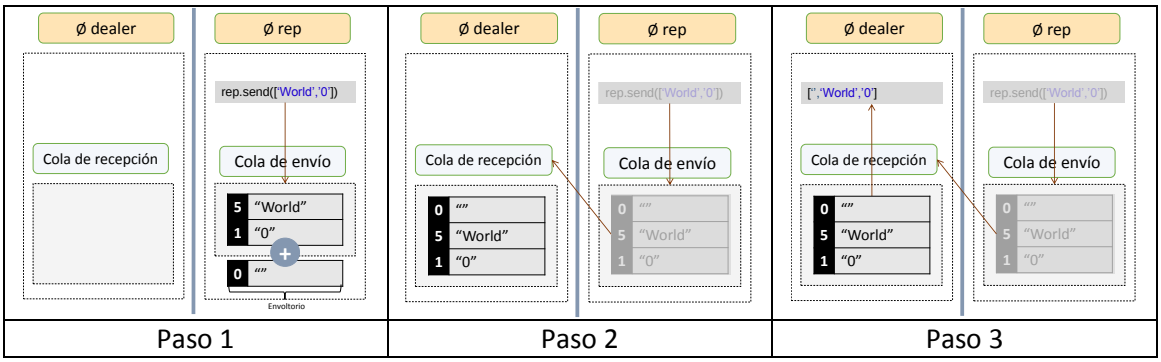

Ilustración 6. Pasos en la transmisión de una respuesta del servidor al cliente.

El cliente utilizado en este ejemplo es asincrónico pero todavía interactúa con un servidor sincrónico. Esta configuración muestra un buen comportamiento en caso de fallos. Revisemos para ello la ejecución mostrada en la Ilustración 3. En aquella ejecución un solo cliente interactuaba con dos servidores. En aquel caso, el cliente era sincrónico y no podía recuperarse del fallo de uno de los servidores de una manera transparente. Si ese servidor fallaba tras recibir la petición y antes de enviar su respuesta, el socket Req del cliente se bloqueaba para siempre.

Sin embargo, ahora hemos sustituido ese cliente con socket Req por otro con socket Dealer. Los sockets Dealer también utilizan una política de turno circular cuando son conectados con varios sockets. Por ello, este cliente también envía los mensajes impares al servidor del puerto 8888 y los mensajes pares al servidor del puerto 8889. Si el segundo servidor falla, el cliente continúa enviando peticiones sin bloquearse, pues utiliza un socket asincrónico. Más pronto o más tarde, reiniciaremos el segundo servidor, para que escuche de nuevo en el puerto 8889. En ese momento, como ØMQ gestiona las conexiones asincrónicamente y con persistencia débil (que permite mantener los mensajes enviados y todavía no recibidos), ese servidor será automáticamente visible para el proceso cliente, y todos esos mensajes pendientes se podrán entregar al servidor. Así, para este escenario concreto, ØMQ consigue proporcionar transparencia de fallos para el proceso cliente y ninguno de los mensajes llegará a perderse.

4.4.2. Sockets Router

Los sockets Router son bidireccionales asincrónicos. Son utilizados por los servidores cuando implantamos interacciones cliente/servidor asincrónicas. Así, los procesos servidores reciben peticiones y envían respuestas con este tipo de socket. Por tanto, estos sockets sustituyen a los sockets Rep.

La principal diferencia entre los sockets Rep y Router es su nivel de sincronía. Los sockets Rep son sincrónicos (por ejemplo, bloquean la recepción de nuevas peticiones hasta que se haya enviado la respuesta para la petición actual) y los sockets Router son asincrónicos (jamás se bloquea con ellos la recepción de una petición). Pero también existe una segunda diferencia: los sockets Rep gestionan automáticamente el uso de delimitadores y envoltorios, tanto en la recepción como en el envío, mientras que los sockets Router gestionan la identidad de las conexiones en lugar de preocuparse por los delimitadores y los envoltorios.

La gestión de la identidad de las conexiones comprende estas características:

- Cada conexión con otro socket es etiquetada con una identidad. El identificador de la conexión puede ser asignado aleatoriamente por la biblioteca si el programador no ha asignado explícitamente algún identificador.
- Para especificar explícitamente el identificador de una conexión, el programador debe asignar un valor a la propiedad "identity" del otro socket. **¡Esa asignación debe darse antes de utilizar el método connect() de ese socket!** Es una restricción importante. ¡No lo olvides si pretendes asignar algún identificador a una conexión!
- Durante la recepción de mensajes, el socket Router añade automáticamente como primer segmento el identificador de la conexión a cada uno de los mensajes entregados a la aplicación servidora.
- Durante el envío de mensajes, el socket Router necesita que el primer segmento de cada mensaje sea el identificador de alguna de las conexiones existentes. Si ese primer segmento no corresponde a ningún identificador de conexión, el mensaje es descartado y no se envía nada. Por el contrario, si es un identificador de conexión válido, el socket Router lo utiliza para seleccionar la conexión a usar en ese envío y ese segmento se elimina del mensaje. En ese caso, el resto del mensaje se deja en la cola de envío de esa conexión y es transmitido hacia el cliente.

Revisemos qué pasos se darán cuando un proceso cliente ejecute estas sentencias…

```
const zmq=require('zmq')
let sock = zmq.socket('req')
sock.identity='peer1'
sock.connect('http://127.0.0.1:8000')
sock.send('my request')
```

...si el proceso servidor correspondiente ejecuta este programa:

```
const zmq=require('zmq')
let so = zmq.socket('router')
so.bindSync('http://127.0.0.1:8000')
so.on('message', function(conId,del,msg) {
   so.send([conId,del,'my answer'])
})
```

Obsérvese que el socket Router no gestiona delimitadores de forma implícita. Por ello, los pasa durante la recepción al programa servidor y el programa servidor debe mantenerlos en su respuesta. De otra manera, el socket Req cliente no recibirá el delimitador que espera en su respuesta.

Para empezar, ambos programas han creado en sus respectivas dos primeras líneas de código un socket. Ese socket es de tipo Req en el cliente y de tipo Router en el servidor. Ese hecho se muestra en el primer paso de la Ilustración 7. El paso 2 corresponde a la ejecución en el cliente de la sentencia que asigna el valor "peer1" a la identidad del socket Req. Esa identidad se muestra en la caja que modela el socket, situada en la parte superior de la mitad izquierda (proceso cliente) de esa figura. La instrucción siguiente conecta ese socket al URL en el que el socket Router del servidor está escuchando. El efecto de esa instrucción se muestra en el paso 3 de la Ilustración 7. Ahora, el socket Router ha creado un par de colas de recepción y envío para esa conexión y ambas están etiquetadas con el identificador "peer1".

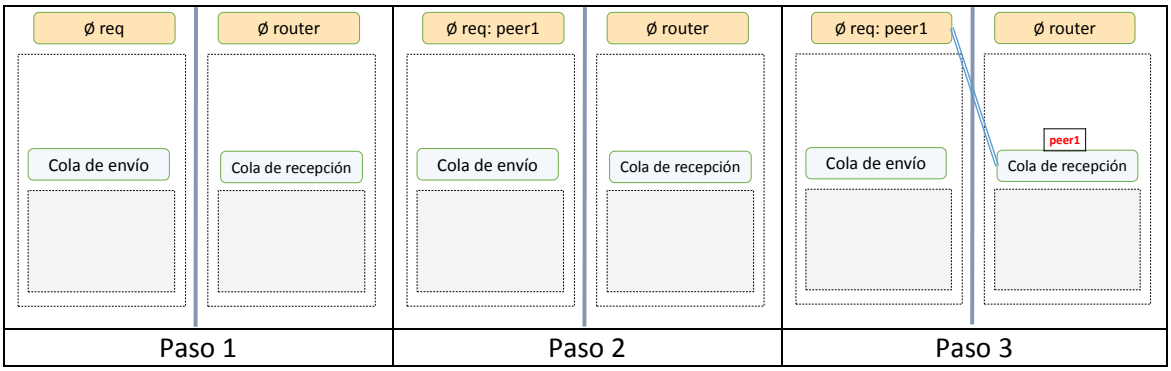

Ilustración 7. Pasos en la conexión de un socket Router.

Una vez se ha establecido la conexión, el proceso cliente puede enviar peticiones al servidor. La Ilustración 8 muestra los pasos en esas transmisiones de peticiones. En el paso 1, el cliente invoca el método send(). Como resultado, la biblioteca deja la petición en la cola de envío, añadiendo un delimitador al principio de su lista de segmentos, pues ese socket es de tipo Req. Posteriormente, en el paso 2, ese mensaje es transmitido al servidor. Así, el mensaje se añade a la cola de recepción etiquetada como "peer1" del Router. Podría haber otras colas de recepción en ese socket, dependiendo del número de conexiones realizadas hasta ahora. Esa petición recibida llegará a entregarse al proceso servidor. En ese momento, tal como muestra el paso 3, la biblioteca añade el identificador de conexión como primer segmento a ese mensaje. Como los sockets Router no eliminan ni añaden delimitadores, el delimitador añadido en el paso 1 por el socket Req todavía estará ahí. Por tanto, en este ejemplo, el proceso cliente ha enviado un mensaje con un solo segmento, pero el proceso servidor ha recibido un mensaje con tres segmentos. Eso indica que, en todas las interacciones Req-Router, se añadirán dos segmentos en la parte inicial de cada mensaje transmitido.

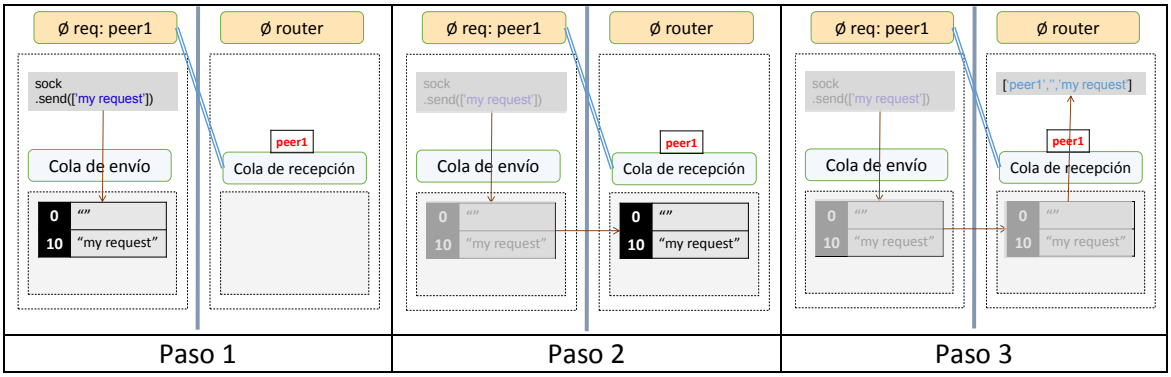

Ilustración 8. Pasos en la transmisión de una petición (Req-Router).

Posteriormente, el servidor procesa la petición recibida y construye un mensaje de respuesta. La Ilustración 9 muestra los pasos en la transmisión de una respuesta al cliente. Como podemos ver en el programa, el servidor mantiene el contenido de los dos primeros segmentos en los parámetros 'conId' y 'del' del callback. Esos dos segmentos son ubicados en esas mismas posiciones en el mensaje de respuesta. El contenido real de la respuesta debe empezar a partir del tercer segmento. En nuestro ejemplo, solo la cadena "my answer" es la respuesta real. El identificador de conexión, utilizado como primer segmento, lo utiliza la biblioteca para elegir la conexión a usar para enviar la respuesta. Una vez localizada, ese segmento es eliminado del mensaje. Por tanto, solo el segundo y tercer segmentos llegan a dejarse en la cola de envío.

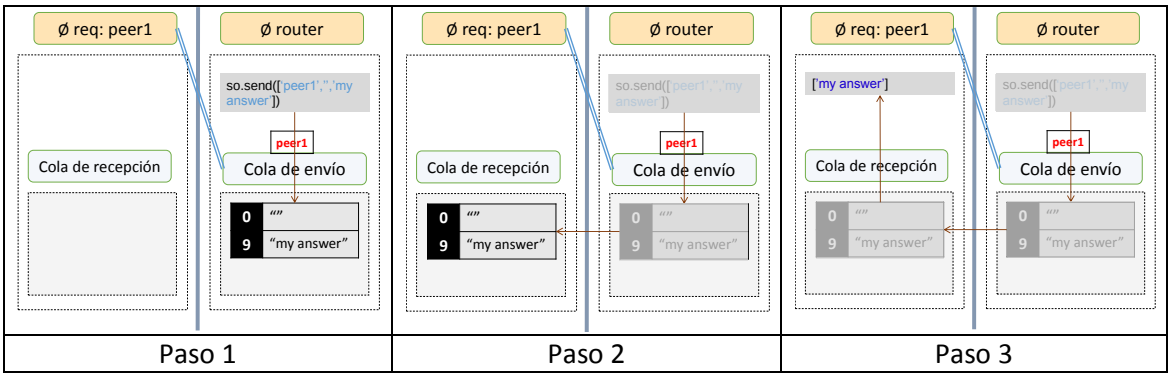

Ilustración 9. Pasos en la transmisión de una respuesta (Router-Req).

En el paso 2, ese mensaje llega a ser el primero en la cola de envío y por ello se transmite al proceso cliente. En algún momento será recibido y ubicado al final de la cola de recepción del socket Req del cliente. En el paso 3 ese mensaje llega a ser el primero en esa cola de recepción. En ese momento, se entrega la respuesta al cliente. Como el proceso cliente utiliza un socket Req, ese socket comprueba que el primer segmento del mensaje sea un delimitador. Si es así, como en este caso, el mensaje puede ser entregado y el delimitador es eliminado. Así, el segmento "my answer" es entregado como el contenido real de este mensaje de respuesta.

Obsérvese que podría haber múltiples clientes conectados al mismo servidor. Además, el servidor puede tener múltiples operaciones en su interfaz. Alguna de ellas podría necesitar una ejecución larga y asincrónica (por ejemplo, la lectura de un fichero para devolver su contenido). Así, el tiempo de servicio para peticiones diferentes puede variar mucho. Por ello, la gestión de las conexiones en el Router es crucial para devolver cada respuesta a su cliente correcto. En el programa servidor, solo necesitamos mantener los dos primeros segmentos de la petición, copiando su contenido en los dos primeros segmentos de la respuesta, como hemos mostrado en el ejemplo que acabamos de describir.

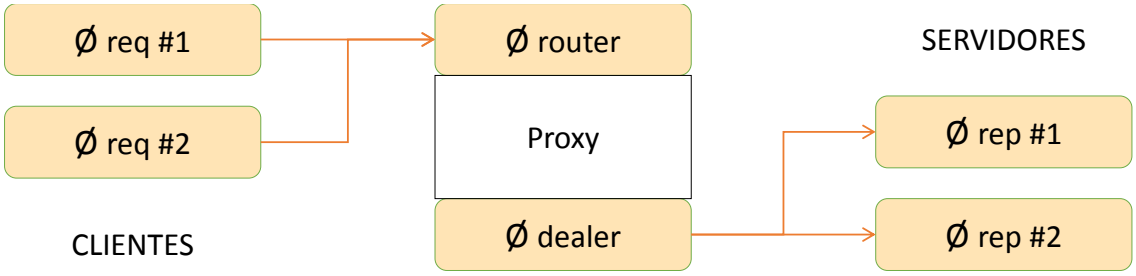

Ilustración 10. Broker Router-dealer.

4.4.3. Un broker Router-Dealer

Podemos utilizar sockets Router y Dealer para desarrollar un agente proxy que proporcione transparencia de replicación. Cuando un servicio determinado se replica, en un escenario ideal los procesos clientes no necesitarían conocer la identidad y dirección de cada réplica de ese servicio. En su lugar, deberían conocer una única dirección: la de un proxy servidor. Ese proxy oculta cuántas instancias servidoras hay. Además, podrá equilibrar la carga entre todas las réplicas servidoras. Para ello, facilita un socket Router a los clientes y un socket Dealer a las réplicas servidoras, tal como muestra la Ilustración 10. Esos dos sockets están asociados a sus "endpoints" respectivos. Los clientes se conectarán al "endpoint" del Router (y así solo

necesitan conocer un solo URL, proporcionándoles transparencia de replicación) y las réplicas servidoras al "endpoint" del Dealer. Este agente proxy que gestiona los sockets Router y Dealer se comporta como un gestor o "broker" de mensajes. Por ello, podemos referirnos a él como agente proxy o broker de mensajes.

Su código es muy sencillo: todo lo recibido desde el socket Router (normalmente, peticiones de los clientes) debe ser enviado, tal cual, por el socket Dealer y todo lo recibido desde el socket Dealer (normalmente, respuestas de los servidores) debe enviarse por el socket Router. Por tanto, su código podría ser el siguiente:

```
const zmq=require('zmq')
let fe=zmq.socket('router')  // frontend socket (clients)
let be=zmq.socket('dealer')  // backend socket (servers)
fe.bindSync('tcp://*:8000')
be.bindSync('tcp://*:8001')
fe.on('message', function() {
  let args=Array.from(arguments)
  be.send(args)
})
be.on('message', function() {
  let args=Array.from(arguments)
  fe.send(args)
})
```

Ya hemos visto programas clientes que usan sockets Req y programas servidores que utilizan sockets Rep. Cualquiera de ellos podría combinarse con este broker. Por tanto, supondremos que tenemos un cliente que envía peticiones con un único segmento con valor numérico (por ejemplo, 100) a un servidor que calcula la raíz cuadrada del valor recibido. La identidad del socket Req cliente podría ser "id234" en este ejemplo. Realizaremos seguidamente una traza de los pasos a considerar para estos mensajes de petición y respuesta. Con ella se ilustrará cómo funciona este broker y cómo los sockets Rep gestionan los envoltorios. Los pasos de la traza serían los siguientes:

1. El cliente envía por un socket Req el valor 100 como único contenido de su petición. El contenido de la petición es: ['100'].
2. El socket Req añade un delimitador inicial a ese mensaje antes de transmitirlo hacia el servidor. Así, el contenido de la petición es ahora: ['','100'].
3. La petición llega ahora a la cola de recepción del Router. El contenido del mensaje sigue siendo el mismo: ['','100'].
4. El socket Router entrega la petición al programa broker. Como esa petición se ha recibido por la conexión 'id234', su identificador de conexión se añade como segmento inicial del mensaje. El contenido de la petición es ahora: ['id234','','100'].
5. El programa broker envía la petición por el socket Dealer. Vuelca todos sus segmentos. Por ello, el contenido de la petición sigue siendo el mismo.
6. El socket Dealer comprueba sus conexiones actuales. Puede haber múltiples réplicas servidoras. Escoge la siguiente en orden circular y propaga el mensaje por esa conexión.
7. Como resultado del paso anterior, la petición estará ubicada en la cola de recepción del socket Rep de alguna réplica servidora. Por ser un socket Rep, esa petición será bloqueada hasta que no haya una petición en curso en esa réplica. Eso implica que to-

das las respuestas anteriores ya habrán sido enviadas. En ese momento, el socket Rep revisa el mensaje entrante para localizar su primer delimitador. Ese delimitador y todos los segmentos que lo precedan serán el "envoltorio" de esta petición. Ese envoltorio es separado y mantenido por el socket Rep. El resto del mensaje se entrega al proceso servidor. Por tanto, el mensaje recibido por el servidor es ['100'] y el envoltorio guardado es ['id234',''].

8. La réplica servidora procesa esa petición. La raíz cuadrada de 100 es 10. Por tanto, ['10'] será la respuesta enviada al cliente.
9. El socket Rep gestiona ese envío. Lo primero que hace es incluir el envoltorio en sus primeros segmentos. Por ello, la respuesta que se deja en la red es ['id234','','10'].
10. El socket Dealer del broker recibe esa respuesta. Los sockets Dealer no transforman de ninguna manera los mensajes. Por tanto, esos mismos tres segmentos son entregados al proceso broker.
11. El broker envía esos segmentos a través del socket Router. En ese punto, el socket Router examina el primer segmento para comprobar si corresponde a alguno de los identificadores de conexión existentes. Sí que corresponde. Por tanto, la conexión "id234" es seleccionada y ese primer segmento es descartado en este paso. Como resultado, el mensaje ['','10'] se envía al cliente "id234".
12. La cola de recepción en el socket Req del cliente recibe esa respuesta, elimina su delimitador inicial y entrega el resto de la respuesta al proceso cliente. Como se esperaba, el único contenido de esa respuesta es el valor 10 en este paso. Por tanto, el cliente recibe el valor correcto a su petición: la raíz cuadrada de 100 es 10.

Como hemos visto, el segmento que contiene el identificador de conexión y que fue incluido por el socket Router durante su gestión de la petición es clave para devolver la respuesta al cliente apropiado. Además, en estas gestiones el programador no necesita realizar ninguna acción especial sobre el contenido de los mensajes. La gestión es realizada de manera automática en cada socket.

Esta traza también ha mostrado un ejemplo sobre el funcionamiento de la gestión de "envoltorios" en los sockets Rep. Esto ha permitido mantener la identidad de las conexiones cliente-broker en los procesos servidores, sin necesidad de modificar para nada el propio programa servidor.

Como resumen de lo que acabamos de describir, la Ilustración 11 muestra de manera esquemática cómo se llega a transformar el contenido de los mensajes en una interacción cliente/trabajador mediada por un *broker* Router/Dealer. Se asume que el cliente envía una petición con un solo segmento 'm' y el trabajador devuelve una respuesta con un solo segmento 'r'. Los sockets Req añaden al enviar un delimitador (representado como ',') que después eliminan al recibir. Los sockets Router añaden la identidad del cliente (segmento 'c') al recibir y la eliminan al enviar, aprovechándola para seleccionar qué conexión cliente recibirá ese mensaje. Los sockets Dealer no añaden ni eliminan segmentos en los mensajes. Los sockets Rep guardan (y eliminan del mensaje) el envoltorio del mensaje al recibir la petición y lo añaden al enviar la respuesta.

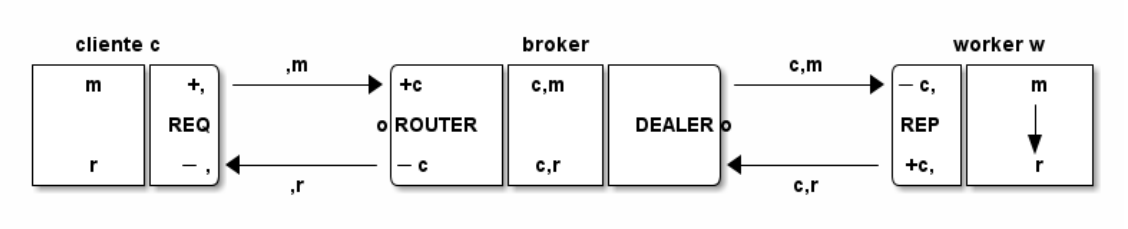

Ilustración 11. Gestión de mensajes en un broker Router-Dealer.

5. Otros middleware

En esta unidad se han analizado algunos tipos de middleware centrados en comunicaciones. Las comunicaciones no son el único ámbito donde se han utilizado soluciones basadas en *middleware*. Hay bastantes más. Como otros ejemplos cabe citar la temática de seguridad y la de soporte transaccional.

Los *middleware* relacionados con la seguridad gestionan protocolos de autenticación y de autorización.

La autenticación consiste en que una tercera parte pueda asegurar que un determinado agente es quien dice ser, verificando que el identificador presentado le pertenece. Un ejemplo de protocolo estándar de autenticación es OpenID [9]. Cualquier capa de software que soporte OpenID sobre diferentes plataformas será un ejemplo de middleware de autenticación.

La autorización es el mecanismo mediante el que una tercera parte autoriza una petición. El protocolo estándar comúnmente aceptado para regular las tareas de autorización es OAuth 2.0 [10]. Tomando como base este estándar de autorización se rediseñaron los procedimientos de autenticación, generando así un nuevo estándar: OpenID Connect [11].

En un sistema distribuido pueden necesitarse transacciones para asegurar que cierta secuencia de operaciones ejecutadas tendrá garantías de atomicidad. Con ese soporte, si alguna de las operaciones de la secuencia abortase, toda la secuencia sería abortada. Para proporcionar estas garantías se han utilizado transacciones anidadas [12]. Existen algunos middleware para proporcionar este soporte. CORBA [13] es un ejemplo.

La lista de áreas donde se utilizan sistemas *middleware* no termina aquí. En un informe [14] del año 2004, Gartner clasificó los sistemas middleware de la siguiente manera:

BÁSICOS	Gestión de datos	
	Comunicaciones	• Llamada remota a procedimiento • Sistemas de mensajería (MOM) • Comunicación multifuncional
	Plataformas	• Monitores de procesamiento transaccional • ORB • Servidores de aplicaciones • Servicios web • Portales • Suites de aplicaciones • Sistemas gestores de bases de datos

INTEGRACIÓN DE OTROS MIDDLEWARE	Adaptadores Transformadores Encaminamiento inteligente Navegación de datos Gestión de procesos empresariales Gestión de eventos empresariales Motores de reglas de negocio
INTEGRACIÓN DE APLICACIONES	Integración de módulos Monitorización de la actividad empresarial

Como puede observarse, en esta unidad sólo hemos cubierto una parte muy pequeña de este conjunto. Es suficiente. Lo interesante es conocer el concepto y las variedades más importantes para el desarrollo de aplicaciones distribuidas. Para ello los middleware de comunicaciones proporcionan una buena base que podrá extenderse posteriormente.

6. Conclusiones

La complejidad de los sistemas distribuidos debe ser resuelta mediante una gestión adecuada de su código y servicios. Los estándares facilitan este escenario familiarizándonos con las técnicas a emplear.

Los *middleware* (nivel de la arquitectura entre la aplicación y el sistema operativo, siendo este último el responsable de las comunicaciones) implantan soluciones comunes a estos problemas. Para ello utilizan una interfaz (API) y unos protocolos estándar, implantándolos sobre múltiples plataformas. De esta manera se asegura la interoperabilidad entre los componentes del sistema distribuido.

Los *middleware* más importantes proporcionan funcionalidad relacionada con la comunicación. En una primera etapa (década de los ochenta del siglo pasado) se basaron en el mecanismo de llamada a procedimiento remoto (RPC). Posteriormente, con la llegada de los sistemas orientados a objeto pasaron a proporcionar invocaciones a métodos remotos (RMI). En ambos casos se facilitaba transparencia de ubicación, ofreciendo la imagen de un espacio de recursos común. Eso simplificaba las tareas de programación pero comprometía la eficiencia del sistema, pues inducía a bloqueos a la hora de acceder a los recursos compartidos. Los sistemas de mensajería (MOM) solucionaron estos problemas al ofrecer un modelo de programación asincrónico, sin recursos compartidos.

Existe una gran variedad de sistemas *middleware*. Los relacionados con comunicaciones son sólo una pequeña parte dentro del amplio conjunto de sistemas existentes.

Referencias

[1] C. A. R. Hoare, «Monitors: An Operating Systems Structuring Concept,» *Commun. ACM,* vol. 17, nº 10, pp. 549-557, 1974.

[2] H. Seifzadeh, H. Abolhassani y M. S. Moshkenani, «A survey of dynamic software updating,» *Journal of Software: Evolution and Process,* vol. 25, nº 5, pp. 535-568, 2013.

[3] P. A. Bernstein, «Middleware: A Model for Distributed System Services,» *Commun. ACM,* vol. 39, nº 2, pp. 86-98, 1996.

[4] A. Birrell y B. J. Nelson, «Implementing Remote Procedure Calls,» *ACM Trans. Comput. Syst.,* vol. 2, nº 1, pp. 39-59, 1984.

[5] T. D. Chandra y S. Toueg, «Unreliable Failure Detectors for Reliable Distributed Systems,» *J. ACM,* vol. 43, nº 2, pp. 225-267, 1996.

[6] A. S. Tanenbaum y M. van Steen, Distributed Systems: Principles and Paradigms, 1ª ed., ISBN: 978-0130888938, Prentice-Hall, 803 pgs, 2002.

[7] S. Vinosky, «Advanced Message Queueing Protocol,» *IEEE Internet Computing,* vol. 10, nº 6, pp. 87-89, 2006.

[8] P. Hintjens, «ØMQ - The Guide,» [En línea]. Available: http://zguide.zeromq.org/page:all. [Último acceso: 20 junio 2014].

[9] OpenID_Foundation, «Open ID Authentication 2.0 - Final Specification,» 5 diciembre 2007. [En línea]. Available: http://openid.net/specs/openid-authentication-2_0.html. [Último acceso: 20 junio 2014].

[10] D. Hardt, "RFC6749: The OAuth 2.0 Authorization Framework", Internet Engineering Task Force (IETF), 2012.

[11] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros y C. Mortimore, «OpenID Connect Core 1.0 Specification,» 25 febrero 2014. [En línea]. Available: http://openid.net/specs/openid-connect-core-1_0.html. [Último acceso: 20 junio 2014].

[12] J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Tesis doctoral, Massachusetts Institute of Technology, 1981.

[13] OMG, «Common Object Request Broker Architecture (CORBA),» 16 noviembre 2012. [En línea]. Available: http://www.omg.org/spec/CORBA/3.3/. [Último acceso: 19 junio 2014].

[14] B. Lheureux, R. Schulte, Y. Natis, D. McCoy, B. Gassman, J. Sinur, J. Thomson, M. Pezzini, F. Kenney, T. Friedman, M. Gilbert y M. Phifer, "Who's Who in Middleware, 1Q04", Gartner RAS Core Strategic Analysis Report R-22-2153: Disponible en: http://www-01.ibm.com/software/info/websphere/partners4/articles/gartner/garwho.html, 2004.