E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Replicación Activa en NodeJS con 0MQ

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Raúl Osés Iturmendi

Director: José Ramón González de Mendivil

Pamplona, 26/01/2015

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Dedicado a todas las personas

que me apoyan,

me quieren y

creen en mí.

Gracias a José Ramón González de Mendivil

por su ayuda y esfuerzo.

Resumen

Mi trabajo final de grado se centra en el aprendizaje, desarrollo y divulgación del entorno de programación NodeJS, un entorno orientado a eventos que está basado en el lenguaje de programación JavaScript en el lado del servidor. El enfoque de NodeJS es el desarrollo de aplicaciones de red altamente escalables. La escalabilidad se proporciona por la eficiencia obtenida a lo largo de los años en la optimización de código en los motores de JavaScript y por la delegación asíncrona a tareas propias del sistema operativo. Aprovechando estas cualidades, desarrollo la implementación de un algoritmo basado en el modelo de replicación activa consistente en mantener copias de objetos de datos en distintos almacenes de datos. Con este modelo se construye un sistema distribuido tolerante a fallos y de alta disponibilidad. Para los protocolos de comunicación se utilizan de los socket 0MQ y se desarrollan algunos ejemplos del uso de cada tipo de ellos.

Palabras clave

NodeJS, JavaScript, Alta escalabilidad, Orientado a eventos, 0MQ, Protocolo, Aplicación de red, Copias de objetos, Replicación activa, Tolerancia a fallo.

1 Índice

2 Objeto y desarrollo

Los objetivos de este proyecto son los siguientes:

- Estudiar NodeJS junto con JavaScript y, en concreto, la librería 0MQ, para generar la base de unas prácticas futuras para la asignatura de Aplicaciones sobre Redes de Ordenadores (ARO) del Grado en Ingeniería Informática que se imparte en la UPNA.

- Probar las facilidades de la programación asíncrona, que ofrece NodeJS, para el desarrollo e implementación de un protocolo de replicación activa.

El desarrollo de esta memoria es el siguiente:

En el capítulo 3 se explican los aspectos básicos de la plataforma NodeJS, sus características, capacidades y módulos. Se muestran algunas aplicaciones en las que se hace uso de la plataforma, como el típico programa "Hello World".

En el capítulo 4 veremos los patrones de comunicación que nos brinda la librería 0MQ. Se han desarrollado, para cada patrón, ciertos programas como ejemplos para ver y entender el uso y la programación de los mismos. Además, se muestran los resultados de la ejecución cada programa.

En el capítulo 5 presentaremos lo que es la Replicación Activa y todo lo que conlleva el desarrollo de la misma en tres niveles. Se explican todos los programas que la componen mediante diagramas y pseudocódigo. Se muestran los resultados de una ejecución cualquiera y los códigos desarrollados se encuentran en la sección de anexos.

En el capítulo 6 se exponen las conclusiones de este proyecto fin de carrera y de los trabajos que se pueden realizar en un futuro.

En el apéndice se explica la programación asíncrona y sus diferencias con la programación secuencial. Además, se muestran ciertos ejemplos de la programación mediante eventos y promesas.

En los anexos añadimos el código desarrollado de los programas para el proyecto de la replicación activa. Y para finalizar, el capítulo de bibliografía y referencias.

3 Introducción a NodeJS

En este apartado de la memoria se desarrollan algunos de los aspectos básicos que NodeJS posee para que el entendimiento de la plataforma sea más sencillo.

3.1 Qué es

La página oficial www.nodejs.org define a NodeJS como *"a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices"* [1].

Por definirlo de otra manera podemos decir que, NodeJS en su forma más simple es un conjunto de librerías para programar aplicaciones escalables ubicadas en la red, que también necesitan de alto rendimiento usando el lenguaje de JavaScript. Un entorno orientado a la creación de aplicaciones para internet en el contexto del servidor.

3.2 Características

NodeJS es una plataforma prácticamente joven y a continuación se presentan algunas de las características que esta plataforma dispone para la realización de aplicaciones.

3.2.1 Asíncrono

En el apéndice (7) de esta memoria tenemos una explicación con ejemplos de lo que es la programación asíncrona y sus modelos.

La velocidad y escalabilidad de NodeJS provienen de su modelo de programación asíncrona. Para ver la diferencia entre una aplicación síncrona y una asíncrona se muestran las siguientes explicaciones para que se entienda más fácilmente [2].

Un ejemplo básico de aplicación web se realiza mediante un ciclo de peticiones seguidas de sus consiguientes respuestas. Para cada una de las peticiones, la aplicación web construye una respuesta consultando bases de datos, discos, API's etc. Mientras una petición está esperando su respuesta las demás peticiones esperan a que la web este libre, esto es una cola de procesos síncronos, donde cada proceso espera a que la respuesta del anterior haya sido devuelta. Un caso con el que podemos comparar este funcionamiento en la vida real es una cola de supermercado, donde cada cliente espera a que el que está en frente de él haya terminado.

Para entender el funcionamiento de la ejecución de una aplicación asíncrona podemos observar ejemplos de la vida real como por ejemplo ciertos restaurantes donde una vez has hecho tu pedido o "petición", te dan el plato o "respuesta" una vez que la cocina o "aplicación" la tenga preparada. NodeJS intenta no bloquear nunca subsecuencias de código susceptibles de ser ejecutadas, no va a mantener una petición esperando a que la aplicación le devuelva su respuesta bloqueando así a otras peticiones. Atiende a todas las peticiones que llegan, y devuelve la respuesta a cada una de ellas en cuanto la tiene preparada [3].

3.2.2 Lenguaje común

El desarrollo de aplicaciones donde el lado del cliente y lado del servidor se escriban en un mismo lenguaje hace que compartir la información sea sencillo. Esta característica hace que decremente el coste de desarrollo y mantenimiento para funcionalidades compartidas.

3.2.3 JavaScript en el lado del servidor

La programación asíncrona utilizada para el desarrollo de aplicaciones de red no es algo nuevo, pero sí lo es el uso de JavaScript, no solo en el lado del cliente como hasta ahora se venía haciendo, sino también en el lado del servidor. Esto es algo novedoso y lo que diferencia a NodeJS de los demás lenguajes para la programación asíncrona como pueden ser *EventMachine*[1] o *Twisted*[2].

JavaScript es un lenguaje asíncrono pero que no posee librerías estándar de bloqueos de I/O o código para red y es aquí donde NodeJS provee toda la funcionalidad que se espera para la programación con un lenguaje de manera totalmente asíncrona y no bloqueante.

3.2.4 Velocidad

La velocidad que presenta NodeJS como plataforma para el desarrollo de aplicaciones web se debe principalmente a los avances recientes que se han obtenido en los motores de JavaScript. Dada la expansión de las aplicaciones en la red, la ejecución de JavaScript se ha convertido en un factor importante en el desarrollo de las mismas.

La competencia entre buscadores sobre la ejecución de código JavaScript ha ido mejorando con el tiempo haciendo así que el lenguaje sea mejor que otros lenguajes interpretados. El resultado de esto es el interés general de ejecutar este lenguaje en el lado del servidor, lo cual permite la plataforma NodeJS de manera sencilla y práctica.

[1] Librería para la programación asíncrona dirigida a eventos en Ruby.

[2] Librería para la programación asíncrona dirigida a eventos en Python.

3.3 Event loop

Una parte fundamental de NodeJS es el concepto de la utilización de eventos. La utilización de eventos es uno de los pilares fundamentales en los que se sustenta Javascript. Esto se debe a que Javascript siempre ha tenido que trabajar en interacción con el usuario por lo que ya tiene experiencia. Estos eventos los podemos apreciar en las típicas páginas web donde utilizan eventos como "*onClick*" o "*onMouseover*" para interactuar con el usuario y cambiar partes de la página web de manera dinámica. NodeJS utiliza los eventos para la interacción mediante peticiones de todas las partes del sistema.

Todas las actividades del sistema deberán ser no bloqueantes, es decir, cada una de ellas se ejecuta de manera independiente y emiten los eventos cuando sea preciso. Esto hace que la ejecución de los programas se realice mediante diversos *callbacks*. Al ser en cierto modo un estilo de programación no muy familiar, necesitamos observar ciertos patrones para programar de manera eficiente con eventos en el servidor. Este tipo de programación es comprensible ya que es intuitiva. Como en la vida real, donde estamos habituados a realizar acciones unas después de otras y algunas de ellas a partir de "*callbacks*" de otros eventos o acciones.

3.4 Especialidades de NodeJS

La especialidad de la plataforma son las aplicaciones en tiempo real y de alta concurrencia. Cualquier aplicación que necesite tener una cantidad alta de usuarios conectados al mismo tiempo, es un buen proyecto para la utilización de NodeJS, pero también se puede usar simplemente para servir páginas web o aplicaciones.

3.4.1 Notificaciones Push

Al mismo tiempo que la red avanza incluyendo páginas web altamente interactivas, un elemento común para añadir a las aplicaciones son las notificaciones de tipo *push*. Existen tres maneras distintas de obtener información del servidor para hacérsela llegar al cliente.

3.4.1.1 Traditional Polling

El cliente realiza una petición al servidor en un momento determinado pidiéndole cierta información. Esta petición solo se puede realizar mientras el servidor tenga el intervalo para escuchar activo.

Reduciendo este intervalo obtenemos un aumento de carga en el servidor ya que todas las peticiones de clientes se realizan en ese intervalo aun cuando no hay nueva información para ellos. Este modelo no escala correctamente, por ello no se utiliza en sitios con tráfico alto.

3.4.1.2 Realtime Socket

Es un canal persistente, full-duplex y bidireccional que se abre entre cliente/servidor y ambos pueden enviar información por el en cualquier momento sin casi tener superposición. El problema con este método es que una gran cantidad de clientes y servidores ya no soportan este tipo de notificaciones.

3.4.1.3 Long Polling

Es la manera en la que trabajan la gran mayoría de webs en tiempo real. Los clientes realizan la petición y esperan la respuesta. Si el servidor no tiene la respuesta para el cliente, mantiene la petición abierta y no devuelve nada. Una vez tenga la información requerida, se la devuelve al cliente.

Esta es una manera eficiente de realizar las notificaciones *push* pero el costo es que para cada cliente, su conexión con el servidor debe estar abierta indefinidamente. La ventaja de NodeJS es que puede manejar grandes cantidades de conexiones con un solo proceso del servidor.

3.4.2 Respuestas Streaming

Las tecnologías web actuales se fundamentan en gran parte en los ciclos de petición y respuesta. Es sencillo de desarrollar pero deja muchos beneficios potenciales sin atender. Esperar a que el servidor nos devuelva toda la respuesta completa o mandar partes de esa respuesta para mejorar la experiencia del usuario.

Las respuestas en *streaming* mejoran la forma de hacerlo. Podemos devolver pequeñas partes de la web que ya están listas para el cliente. Esta forma es imposible de realizar para muchos de los *frameworks* existentes que están construidos con el ciclo de peticiones y respuestas, pero es lo más natural en NodeJS.

Un ejemplo en NodeJS en el que cuando llega una petición, el servidor envía las cabeceras y luego mediante un intervalo envía la palabra *data* cada medio segundo y termina con un *Done!* a los 5 segundos.

```
01: var http = require('http');
02: http.createServer(function (request, response) {
03:   response.writeHead(200, {'Content-Type': 'text/plain'});
04:   response.write("Starting...\n");
05:   var dataWriter = setInterval(function () {
06:     response.write("data\n");
07:   }, 500);
08:   setTimeout(function () {
09:     clearInterval(dataWriter);
10:     response.end("Done!");
11:   }, 5000)
12: }).listen(8124);
```

Tabla 1 : Código servidor http sencillo

3.4.3 Algunos proyectos importantes

La plataforma NodeJS está todavía en sus primeros momentos de vida y aun así evoluciona rápidamente y ya hay proyectos importantes que han surgido para resolver problemas comunes en el desarrollo con NodeJS.

3.4.3.1 Express

Express es un *framework* para aplicaciones web que ofrece un conjunto robusto de herramientas para aplicaciones web y móvil [4]. Todas estas funciones de Express se realizan junto con las funciones que ofrece NodeJS. Express es un framework para web similar a *Sinatra*[3] y *Bottle*[4].

3.4.3.2 Socket.IO

Es una librería para escribir de forma sencilla el código para aplicaciones web que actúan en tiempo real abstrayéndose del navegador [5]. Ofrece el desarrollo de comunicaciones en tiempo real, bidireccionales basadas en eventos en cualquier plataforma, navegador o dispositivo centrándose en la velocidad y en la confiabilidad.

3.4.3.3 Mongoose

Es un ORM[5] que interactúa con una base de datos MongoDB[6]. Esta base de datos es una elección común para almacenar datos en aplicaciones NodeJS. Provee de soluciones directas para modelar la información de la aplicación en un entorno asíncrono [6].

3.5 Módulos

NodeJS implementa el estándar de los módulos CommonJS para JavaScript en el lado del servidor. Cada módulo tiene su propio contexto separado de los demás módulos, esto significa que los módulos no pueden interferir en otros módulos. Organizando el código de tu aplicación en diferentes módulos bien definidos puede ayudar a mantener tu código bajo control.

[3] Framework para web de Ruby

[4] Framework para web de Python

[5] El Mapeo objeto-relacional es una técnica de programación para convertir datos entre el sistema y la utilización de una base de datos relacional.

[6] Es un sistema de base de datos NoSQL orientado a documentos desarrollado bajo el concepto de código abierto.

Los módulos son cargados cuando un proceso de NodeJS comienza. Lo mismo sucede cuando se hace uso de otros módulos, por ejemplo los instalados a partir de *npm*[7] [7] o los módulos que tú hayas creado previamente.

Para usar cualquier módulo se debe usar la función *require*. Esta función importa el módulo requerido y devuelve un objeto que representa la API de JavaScript del módulo. Para exportar desde un fichero que hayamos creado, debemos indicar que objetos y funciones exporta, estos elementos se asignan como propiedades del objeto.

3.5.1 Múltiples Procesadores (Cluster)

NodeJS, como hemos mencionado anteriormente, tiene un único hilo. Esto significa que NodeJS solo está utilizando un procesador para realizar su trabajo. Para hacer uso de más procesadores, Node provee de un módulo llamado *cluster* que permite delegar el trabajo a procesos hijos, se realiza una copia del programa en este nuevo proceso. Cada proceso hijo tiene la capacidad de comunicarse con los demás procesos mediante un socket, lo que nos permite escribir programas en NodeJS que ejecutan otros tantos programas NodeJS y delega el trabajo a ellos.

Haciendo uso de la API del módulo *cluster* podemos distribuir el trabajo a un proceso de Node en cada núcleo disponible del servidor. De esta manera se hace el mejor uso de los recursos. A continuación se muestra un ejemplo sencillo del uso de este módulo que aparece en el libro Node: Up and Running [3].

Cada proceso Node se convierte en "*master*" o "*worker*". Cuando se llama al método *fork()*, se crea un proceso hijo que es idéntico al master. El ejemplo que vemos a continuación, crea un *worker* por cada CPU, cada hijo comienza un servidor http.

```
01: var cluster = require('cluster');
02: var http = require('http');
03: var numCPUs = require('os').cpus().length;
04: if (cluster.isMaster) {
05:   console.log('Maestro');
06:   // Fork workers.
07:   for (var i = 0; i < numCPUs; i++) {
08:     console.log('Creando el hijo: '+i);
09:     cluster.fork();
10:   }
11:   cluster.on('death', function(worker) {
12:     console.log('worker ' + worker.pid + ' died');
13:   });
14: }
15: else {
16:   // Worker processes have a http server.
17:   http.Server(function(req, res) {
18:     console.log('Http Server');
19:     res.writeHead(200);
20:     res.end("hello world\n");
21:   }).listen(8000);
22: }
```

Tabla 2 : Código ejemplo uso de cluster [3]

[7] Node Package Manager

3.6 NodeJS API Guide

A continuación se exponen algunos de los módulos de NodeJS más utilizados, para ver la lista completa de módulos disponibles, consultar la página *http://www.nodejs.org/api/* [1].

- **Timers**

  El módulo *Timers* provee de funciones para establecer y borrar *timeouts* e intervalos tal y como se hace en un navegador.

- **Events**

  Contiene la clase de *EventEmitter*, o emisor de eventos, usada por muchos otros objetos de NodeJS. Define la API para instanciar generadores de eventos en los cuales se crean y quitar eventos escuchadores y como se interacciona con ellos.

- **File System**

  Funciones de escritura y lectura de ficheros y directorios, mover, copiar y renombrar ficheros. Algunas funciones tienen versiones síncronas las cuales se diferencian de las funciones originales y asíncronas escribiendo *Sync* después del nombre de la función.

- **Net**

  Crea objetos de servidores en red para escuchar conexiones y realizar algo en ellas. Permite la lectura y escritura en sockets TCP, pero en vez de ello se pueden usar otros módulos como el de http para crear servidores http.

- **Http**

  Es el módulo más usado y más importante para un desarrollador web. Creación de servidores http y tenerlos escuchando en un puerto. También contiene los objetos para las peticiones y las respuestas.

- **Https**

  Contiene las funciones para crear, esta vez, un servidor de https o crear peticiones. Es un http seguro con SSL.

3.7 Ejemplos de aplicaciones

A continuación se muestran alguna de las aplicaciones básicas por excelencia que se ven en todos los lenguajes. El famoso Hello World para la iniciación en la programación de NodeJS.

3.7.1 Servidor Hello World

Uno de los primeros ejemplos que se muestran en una gran cantidad de tutoriales es el que se muestra en la página oficial de NodeJS. Es un servidor web sencillo, escrito en NodeJS que responde con un *"Hello World"* para cada petición.

```
01: var http = require('http');
02: http.createServer(function (req, res) {
03:   res.writeHead(200, {'Content-Type': 'text/plain'});
04:   res.end('Hello World\n');
05: }).listen(1337, '127.0.0.1');

06: console.log('Server running at http://127.0.0.1:1337/');
```

Tabla 3 : Código Hello World de NodeJS [1]

Una vez escrito el código anterior en un fichero de texto, habrá que guardarlo con la extensión *.js*. Para ejecutarlo simplemente abrimos la consola de comandos y ejecutamos el comando `node` seguido del nombre del fichero, no es necesario escribir la extensión del fichero. Abriremos un navegador y escribimos la dirección IP en la que está escuchando el servidor http, y se mostrara el "Hello World".

En la página oficial también se presenta otro ejemplo utilizando el módulo net que responde a cada petición con un *echo*. Todos los ficheros *.js* se ejecutan de la misma manera.

```
01: var net = require('net');
02: var server = net.createServer(function (socket) {
03:   socket.write('Echo server\r\n');
04:   socket.pipe(socket);
05: });
06: server.listen(1337, '127.0.0.1');
```

Tabla 4 : Código usando NET de NodeJS [1]

3.8 Convenciones

Existen algunas convenciones para el desarrollo del código usando NodeJS así como una guía de estilo para la programación del código en Node [8].

3.8.1 Manejo de errores

En el desarrollo de aplicaciones con la plataforma de NodeJS no realizamos un manejo de errores como podemos ver en otros lenguajes con *try* y *catch*. Se utiliza la convención de, "el error primero" [9].

Esto es, cuando tenemos un *callback*, devolverá un objeto error que pondremos en primer lugar. Si existe el objeto error es que hemos tenido un error en la ejecución del código y será lo primero que manejamos lanzando una excepción del tipo "err".

Un ejemplo sencillo para visualizar esta convención sobre el manejo de los errores es este código que se muestra a continuación, donde podemos ver que lo primero que se realiza en la función es una condición de la existencia de un error.

```
01: var fs = require('fs');
02: fs.readFile('/etc/no.such.file',function(err,data){
03:   if(err) throw err;
04:   console.log(data.toString());
05: });
```

Tabla 5 : Código para visualizar el manejo de errores [9]

3.8.2 Callbacks

Del mismo modo que en la convención de errores, esta convención trata sobre el manejo de los callbacks. Está directamente relacionada con la convención de los errores, ya que esta trata sobre la utilización en último lugar del objeto de callback.

Podemos ver esta convención en el siguiente código. Donde no devuelve el objeto c hasta la última línea de código dentro de la función.

```
01: var net = require('net');
02: var server = net.createServer(function(c){
03:   c.write('hello world\r\n');
04:   c.pipe(c);
05: });
06: server.listen(8000,function(){
07:   console.log('server bound');
08: });
```

Tabla 6 : Código para visualizar la convención de los callback [9]

4 Sockets ØMQ

Para la implementación de los diferentes patrones de comunicación en el proyecto de replicación activa en Nodejs, se ha escogido la librería 0MQ. 0MQ fue originalmente desarrollada por *Pieter Hintjens* [10] en C++ y debido a sus características ha sido extendida a otros lenguajes como NodeJS.

Esta librería se ha escogido debido a la variedad de patrones que posee para el envío de mensajes, además, por ser útil en la programación asíncrona. Su uso está extendido entre los desarrolladores, lo que hace que sea más fácil encontrar colaboraciones sobre posibles aplicaciones y soluciones a problemas comunes.

4.1 Qué es ØMQ

También llamado ZeroMQ, ZMQ o 0MQ, es una librería para mensajería asíncrona de alto rendimiento que se usa en aplicaciones distribuidas, donde la escalabilidad es la mayor preocupación. 0MQ es multiplataforma y multilenguaje. Esta librería provee una cola de mensajes para el almacenamiento temporal de los mismos, pero, también funciona sin un intermediario que maneje los mensajes. La librería fue diseñada para tener un estilo similar a la API de los sockets tradicionales en cuanto a la recepción y envío de mensajes, pero es más clara el definir las conexiones entre los diferentes lugares [11].

4.2 Características

Como se ha expuesto en la introducción, la comunicación de 0MQ está basada en mensajes con una persistencia débil, es decir, con colas en memoria principal. 0MQ es sólo una librería donde no se necesita ningún servidor específico. Está desarrollada en C++ y está disponible para la mayoría de los sistemas operativos.

Esta librería proporciona al desarrollador los sockets para enviar y recibir mensajes con una interfaz sencilla similar a los sockets tradicionales utilizando *send y receive* para enviar y recibir los mensajes, y *bind* y *connect* para ofrecer un punto de conexión y conectarse.

Una de las características de los mensajes es que pueden ser multi-parte, el soporte para su estructuración es sencillo. Otra de las características es que todos los mensajes se entregan atómicamente, es decir, o se entrega el mensaje en su totalidad o no se entrega nada. Además, es importante recordar que tanto el envío de mensajes como su recepción son asíncronos.

4.3 Estructura de los mensajes

El manejo de los mensajes de 0MQ se hace mediante una estructura específica. El primer segmento de cada mensaje debe de ser un segmento vacío, es decir, no contener nada debido a que este segmento es el delimitador de mensajes. El socket del cliente adjunta este segmento en cada mensaje sin que la aplicación intervenga, lo realiza de manera automática. Por ejemplo, como en el caso de este proyecto donde se envían y reciben *strings*, el socket del programa que va a enviar el mensaje adjuntará este segmento vacío. El programa que reciba este mensaje identificara este segmento y sabrá entonces que ha recibido un nuevo mensaje.

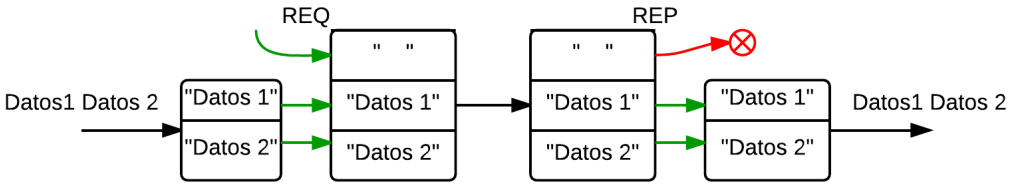


Ilustración 1 : Diagrama de la estructura de los mensajes en cada instante.

También se realiza de manera autónoma la eliminación, por parte del socket, del segmento, para que el contenido real del mensaje pueda ser utilizado por la aplicación en el programa que lo ha recibido.

4.4 Algunos métodos de los sockets 0MQ en NodeJS

Para la realización de los ejemplos que se van a mostrar más adelante, se hace uso de unos métodos útiles para conseguir lo deseado como son la conexión, la desconexión, crear escuchadores, enviar mensajes, etc.

- `var zmq=require('zmq')`: Escoger la librería de la que vamos a hacer uso.
- `var req=zmq.socket('req')`: Definir el tipo de socket, req, rep, pub, sub...
- `requester.connect("tcp://IP:Puerto")`: Conectarse a una dirección.
- `responder.bind('tcp://IP:Puerto, function(err){})`: Crear un punto de conexión.
- `requester.send(msg)`: Método para enviar mensajes.
- `requester.on("message", function(reply){})`: Método para crear un escuchador de mensajes.
- `requester.close()`: Forma correcta de cerrar los sockets.
- `requester.disconnect("tcp://IP:Puerto")`: Desconectar de la dirección especificada por parámetro.

4.5 Patrones básicos y ejemplos prácticos

0MQ ofrece al desarrollador diferentes patrones para el envío de mensajes como se ha comentado anteriormente. En este apartado se desarrollan los aspectos básicos de cada uno de ellos así como ejemplos prácticos de su uso en programas NodeJS.

4.5.1 Patrón Request/Reply

Con este patrón de comunicación el cliente envía un mensaje de petición a un sistema que procesa la petición y genera una respuesta. De esta forma se permite a dos aplicaciones tener una conversación de dos vías. Es un patrón muy común en arquitecturas típicas cliente/servidor.

Cada mensaje enviado vía *request* necesita asociarse a una contestación desde el socket *reply* del servidor. Todos los pares están totalmente ordenados y las conexiones pueden reaccionar asíncronamente. No obstante, su funcionamiento es más asíncrono de lo que puede parecer debido a su funcionamiento.

El funcionamiento del patrón comienza con una cola local del socket *request*, donde se van encolando cada una de las peticiones del cliente, y no se desencola la primera posición hasta que esta tenga la respuesta que le corresponde. Una vez la haya recibido, la segunda petición ocupará el primer lugar de la cola y se lanzará por el socket esperando su respuesta. El comportamiento del socket *reply* de respuestas funciona de forma simétrica al de *request*. Cuando se ha enviado un mensaje a través del socket, otro envío posterior será encolado localmente y las respuestas no se envían hasta que un nuevo mensaje de petición sea recibido.

4.5.1.1 Variantes

Este tipo de patrón se puede desarrollar con un único punto de peticiones conectándose con varios puntos de respuesta. Cuando esto ocurre el patrón sigue una política de *Round-Robin*[8], es decir, balanceando la carga de los puntos de respuesta. Este caso se da cuando un cliente que realiza peticiones se conecta con varios servidores. Con este escenario, cada petición del cliente se enviará a un servidor diferente siguiendo la política de *Round-Robin*. La operación sigue siendo sincrónica, es decir, 0MQ no envía nuevas peticiones hasta que cada respuesta para cada una de las anteriores sea recibida. No se da la paralelización de peticiones.

[8] Es un método para seleccionar todos los elementos en un grupo de manera equitativa y en un orden racional.

Para probar este caso solo hace falta utilizar los programas del ejemplo de uso del patrón *Req/Rep* [4.5.1.3]. Se han de crear tantos servidores nuevos como se quieran y en el cliente realizar las conexiones con cada uno de ellos. Las peticiones del cliente, si un servidor está ocupado, se redirigirán a otro.

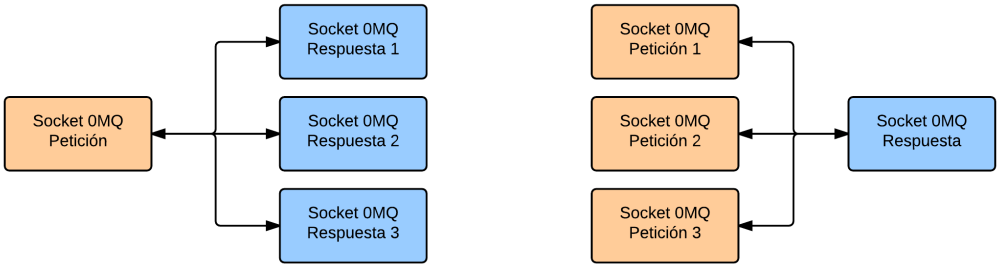


Ilustración 2 : Izq: Diagrama varios socket de respuesta. Der: Diagrama varios socket de petición.

Cuando existen múltiples peticionarios la forma como se tratan los mensajes en el servidor es un encolado equitativo, donde ninguno de los mensajes de petición sufrirá *inanición*[9].

4.5.1.2 Problemas respecto a fallos del cliente y/o el servidor

Este patrón de petición/respuesta es en cierto modo síncrono ya que una petición enviada por un cliente estará a la espera de la respuesta de un servidor.

El problema del fallo o caída del servidor es que el cliente se quedará esperando una respuesta que no puede llegar debido a que el servidor no está operativo. Si el servidor ha fallado antes de la conexión el cliente simplemente no podrá conectarse, pero si esto ocurre durante el intercambio de mensajes, el cliente no recibirá respuesta. En este caso se debería hacer uso de un *timeout* en el programa cliente, que una vez pasado un tiempo x sin recibir una respuesta a su petición realice alguna acción o bien para desconectar del servidor e informar al usuario o conectarse a otros servidores.

4.5.1.3 Ejemplo del patrón Req/Rep

Se ha realizado el desarrollo de dos programas para practicar el patrón básico de peticiones y respuestas programándolo en la plataforma NodeJS y así familiarizarnos con ambas partes. A partir de una versión básica de comunicación entre cliente y servidor con *Req/Rep* se realizan diferentes modificaciones y características que se presentan a continuación.

[9] Se refiere al problema en el ámbito de la programación en el cual un programa no recibe los recursos necesarios para su ejecución.

En el programa cliente se añadirá el uso de argumentos para que el usuario introduzca la *URL* del servidor, el número de peticiones a enviar y el texto de la petición. Mientras que en el programa servidor, al ser ejecutado, se pedirá por argumentos el puerto donde escucha el servidor, el número de segundos a esperar y el texto para enviar como respuesta de las peticiones.

Además, la respuesta que el servidor envía al cliente deberá ser la concatenación de la petición del cliente y el texto de respuesta propia del servidor. Finalmente si cualquiera de los dos programas no recibe el número correcto de argumentos deberá mostrar un mensaje que informe sobre que parámetros son necesarios y en qué orden han de ser introducidos y así concluir su ejecución. El código que se muestra a continuación muestra el uso del patrón *Req/Rep* en el programa cliente y la realización de algunas de las características.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var requester = zmq.socket('req');
05: var i = 0;
06: //Comprobacion de parametros
07: if( process.argv.length < 5 ) {
08:     console.log("Parametros incorrectos");
09:     console.log("Modo de ejecucion: node hwclient.js IP_SERVIDOR
                    PUERTO_SERVIDOR NUM_PETICIONES TXT_PETICION");
10:     process.exit(1);
11: }
12: //Captura de parametros
13: var ipServer = process.argv[2];
14: var puertoServer = process.argv[3];
15: var numPeticiones = process.argv[4];
16: var msg = process.argv[5];
17:
18: //Socket para la comunicacion
19: console.log("Conectando al server...");
20: requester.connect("tcp://" + ipServer + ":" + puertoServer);
21: //Envio de peticiones
22: for (var i = 0; i < numPeticiones; i++) {
23:     console.log("Enviando peticion", i, '...');
24:     requester.send(msg);
25: }
26:

27: //LISTENERS
28: requester.on("message", function(reply) {
29:     console.log("Reply recibido", i, ": [", reply.toString(),"]");
30:     i += 1;
31:     if (i == numPeticiones) {
32:             console.log("Desconecto");
33:             requester.close();
34:             process.exit(0);
35:     }
36: });
37: process.on('SIGINT', function() {
38:     requester.close();
39: });
```

Tabla 7 : Código programa cliente que muestra el uso del socket Req/Rep

Como se puede observar en el código mostrado, el programa cliente realiza un *require* de la librería de *zmq* para hacer uso del patrón deseado. Define una variable que asigna al tipo de socket requerido, en este caso un socket de tipo *req* o de petición y se conecta al servidor mediante el método *connect*. Realiza el envío de mensajes mediante el método *send* y espera las respuestas del servidor

mediante la creación de un escuchador con el método *on*. El desarrollo de las características requeridas en el programa cliente se realiza de la siguiente manera:

- Uso de argumentos para introducir la URL del servidor, el número de peticiones y el texto de las peticiones. Como podemos ver en el código del cliente, el programa asigna a unas variables el argumento específico.

```
var ipServer = process.argv[2];
var puertoServer = process.argv[3];
var numPeticiones = process.argv[4];
var msg = process.argv[5];
```

Tabla 8 : Código para la utilización de parámetros.

- Si no se introduce el número correcto de parámetros, mostrar cómo se deberá ejecutar el programa.

```
if( process.argv.length < 5 ) {
  console.log("Parametros incorrectos");
  console.log("Modo de ejecucion: node hwclient.js
              IP_SERVIDOR PUERTO_SERVIDOR NUM_PETICIONES TXT_PETICION");
  process.exit(1);
}
```

Tabla 9 : Código para mostrar al usuario la forma de ejecución.

El código del programa servidor que hace uso del patrón *Req/Rep* para responder las peticiones del cliente y la realización de algunas características en el mismo se muestra a continuación.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var responder = zmq.socket('rep');
05: //Comprobacion de parametros
06: if( process.argv.length < 4 ) {
07:     console.log("Parametros incorrectos");
08:     console.log("Modo de ejecucion: node hwserver.js PUERTO_SERVIDOR
                    NUM_SEGUNDOS TXT_RESPUESTA");
09:     process.exit(1);
10: }
11: //Captura de parametros
12: var port = process.argv[2];
13: var segundos = process.argv[3];
14: var msg = process.argv[4];
15:
16: //LISTENERS
17: responder.on('message', function(request) {
18:     console.log("Peticion recibida: [", request.toString(), "]");
19:     // do some 'work'
20:     setTimeout(function() {
21:     // send reply back to client.
22:             responder.send("Petición cliente: " + request.toString()
                                + " Respuesta servidor: " + msg);
23:     }, segundos*1000);
24: });
25:
26: responder.bind('tcp://*:' + port, function(err) {
27:     if (err) console.log(err);
28:     else console.log("Escuchando en el puerto "+port+"...");
29: });
30:
31: process.on('SIGINT', function() {
32:     responder.close();
33: });
```

Tabla 10 : Código programa servidor que muestra el uso del socket Req/Rep

El servidor al igual que el cliente requiere de la librería de *zmq* para usar el tipo de socket necesario y tras asignar una variable al tipo de socket que se necesita, en este caso el socket de tipo *rep*, se programa un *bind* con los parámetros introducidos de la dirección y puerto del servidor para ofrecer conexión a los clientes.

- Realización de un *bind* para ofrecer conexión a los clientes mediante dirección y puerto.

```
responder.bind('tcp://*:' + port, function(err) {
  if (err) console.log(err);
  else console.log("Escuchando en el puerto "+port+"...");
});
```

Tabla 11 : Código de bind en servidor

Como se puede observar en esta sección de código, la convención para comprobar los errores en primer lugar presentado en la sección de NodeJS, se cumple mediante el uso de la condición *if* y la comprobación de si el valor *err* existe. En tal caso se mostrará el error, de lo contrario no habrá error alguno y, en este caso el servidor, continuará su ejecución normalmente escuchando conexiones de clientes.

La ejecución de estos ejemplos dan como resultado las siguientes salidas. El programa servidor se queda a la escucha en el puerto 5555 y responderá con el texto de mensaje “adiós” cada 3 segundos una vez recibidas las peticiones. El cliente envía 3 peticiones a la dirección 127.0.0.1:5555 y recibirá la respuesta del servidor.

```
spora@esclavo1:~/Escritorio/0MQ final$ node hwserver.js 5555 3 adios
Escuchando en el puerto 5555...
Peticion recibida: [ hola ]
Peticion recibida: [ hola ]
Peticion recibida: [ hola ]
```

```
spora@esclavo1:~/Escritorio/0MQ final$ node hwclient.js 127.0.0.1 5555 3
Conectando al server...
Enviando peticion 0 ...
Enviando peticion 1 ...
Enviando peticion 2 ...
Reply recibido 3 : [ Petición cliente: hola Respuesta servidor: adios ]
Reply recibido 4 : [ Petición cliente: hola Respuesta servidor: adios ]
Reply recibido 5 : [ Petición cliente: hola Respuesta servidor: adios ]
```

Resultado 1 : Resultados en consola del ejemplo Req/Rep

4.5.2 Patrón Push/Pull

Este patrón de comunicaciones de 0MQ, llamado en español patrón de productor/consumidor, presenta una distribución unidireccional de los mensajes. El emisor envía un mensaje pero no espera ninguna respuesta a su envío, con lo que se puede pensar en un uso de envío de mensajes concurrentes.

Cuando se conectan varios sockets *Pull/Push* la carga del envío de mensaje se balancea de forma equitativa con *Round-Robin* y encolado equitativo.

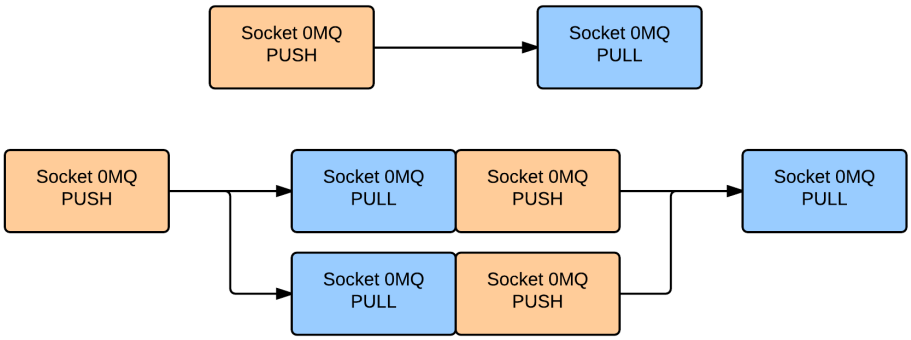


Ilustración 3 : Diagrama de diferentes modelos con Push/Pull

4.5.2.1 Ejemplo del patrón Push/Pull

El ejemplo desarrollado para aprender el uso de este tipo de sockets 0MQ se divide en dos programas, el productor y el consumidor. El cliente, que muestro a continuación es el que se conectará al punto del servidor para recibir los mensajes. El cliente, al ser un socket de tipo *Pull*, no tendrá la capacidad de enviar mensajes al servidor.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var pull = zmq.socket('pull');
05: var i = 0;
06:
07: if( process.argv.length < 3 ) {
08:     console.log("Parametros incorrectos");
09:     console.log("Modo de ejecucion: node pullpushclient.js
                    IP_SERVIDOR PUERTO_SERVIDOR ");
10:     process.exit(1);
11: }
12:
13: var ipServer = process.argv[2];
14: var puertoServer = process.argv[3]
15:
16: console.log("Conectando al server...");
17: pull.connect("tcp://" + ipServer + ":" + puertoServer);
18:
19: //LISTENERS
20: pull.on("message", function(msg) {
21:     console.log("Mensaje recibido: [", msg.toString(), "]");
22: });
23:
24: process.on('SIGINT', function() {
25:     requester.close();
26: });
```

Tabla 12 : Código programa consumidor que muestra el uso del socket Push/Pull

El programa servidor que se presenta a continuación crea un *bind*, del mismo modo que en el ejemplo del patrón Req/Rep, para que los clientes se conecten. Enviará los mensajes definidos por los parámetros a los clientes. Estos últimos no le responderán, por lo que no es necesario la creación de un escuchador de mensajes en este caso.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var push = zmq.socket('push');
05:
06: if( process.argv.length < 4 ) {
```

```
07:     console.log("Parametros incorrectos");
08:     console.log("Modo de ejecucion: node pullpushserver.js
                    PUERTO_SERVIDOR NUM_MENSAJES TEXTO");
09:     process.exit(1);
10: }
11:
12: var port = process.argv[2];
13: var repeticiones = process.argv[3];
14: var msg = process.argv[4];
15:
16:
17: push.bind('tcp://*:' + port, function(err) {
18:     if (err) console.log(err);
19:     else console.log("Escuchando en el puerto "+port+"...");
20: });
21:
22: for(i=0; i<repeticiones; i++) {
23:     push.send(msg);
24: };
25:
26: process.on('SIGINT', function() {
27:     push.close();
28: });
```

Tabla 13 : Código programa productor que muestra el uso del socket Push/Pull

La ejecución de estos ejemplos dán como resultado las siguientes salidas. El programa productor tiene un escuchador creado en el puerto 5555 y manda el mensaje al cliente cada 3 segundos. El cliente se conecta al servidor y recibe esos mensajes, no tiene la capacidad de mandar ningún mensaje al servidor.

```
spora@esclavo1:~/Escritorio/0MQ final$ node pushpullserver.js 5555 3 adios
Escuchando en el puerto 5555...

spora@esclavo1:~/Escritorio/0MQ final$ node pushpullclient.js 127.0.0.1 5555
Conectando al server...
Mensaje recibido: [ adios ]
Mensaje recibido: [ adios ]
Mensaje recibido: [ adios ]
```

Resultado 2 : Resultados en consola del ejemplo Push/Pull

4.5.3 Patrón Pub/Sub

Este patrón se forma a partir de la idea de la subscripción de los clientes a diferentes tipos de mensajes que pueda lanzar el servidor. Existe un servidor que envía los mensajes a todos los agentes disponibles y conectados.

Su funcionamiento comienza con la subscripción del cliente al publicador. Un publicador solo ofrece varios tipos de mensajes, de los cuales el cliente, tendrá que escoger a cuál de ellos subscribirse, ninguno, uno, varios o todos. Durante este proceso, el publicador puede haber empezado el envío de mensajes a sus subscriptores conectados por lo que si el cliente no está conectado a tiempo perderá esos mensajes.

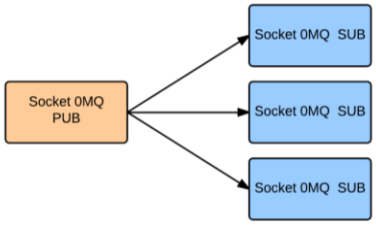


Ilustración 4 : Diagrama patrón publicador y subscriptores

4.5.3.1 Ejemplo del patrón Pub/Sub

En la realización de este ejemplo se desarrollan dos programas para aprender el uso del patrón Pub/Sub de los socket 0MQ.

A partir de una versión básica de comunicación con este patrón entre cliente y servidor se realizan diferentes modificaciones y añadidos. En el programa cliente o subscriptor se añade el uso de argumentos para que el usuario introduzca la URL del servidor o publicador y el descriptor del tipo de mensajes a los que se subscribe. Mientras que en el programa publicador se pide por argumentos el puerto, el número de mensajes a publicar y los descriptores de dos tipos de mensajes a publicar. Además, el publicador publicará el mismo número de mensajes de los dos tipos especificados y el envío al cliente debe ser la concatenación del descriptor de tipo de mensaje y un número aleatorio.

El código del programa cliente que se ha desarrollado como ejemplo de este patrón Pub/Sub es el que se muestra a continuación. Las características del ejemplo son similares a las de los ejemplos anteriores.

```
01: //REQUIRES
02: var zmq = require('zmq')
03: //VARIABLES
04: var subscriber = zmq.socket('sub')
05:
06: if( process.argv.length < 4 ) {
07:     console.log("Parametros incorrectos");
08:     console.log("Modo de ejecucion: node subscriber.js IP_SERVIDOR
09:                  PUERTO_SERVIDOR DESCRIPTOR");
10:     process.exit(1);
11: }
12:
13: var ipServer = process.argv[2];
14: var puertoServer = process.argv[3];
15: var descriptor = process.argv[4];
16:
17: subscriber.connect("tcp://" + ipServer + ":" + puertoServer);
18: subscriber.subscribe(descriptor);
19:
20: //LISTENERS
21: subscriber.on("message", function(msg){
22:     console.log("Recibido: "+msg);
23: });
24: process.on('SIGINT', function() {
25:   subscriber.close()
26:   console.log('\nClosed')
27: });
```

Tabla 14 : Código programa cliente que muestra el uso del socket Pub/Sub

En este caso el tipo de patrón es *Pub/Sub* y para el programa cliente habrá que tener una variable asignada al tipo sub de la librería de *zmq*. En lo que se diferencia este cliente de los anteriormente explicados es que este cliente se subscribe a uno o varios tipos específicos de mensajes que manda el servidor a todos los subscriptores conectados.

El código del servidor o publicador es el siguiente:

```
01: //REQUIRES
02: var zmq = require('zmq')
03: var mensaje = require("./randnum.js");
04: //VARIABLES
05: var publisher = zmq.socket('pub')
06:
07: if( process.argv.length < 5 ) {
08:   console.log("Parametros incorrectos");
09:   console.log("Modo de ejecucion: node publisher.js PUERTO_PUBLICADO
            NUM_MENSAJES DESCRIPTOR1 DESCRIPTOR2");
10:   process.exit(1);
11: }
12:
13: var puertoPub = process.argv[2];
14: var numMensajes = process.argv[3];
15: var descriptor1 = process.argv[4];
16: var descriptor2 = process.argv[5];
17:
18: publisher.bind('tcp://*:'+puertoPub, function(err) {
19:   if(err)
20:     console.log(err)
21:   else
22:     console.log("Listening on "+puertoPub+"...")
23: })
24:
25: for (var i=1 ; i<=numMensajes ; i++)
26:   setTimeout(function() {
27:     publisher.send(descriptor1+":"+mensaje.randNumber(100,1));
28:     console.log("Enviado Mensaje del descriptor " +descriptor1);
29:     publisher.send(descriptor2+":"+mensaje.randNumber(100,1));
30:     console.log("Enviado Mensaje del descriptor " +descriptor2);
31:
32:    }, 1000 * i)
33:
34: //LISTENERS
35: process.on('SIGINT', function() {
36:  publisher.close()
37:  console.log('\nClosed')
38: })
```

Tabla 15 : Código programa servidor que muestra el uso del socket Pub/Sub

El servidor o publicador desarrolla un *bind* para ofrecer servicio a los clientes o subscriptores y envía mensajes a los clientes conectados mediante el método *send* que se ha visto ya en los ejemplos anteriores. Una de las características es que se envían el mismo número de mensajes de los dos tipos de mensajes especificados a partir de los parámetros. Además, los mensajes que se envían son una concatenación del descriptor de tipo de mensaje y un número aleatorio.

```
for (var i=1 ; i<=numMensajes ; i++)
  setTimeout(function() {
  publisher.send(descriptor1+":"+mensaje.randNumber(100,1));
  console.log("Enviado Mensaje del descriptor " +descriptor1);
  publisher.send(descriptor2+":"+mensaje.randNumber(100,1));
  console.log("Enviado Mensaje del descriptor " +descriptor2);
}, 1000 * i)
```

Tabla 16 : Código número de mensajes y envío.

Para la obtención de los números aleatorios, en un rango determinado, puede usarse la función *randNumber*, importando el modulo cuyo código se muestra a continuación.

```
01: module.exports = {
02:  // *** getLoad function
03:  getLoad : function() {
04:    var fs = require('fs')
05:    , data = fs.readFileSync("/proc/loadavg") // version sincrona
06:    , tokens = data.toString().split(' ')
07:    , min1 = parseFloat(tokens[0])+0.01
08:    , min5 = parseFloat(tokens[1])+0.01
09:    , min15 = parseFloat(tokens[2])+0.01
10:    ,m
11:    = min1*10 + min5*2 + min15;
12:    return m;
13:  },
14:  // *** randNumber function
15:  randNumber : function(upper, extra) {
16:    var num = Math.abs(Math.round(Math.random() * upper));
17:    return num + (extra || 0);
18:  },
19:  // *** randTime function
20:  randTime : function(n) {
21:    return Math.abs(Math.round(Math.random() * n)) + 1;
22:  },
23:  // *** showArguments function
24:  showArguments : function(a) {
25:    for (var k in a)
26:    console.log('\tPart', k, ':', a[k].toString());
27:  }
28: }
```

Tabla 17 : Código del módulo para números aleatorios.

La ejecución de Pub/Sub da como resultado las siguientes salidas. El programa publicador sobre el puerto 5555 manda dos tipos de mensajes a todos los subscriptores conectados, mensaje "*hola*" y mensaje "*hello*". El cliente se conecta al servidor y se subscribe solamente al mensaje "*hello*". Recibirá entonces esos mensajes, no tiene la capacidad de mandar ningún mensaje al servidor.

```
spora@esclavo1:~/Escritorio/0MQ final$ node publisher.js 5555 4 hola hello
Listening on 5555...
Enviado Mensaje del descriptor hola
Enviado Mensaje del descriptor hello
Enviado Mensaje del descriptor hola
Enviado Mensaje del descriptor hello
Enviado Mensaje del descriptor hola
Enviado Mensaje del descriptor hello
Enviado Mensaje del descriptor hola
Enviado Mensaje del descriptor hello
```

```
spora@esclavo1:~/Escritorio/0MQ final$ node subscriber.js 127.0.0.1 5555 hello
Recibido: hello:99
Recibido: hello:76
Recibido: hello:64
Recibido: hello:26
```

Resultado 3 : Resultados en consola del ejemplo Pub/Sub

4.6 Patrones avanzados y ejemplos prácticos

4.6.1 Patrón Router/Dealer

Es un tipo de patrón que se coloca en el mismo agente, para interactuar con otros clientes o servidores actuando como conexión intermedia. Su propósito es general y por ello se puede conectar con los demás tipos. Es usado frecuentemente como socket de tipo *request* pero asincrónico. No llegándose a bloquear por fallos en los agentes.

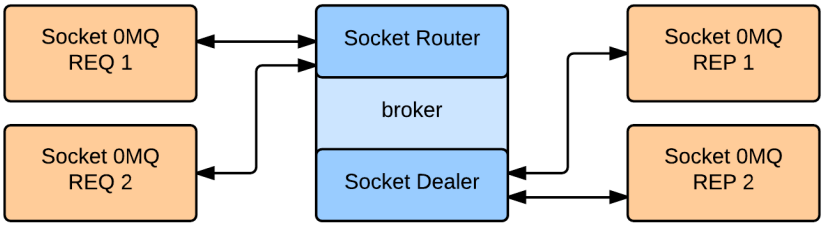


Ilustración 5 : Diagrama de flujo con el patrón router/dealer.

La parte *router* del patrón recibe los mensajes de manera asíncrona siendo capaz este de distinguir entre los agentes de envío para encaminar luego las respuestas. La parte del *dealer* es la que manda los mensajes a la parte final. Al realizar este patrón junto con Req/Rep deberemos de tener en cuenta que el proxy actuará de manera síncrona.

4.6.1.1 Ejemplo del patrón Router/Dealer

El ejemplo de este patrón consta de 3 programas en combinación con el patrón *Req/Rep* visto anteriormente que se utiliza para el envío de las peticiones por parte del cliente y el envío de las respuestas por parte del servidor. El programa intermedio es el llamado bróker o proxy. El código desarrollado de los tres programas se presenta a continuación.

El programa cliente hace uso del primer tipo de patrón visto en la sección de 0MQ, el patrón *Req/Rep*. El cliente es el que realiza las peticiones al servidor, pero para ello se debe conectar al programa intermedio, bróker o proxy.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var requester = zmq.socket('req');
05: var replyNbr = 0;
06:
07: if( process.argv.length < 5 ) {
08:     console.log("Parametros incorrectos");
09:     console.log("Modo de ejecucion: node rrclient.js IP_BROKER PUERTO_BROKER
                    NUM_PETICIONES TXT_PETICION");
10:     process.exit(1);
11: }
12:
13: var ipBroker = process.argv[2];
14: var puertoBroker = process.argv[3];
15: var numLoop = process.argv[4];
16: var msg = process.argv[5];
17:
18: requester.connect('tcp://' + ipBroker + ':' + puertoBroker);
19: requester.on('message', function(msg) {
20:     console.log('Respuesta recibida numero:', replyNbr, msg.toString());
21:     replyNbr += 1;
22:     if( replyNbr == numLoop ) {
23:             console.log("Desconecto");
24:             requester.close();
25:             process.exit(0);
26:     }
27: });
28: for (var i = 0; i < numLoop; ++i) {
29:     requester.send(msg);
30: }
```

Tabla 18 : Código de cliente en Router/Dealer

El programa que actúa como proxy simplemente es un programa que hace uso de los tipos de socket router y dealer. Ofrece dos puntos de conexión, uno para los clientes, y otro para el servidor. La función que realizan los dos escuchadores es la de reenviar el mensaje que ha llegado de un cliente o un servidor al contrario, es decir, si el cliente envía una petición, el escuchador del bróker enviará esa petición al servidor y viceversa.

Al trabajar con un cliente y un servidor que utilizan el patrón de 0MQ Req/Rep, deberemos sincronizar el bróker mediante el uso de un bind síncrono, podemos verlo en las líneas *14* y *15*. Un cliente se mantendrá a la espera de una respuesta después de haber realizado una petición.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var frontend = zmq.socket('router');
05: var backend = zmq.socket('dealer');
06:
07: if( process.argv.length < 3 ) {
08:     console.log("Parametros incorrectos");
09:     console.log("Modo de ejecucion: node rrbroker.js PUERTO_ROUTER
10:                       PUERTO_DEALER");
11:     process.exit(1);
12: }
13:
14: var puertoRouter = process.argv[2];
15: var puertoDealer = process.argv[3];
16:
17: frontend.bindSync('tcp://*:' + puertoRouter);
18: backend.bindSync('tcp://*:' + puertoDealer);
19:
20: console.log('Intercambiando mensajes...');
21: frontend.on('message', function() {
22:     // Note that separate message parts come as function arguments.
23:     var args = Array.apply(null, arguments);
24:     // Pass array of strings/buffers to send multipart messages.
25:     backend.send(args);
26: });
27: backend.on('message', function() {
28:     var args = Array.apply(null, arguments);
29:     frontend.send(args);
30: });
```

Tabla 19 : Código de broker en Router/Dealer

El programa que servidor tiene la función de responder a las peticiones de los clientes que le llegan a través del programa broker. En su código tiene el escuchador necesario para recibir las peticiones y devolverá al usuario a través del bróker otra vez la respuesta.

```
01: //REQUIRES
02: var zmq = require('zmq');
03: //VARIABLES
04: var responder = zmq.socket('rep');
05:
06: if( process.argv.length < 5 ) {
07:     console.log("Parametros incorrectos");
08:      console.log("Modo de ejecucion: node rrworker.js IP_BROKER
                    PUERTO_BROKER TIEMPO TXT_RESPUESTA");
09:      process.exit(1);
10: }
11:
12: var ipBroker = process.argv[2];
13: var puertoBroker = process.argv[3];
14: var time = process.argv[4];
15: var msgServer = process.argv[5];
16:
```

```
17: responder.connect('tcp://' + ipBroker + ':' + puertoBroker);
18:
19: responder.on('message', function(msg) {
20:     console.log('Peticion recibida:', msg.toString());
21:     setTimeout(function() {
22:             responder.send(msgServer);
23:     }, time);
24: });
```

Tabla 20 : Código de servidor en Router/Dealer

Para la realización de este ejemplo se han considerado algunas características que se presentan a continuación junto al código específico extraído del código mostrado anteriormente. Estas características son:

- En el programa cliente se terminará la ejecución del mismo, cuando reciba respuesta a todas sus peticiones.

```
requester.on('message', function(msg) {
  console.log('got reply', replyNbr, msg.toString());
  replyNbr += 1;
  if( replyNbr == numLoop ) {
    console.log("Desconecto");
    requester.close();
    process.exit(0);
  }
});
```

Tabla 21 : Código para que el cliente termine su ejecución.

- El programa servidor, no debe nunca terminar su ejecucion, siempre estará a la espera de nuevas peticiones.

```
responder.on('message', function(msg) {
  console.log('Peticion recibida:', msg.toString());
  setTimeout(function() {
    responder.send(msgServer);
  }, time);
});
```

Tabla 22 : Código para el escuchador del servidor.

- Y por último en el programa intermedio bróker se pedirá por argumentos los puertos que se ofrecerán a cliente y servidores para realizar este patrón.

```
var puertoRouter = process.argv[2];
var puertoDealer = process.argv[3];

frontend.bindSync('tcp://*:' + puertoRouter);
backend.bindSync('tcp://*:' + puertoDealer);
```

Tabla 23 : Código para la utilización de parámetros en el broker.

- Además, igual que con el servidor, el broker no debe nunca terminar su ejecución, siempre estará a la espera de nuevas peticiones.

```
frontend.on('message', function() {
  var args = Array.apply(null, arguments);
  backend.send(args);
});

backend.on('message', function() {
  var args = Array.apply(null, arguments);
  frontend.send(args);
});
```

Tabla 24: Código para los escuchadores en el broker.

La ejecución de los programas del ejemplo para comprobar el funcionamiento del patrón Router/Dealer dan como resultado las siguientes salidas. El programa bróker ofrece dos puntos de conexión síncronos sobre el puerto 5555 y el puerto 5556. Los dos clientes mandan sus mensajes que llegan al bróker y este los manda al worker o servidor. El servidor manda una respuesta a cada mensaje recibido y el bróker distribuye las respuestas correspondientes a cada cliente.

```
spora@esclavo1:~/Escritorio/0MQ final$ node rrbroker.js 5555 5556
Intercambiando mensajes...
```

```
spora@esclavo1:~/Escritorio/0MQ final$ node rrworker.js 127.0.0.1 5556 3 adios
Peticion recibida: holauser1
Peticion recibida: holauser2
Peticion recibida: holauser1
Peticion recibida: holauser2
```

```
spora@esclavo1:~/Escritorio/0MQ final$ node rrclient.js 127.0.0.1 5555 2 holauser1
Respuesta recibida numero: 0 -> mensaje: holauser1_adios
Respuesta recibida numero: 1 -> mensaje: holauser1_adios
Desconecto
```

```
spora@esclavo1:~/Escritorio/0MQ final$ node rrclient.js 127.0.0.1 5555 2 holauser2
Respuesta recibida numero: 0 -> mensaje: holauser2_adios
Respuesta recibida numero: 1 -> mensaje: holauser2_adios
Desconecto
```

Resultado 4 : Resultados en consola del ejemplo Router/Dealer

4.6.2 Ejemplo Proxy (Router/Router)

En este ejemplo se realiza un bróker, pero en este caso los *endpoint* serán ambos de tipo *Router*. A este bróker le llamamos Proxy. Una vez que este reciba las peticiones de clientes, seleccionará un trabajador y le pasará el mensaje para su procesamiento. Para realizar esta tarea el proxy conocerá las identidades de los clientes y de los trabajadores.

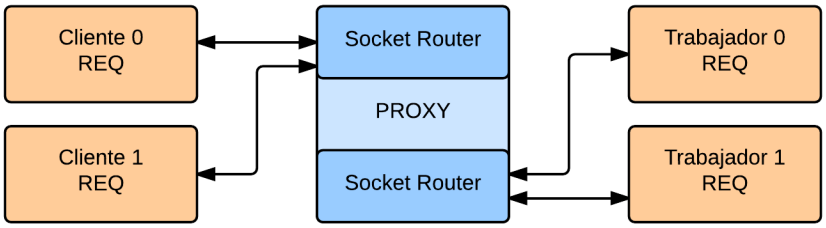


Ilustración 6 : Diagrama de flujo en el Proxy Router/Router

En primer lugar vamos a definir el programa cliente. Este se conectará al *frontend* del proxy, se puede usar el código del ejemplo anterior para el cliente como base. Los argumentos que se piden al usuario son la URL del *router*, un *string* para identificar al usuario y el texto de la petición. Al igual que en el ejemplo anterior el cliente usará el socket *req* para el envío de las peticiones y se quedará a la espera de las respuestas.

```
01: var zmq = require('zmq');
02: var requester = zmq.socket('req');
03: // ARGUMENTS
04: if( process.argv.length != 6 ) {
05:     console.log("Parametros incorrectos");
06:     process.exit(1);
07: }
08:
09: var ipBroker = process.argv[2];
10: var portBroker = process.argv[3];
11: var identityClient = process.argv[4];
12: var serviceRequest = process.argv[5];
13:
14: requester.identity = identityClient;
15: requester.connect('tcp://' + ipBroker + ':' + portBroker );
16:
17: console.log("Client ( " + identityClient + " ) connected to tcp://" + ipBroker
                + ":" + portBroker + " ...");
18: requester.on('message', function(msg) {
19:     console.log("Client ( " + identityClient + " ) has received reply: "
                    + msg.toString());
20:     requester.close();
21:     process.exit(0);
22: });
23:
24: console.log("Client ( " + identityClient + " ) has sent its msg: "
                + 25: serviceRequest);
25: requester.send(serviceRequest);
```

Tabla 25 : Código programa cliente Proxy

En segundo lugar tenemos el trabajador, que al igual que en el programa cliente podemos hacer uso del código del ejemplo anterior como base para su desarrollo. También, como en el cliente, debemos introducir ciertos parámetros. Habrá que darle al programa la URL del proxy al que nos conectamos, un *string* para identificar al trabajador, un mensaje de disponibilidad, otro de servicio y finalmente un valor booleano que indicará si se activa el modo *verbose*[10].

El trabajador también utiliza un socket *req*. Enviará una petición para darse de alta en el proxy, con ello informamos de la disponibilidad de este trabajador. Y enviará por cada petición atendida otra petición al proxy para que notifique al cliente que el servicio ha sido atendido.

```
01: var zmq = require('zmq');
02: var responder = zmq.socket('req');
03: var auxfunctions = require('./auxfunctions.js');
04:
05: if( process.argv.length != 7 ) {
06:     console.log("Parametros incorrectos");
07:     process.exit(1);
08: }
09:
10: var endpoint = process.argv[2];
11: var id = process.argv [3];
12: var disponibilidad = process.argv[4];
13: var atencion = process.argv[5];
14: var verbose = process.argv[6] == "S";
15: var num = 0;
16:
17: if( verbose ) {
18:     console.log('worker ( ' + id + ' ) connected to ' + endpoint);
19:     console.log('worker ( ' + id + ' ) has sent READY msg: ' + disponibilidad);
20: }
21:
22: responder.identity = id;
23: responder.connect('tcp://127.0.0.1:'+endpoint);
```

[10] Proporciona información detallada en la salida de la consola

```
24: responder.on('message', function() {
25:     var args = Array.apply(null, arguments);
26:     if( verbose ) {
27:             console.log("worker ( " + id + " ) has received request: ( "
                            + args[2] + " ) from client ( " + args[0] + " ).");
28:             auxfunctions.showArguments(args);
29:     }
30:     setTimeout(function() {
31:             if( verbose ) {
32:                     console.log("worker ( " + id + " ) has send its reply");
33:                     auxfunctions.showArguments([args[0], "", atencion ]);
34:                     console.log("worker ( " + id + " ) has sent " + (++num)
                                    + " replies");
35:             }
36:             responder.send([args[0], "", atencion ]);
37:     }, 1000);
38: });
39: responder.send(disponibilidad);
```

Tabla 26 : Código programa worker Proxy

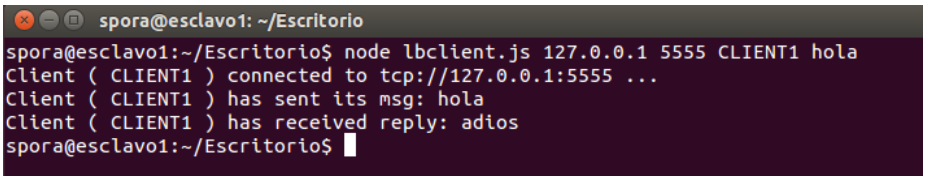
El proxy también debe de tener los parámetros de los puertos a los cuales los trabajadores y clientes pueden conectarse para recibir su servicio y un valor booleano para el modo *verbose*. Como se ha comentado antes, se usan dos socket de tipo Router para gestionar las comunicaciones. Cuando recibe las peticiones de los clientes comprobará si existen trabajadores conectados, en caso afirmativo se enviarán las peticiones a uno de los trabajadores para que responda a la petición. El trabajador que este ocupado tendrá el estado de no disponible.

```
01: var zmq = require('zmq');
02: var frontend = zmq.socket('router');
03: var backend = zmq.socket('router');
04: var auxfunctions = require('./auxfunctions.js');
05: var WORKING = true;
06: var workers = {};
07:
08: function getWorker() {
09:     var minWorks = calculeWorksmin();
10:     for (var key in workers) {
11:       if( workers[key][1] == minWorks )
12:             return key;
13:     }
14:     return null;
15: }
16: function calculeWorksmin() {
17:     var min = 999999999;
18:     for (var key in workers) {
19:       if( workers[key][1] < min )
20:             min = workers[key][1];
21:     }
22:     return min
23: }
24: function clearArgs(args) {
25:     var newArgs = args.reverse();
26:     newArgs.pop();
27:     newArgs.pop();
28:     return newArgs.reverse();
29: }
30:
31: // ARGUMENTS
32: if( process.argv.length != 5 ) {
33:     console.log("Parametros incorrectos");
34:     process.exit(1);
35: }
36: var portClient = process.argv[2];
37: var portWorker = process.argv[3];
38: var verbose = process.argv[4] == "S";
39:
40: // ARGUMENTS
41: if( verbose ){
42:     console.log('broker: frontend-router listening on tcp://*:'
                    + portClient + ' ...');
```

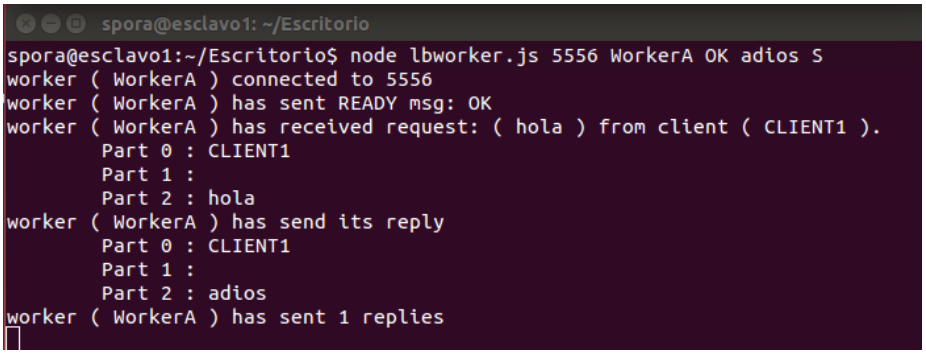
```
43:     console.log('broker: backend-router listening on tcp://*:'
                    + portWorker + ' ...');
44: }
45:
46: frontend.bindSync('tcp://*:' + portClient);
47: backend.bindSync('tcp://*:' + portWorker);
48: frontend.on('message', function() {
49:     var args = Array.apply(null, arguments);
50:     var worker = getWorker();
51:     if(verbose) {
52:             console.log("Received request: " + args[2] + " from client ( "
                            + args[0] + " ).");
53:             auxfunctions.showArguments(args);
54:     }
55:     if( worker == null ) {
56:             console.log("We have not workers")
57:             frontend.send([args[0],"",'We have not workers']);
58:             return
59:     }
60:     if(verbose){
61:             console.log("Sending client: ( " + args[2] + " ) req to worker ( "
                                                  + worker + " ) through bakend.");
62:             auxfunctions.showArguments(args);
63:     }
64:     workers[worker][0] = WORKING;
65:     workers[worker][1] += 1;
66:     backend.send([worker,"",args]);
67: });
68:
69: backend.on('message', function() {
70:     var args = Array.apply(null, arguments);
71:
72:     if(workers[args[0]] == undefined) {
73:
74:             workers[args[0]] = [!WORKING, 0];
75:             if(verbose) {
76:                     console.log("Received request: ( " + args[2]
                                    + " ) from worker ( " + args[0] + " ).");
77:                     auxfunctions.showArguments(args);
78:             }
79:     }
80:     else {
81:             workers[args[0]][0] = !WORKING;
82:             if(verbose) {
83:                     console.log("Received request: ( " + args[2]
                                    + " ) from worker ( " + args[0] + " ).");
84:                     auxfunctions.showArguments(args);
85:             }
86:     }
87:
88:     if(args[2] != "READY") {
89:             console.log("Sending worker: ( " + args[0] + " ) rep to client ( "
                            + args[2] + " ) through frontend.");
90:             args = clearArgs(args);
91:             auxfunctions.showArguments(args);
92:             frontend.send([args[0], "",args[2]]);
93:     }
94:
95:     console.log(workers);
96: });
```

Tabla 27 : Código programa bróker Proxy

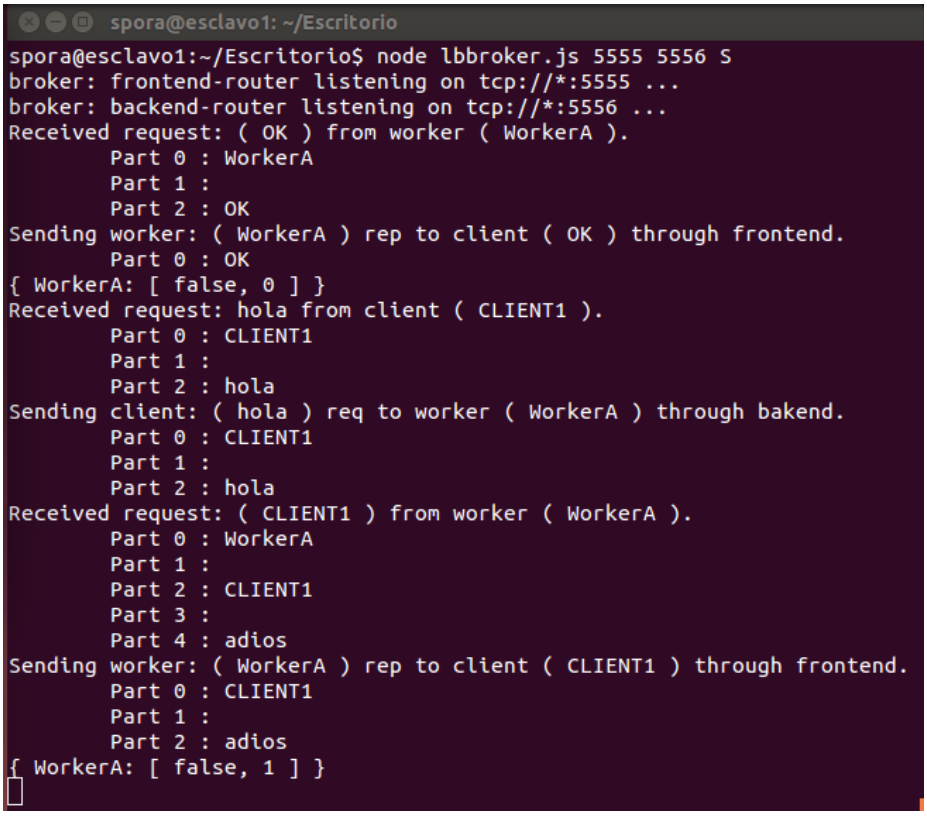
El código del módulo utilizado para ciertas funciones se encuentra en los anexos al final de esta memoria 8.7. Los resultados de la ejecución de estos programas los podemos ver a continuación:



Resultado 5 : Ejecución y resultados del cliente en el ejemplo Proxy



Resultado 6 : Ejecución y resultados del cliente en el ejemplo Proxy



Resultado 7 : Ejecución y resultados del cliente en el ejemplo Proxy

4.7 Ejemplo práctico con promesas

Haciendo uso de promesas en los programas NodeJS junto con los patrones de 0MQ, en este caso el patrón Req/Rep, vemos el funcionamiento de las mismas, a través de este código que se muestra a continuación.

Se crean dos funciones, *onSucess* y *onError*. La primera devolverá por pantalla la respuesta de un servidor, la segunda en cambio el error acontecido. El string "*hello*" se envía cada segundo y mediante el método *then* de la promesa llamaremos a una de las dos funciones dependiendo de cuál sea la que recibimos.

```
01: // Cliente
02: var zmq = require('zmq');
03: var rq = zmq.socket('req');
04: var counter=0;
05: function onSuccess(msg) {
06:     console.log('Response: '+msg);
07: }
08:
09: function onError(err) {
10:     console.log('Error: '+err);
11: }
12: rq.connect('tcp://127.0.0.1:8888');
13:
14: setInterval( function() {
15:   reply = rq.request('Hello ', counter++);
16:   reply.then(onSuccess,onError);
17: }, 1000 );
```

Tabla 28 : Código cliente con promesas y Req/Rep

El servidor de este ejemplo no contiene promesa alguna, una vez recibidas las peticiones de los clientes mandará su respuesta si el contador de mensajes no ha superado el número 10.

```
01: // Servidor
02: var zmq = require('zmq');
03: var rp = zmq.socket('rep');
04: var counter = 0;
05: rp.bind('tcp://127.0.0.1:8888',function(err) {
06:   if (err) throw err;
07: });
08: rp.on('message', function(seg1, seg2) {
09:   console.log('Request: ' + seg1 + seg2);
10:   if (++counter==10)
11:       process.exit();
12:   rp.send('World');
13: });
```

Tabla 29 : Código cliente con promesas y Req/Rep

Se hace uso del siguiente módulo que utiliza la librería *Q* para que nuestro programa cliente pueda funcionar con promesas. Si comprueba que el socket utilizado es de tipo *req* es entonces cuando se añade el método *request* que usaremos en el cliente.

```
01: var zmq = require('zmq');
02: var Q = require('q');
03:
04: var socket = zmq.socket;
05:
06: // Crear el metodo
07: zmq.socket = function(tipo) {
08:
09:     var so = socket.call(zmq, tipo);
10:     if (tipo == 'req') {
11:         // Si es un socket tipo 'req'
12:         so.request = request;
13:     }
14:     return so;
15: };
```

```
16:
17: function request() {
18:     var d = Q.defer();
19:     var that = this; // save the reference to the object
20:     // Define the handler for the 'message' event of the underlying zmq socket
21:     var onResponse = function () {
22:       // we produce an array out of the argument list
23:       d.resolve(Array.prototype.slice.call(arguments));
24:       that.removeListener('error', onError);
25:       // do not forget to remove the handler for the error of this invocation
26:     };
27: // Define the handler for the 'error' event of the underlying zmq socket
28:     var onError = function (e) {
29:         d.reject(e);
30:         that.removeListener('message', onResponse);
31: // do not forget to remove handler for the 'message' event for this invocation
32:   };
33:   this.once('message', onResponse); // set up the handler for only one message
34:   this.once('error', onError);      // set up the handler for only one
35:   this.send(Array.prototype.slice.call(arguments));
36:   return d.promise;
37: }
38: // exports the zmq object we got (with the extra mods)
39: module.exports = zmq;
```

Tabla 30 : Código modulo que incorpora un método y el uso de promesas Q

4.8 Crítica al uso de 0MQ

ZeroMQ no está libre de defectos, sus creadores han intentado rectificar muchos de los fallos a partir de sucesores como pueden ser *Crossroads I/O* y *nanomsg*. *Crossroads I/O* ya ha desaparecido, lo que se va a presentar a continuación es sobre *nanomsg* [12], la cual ha sido reescrita completamente en C. *Nanomsg* es una librería que se basa en las características de rendimiento de ZeroMQ mientras que provee de mejoras vitales tanto de forma interna como externa. Además, *nanomsg* intenta corregir algunos de los comportamientos extraños que de vez en cuando se observan en 0MQ.

Una queja común que la gente tiene con la librería 0MQ es el hecho de que esta no provee una API para nuevos protocolos de transporte. Lo que *nanomsg* realiza para suplir este defecto es ofrecer una interfaz de conexiones para el transporte y los protocolos de mensajes, lo que significa el soporte a nuevos transportes y nuevos patrones de mensajería más allá de los estándares *Pub/Sub*, *Req/Rep*, etc.

Uno de los defectos más fundamentales de 0MQ se presenta en su arquitectura, sus hilos no son seguros. Existe una relación uno a uno entre objetos e hilos, este patrón trabaja correctamente para objetos manejados por hilos de servidores, pero no cuando lo hacen en hilos de usuario. En nanomsg, en vez de basarse en el paso de mensajes, las interacciones son modeladas como conjuntos de estados. Esto hace que sus sockets tengan hilos seguros. Además, nanomsg tiene varias optimizaciones internas con el objetivo de mejorar la eficiencia de la memoria y de la CPU.

4.8.1 Protocolos de escalabilidad de nanomsg

Estos protocolos de escalabilidad son patrones de comunicación que son una abstracción de la capa de transporte. Nanomsg define 6 tipos de protocolos que se presentan a continuación.

- **Protocolos ReqRep y PubSub**

  Estos dos protocolos de nanomsg son los mismos presentados antes de 0MQ. Req/Rep se basa en el envío de peticiones y respuestas y Pub/Sub permite a los clientes subscribirse a diferentes tipos de mensajes y el servidor envía a todos los subscriptores conectados.

- **Protocolo Pair**

  Pair, o comunicación bidireccional, es una comunicación uno a uno entre dos puntos o nodos. Cada uno de los puntos puede enviar y recibir mensajes.

- **Protocolo Pipeline**

  Pipeline ofrece flujo de datos de una sola dirección. Es muy útil para crear balanceadores de carga. El patrón que se ha presentado en 0MQ que es lo mismo que este, es el patrón de *Push/Pull*.

- **Protocolo Bus**

  El protocolo Bus permite enviar mensajes desde cada punto a todos los demás puntos del grupo.

- **Protocolo Survey**

  El protocolo de Survey es similar al patrón visto en 0MQ llamado Pub/Sub que también tiene su homónimo en nanomsg, donde un mensaje de un nodo es mandado a todo el grupo de subscriptores conectados. Donde se diferencia es en que cada *Respondent* de este patrón puede responder al mensaje recibido por el nodo emisor o *Surveyor.*

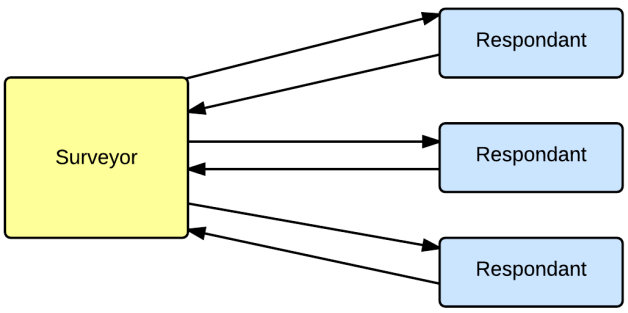


Ilustración 7 : Diagrama Protocolo Surveyor/Respondent

4.8.2 Conclusiones

Aunque se han visto las mejoras de nanomsg respecto de 0MQ hay que apuntar que la vida de 0MQ comenzó antes que la de nanomsg y esto hace que a día de hoy, trabajar con 0MQ sea, simplemente, más sencillo. Más sencillo debido a la gran cantidad de recursos e información de todo tipo respecto al uso de esta librería, así como la gran cantidad de desarrolladores que la han utilizado y siguen utilizándola, lo cual deja a nanomsg aun con sus mejoras, en desventaja.

5 Replicación Activa

5.1 Introducción

Los esfuerzos para construir sistemas distribuidos tolerantes a fallos empezaron hace tiempo. Uno de los problemas mayores que viene con los sistemas distribuidos es su potencial vulnerabilidad a los fallos. En realidad, siempre que cuando un nodo falla o se rompe, la disponibilidad de todo el sistema puede verse comprometida. Sin embargo la naturaleza distribuida de estos sistemas provee de la forma para aumentar la confiabilidad del sistema. Al tener un sistema distribuido, es posible introducir redundancia y con ello hacer el conjunto del sistema más seguro que cada uno de las partes individuales.

5.1.1 Activa V.S. Pasiva

En la técnica de replicación activa, cada réplica maneja las peticiones recibidas del cliente y envían una respuesta. Esta técnica hace que cada réplica actúe independientemente a las demás y consiste en asegurar que todas las réplicas reciben las peticiones en orden correcto. Esta técnica es apreciada ya que el tiempo de respuesta es bajo aun teniendo fallos, pero aun así tiene dos importantes defectos. El primero de ellos es que la redundancia implica un alto uso de recursos y la segunda derivada de la primera es que las peticiones deben ser manejadas de una manera determinista [13].

En la técnica de replicación pasiva en cambio una de las réplicas llamada réplica primaria actúa como principal, recibe las peticiones y devuelve las respuestas. Las réplicas de copias interactúan con la réplica primaria y reciben mensajes para actualizarse de la misma. Esta técnica es muy útil ya que utiliza una menor potencia de procesado que la replicación activa. A pesar de ello, la implementación de la replicación pasiva requiere de un mecanismo para escoger una réplica primaria, si esta falla una de las demás réplicas se coloca como primaria. En caso de fallo el tiempo de respuesta aumenta considerablemente tal que hace a esta técnica inadecuada para aplicaciones donde el tiempo de respuesta es crítico.

5.2 Replicación activa en tres niveles

La replicación es una técnica clásica usada para mejorar la disponibilidad de un servicio software. Las arquitecturas para implementar software de replicación con garantías de alta consistencia se desarrollan en dos niveles. Los clientes envían sus peticiones al nivel de la réplica, que se asegura que todas las réplicas están en un estado consistente antes de devolver la solución al cliente [14], [15].

El uso de complejos protocolos de acuerdo, tanto en la parte de cliente como en el de las réplicas, pueden llevar al sistema a tener momentos de indisponibilidad para con los clientes debido a la gestión de la replicación aun cuando el servicio, en realidad, sea correcto.

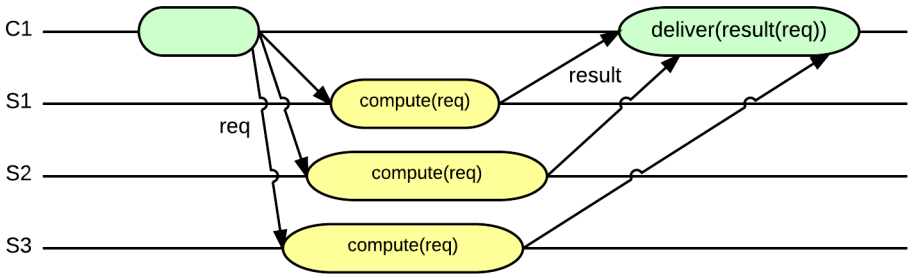


Ilustración 8: Técnica de la Replicación Activa

El protocolo de tres niveles se propone como solución para aliviar este problema de indisponibilidad. Esta arquitectura en tres niveles facilita la separación de la lógica de la replicación y la lógica de negocio.

Por último, cabe remarcar, que la replicación de tres niveles tiene una interacción rápida en comparación con la de dos niveles. Esto es debido a que la replicación de tres niveles solo necesita respuesta rápida de una sola réplica mientras que la replicación en dos niveles requiere que la mayoría de las réplicas respondan rápidamente.

Este modelo de replicación activa de tres niveles hace uso de un nivel intermedio, el cual se sitúa entre los clientes asíncronos o nivel de cliente y los servidores de réplicas asíncronos o nivel final. Este nivel es el encargado de la interacción con el cliente, de aceptar sus peticiones forzando un orden total así como el envío de estas peticiones al nivel final que a su vez devolverá una respuesta al nivel intermedio que seguidamente devolverá ese resultado al nivel de cliente.

En esta sección se explica el modelo del sistema y los tres niveles de la arquitectura para la replicación activa.

5.3 Modelo del sistema

Consideramos que el sistema distribuido está formado por un conjunto de procesos que se comunican mediante el envío de mensajes. Los procesos pueden fallar y después de este fallo el proceso no realizará ninguna acción por ello se dice que si un proceso nunca tiene fallos es un proceso correcto. Las propiedades que modelan la comunicación a través de canales asíncronos confiables son:

1) ***Validez del canal:*** Si el proceso recibe un mensaje m, entonces m ha sido enviado por algún proceso.

2) ***No duplicación de canal:*** Los mensajes se entregan a los procesos como mucho una vez.

3) ***Terminación del canal:*** Si un proceso correcto envía un mensaje m a otro proceso correcto, este último finalmente entrega m.

Los procesos son, en realidad, de tres tipos de conjuntos disjuntos, el primero de los conjuntos es el de procesos de cliente $C = \{c_1,...,c_l\}$ que se presenta en el nivel de cliente, el conjunto de manejadores de réplicas de replicación activa $H = \{h_1,...,h_n\}$ y el conjunto de réplicas de servidor deterministas en el nivel final $R = \{r_1,...,r_m\}$.

Se asume finalmente que:

1) ***Exactitud del ARH:*** La mayoría de las réplicas ARH son correctas.

2) ***Exactitud de la réplica:*** Existe al menos una réplica correcta.

5.4 Arquitectura en 3 niveles

En la ilustración observamos los diferentes componentes de la arquitectura de tres niveles para la replicación activa. El sistema asíncrono y distribuido está constituido por todos los componentes del diagrama menos el componente llamado Seq, el cual es un sistema parcialmente síncrono que se encarga de ofrecer un número único de una secuencia para cada petición.

A continuación se desarrolla una breve explicación de cada componente de la arquitectura. De la retransmisión/redirección, del filtrado y ordenación, del componente para la secuencia, y del ARH o manejador de la replicación activa.

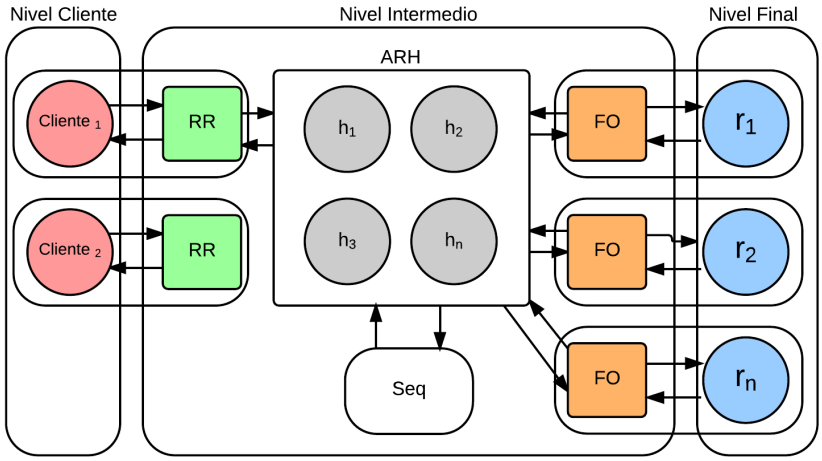


Ilustración 9 : Diagrama arquitectura en tres niveles

5.4.1 Retransmisión/Redirección (RR)

Cada proceso cliente lleva un manejador de mensajes RR, que expone a la aplicación cliente, el método para aceptar peticiones como parámetros introducidos a la aplicación. Este método es invocado por la aplicación y se queda en espera hasta que recibe el resultado de su petición para devolvérsela al cliente.

Operacionalmente, el manejador RR etiqueta a cada petición de cliente con un identificador de petición único y establece un tiempo de espera para transmitir la petición a una réplica ARH diferente si la respuesta no es recibida después de que este tiempo expire. Seguirá de este modo hasta que la respuesta sea devuelta al cliente. El pseudocódigo que se nos presentan en los artículos para este componente es el siguiente:

```
RR MESSAGE HANDLER
 1  hlist := ⟨h_1, ..., h_n⟩;
 2  INTEGER #cl_seq := 0;
 3  IDENTIFIER req_id;
 4  RESULT ISSUEREQ(request)
 5    begin
 6      INTEGER i := 1;
 7      #cl_seq := #cl_seq + 1;
 8      req_id := ⟨cl_id, #cl_seq⟩;
 9      loop
10        send ["Request", ⟨req_id, request⟩] to hlist(i);
11        t.setTimeout := period;
12        wait until ((deliver ["Result", ⟨req_id, result⟩] from h ∈ hlist) or (t.expired()))
13        if (not t.expired())
14          then return (result);
15          else  i := (i + 1) mod |hlist|;
16      end loop
17    end
```

Tabla 31 : Pseudocódigo del manejador RR presentado en [15]

5.4.2 Componente Secuenciador (DSS)

El servicio de este componente está disponible para cada réplica y es el que devuelve un número de secuencia único y consecutivo para cada petición de cliente distinta. Este servicio es capaz de devolver una petición si existe y está asociada a un número de secuencia dado. Esto contribuye a la propiedad de la terminación a pesar de que la réplica no funcione. Este componente tiene dos métodos:

- GetSeq(req_id) que devuelve el número de secuencia asociado con el identificador de petición como parámetro de entrada.
- GetReq(seq) que devuelve el identificador de la petición asociada a seq si existe o devuelve nulo en caso contrario.

Una correcta implementación del servicio del secuenciador debe satisfacer las siguientes propiedades:

1) Terminación: Si una réplica hi es correcta, entonces GetSeq y GetReq devuelven eventualmente una variable.

2) Acuerdo: $\forall(GetSeq(req) = v, GetSeq(req') = v'), req = req' \Rightarrow v = v'$

3) Unicidad: $\forall(GetSeq(req) = v, GetSeq(req') = v'), v = v' \Rightarrow req = req'$

4) Consecutividad: $\forall\, GetSeq(req) = v,\ (v \geq 1) \wedge (v > 1 \Rightarrow \exists\, req'\ s.t.\ GetSeq(req') = v - 1)$

La propiedad de acuerdo garantiza que dos réplicas ARH no pueden obtener distintos números de secuencia para la misma petición de cliente. La propiedad de unicidad previene que dos réplicas ARH obtengan el mismo número de secuencia para dos peticiones de cliente distintas. Finalmente la propiedad de la consecutividad garantiza que las réplicas ARH que llamen al método GetSeq obtendrán números enteros positivos consecutivos.

Además de estas propiedades, al llamar al método GetReq, las réplicas ARH deben garantizar las siguientes propiedades de seguridad:

5) Integridad de lectura: $\forall\, GetReq(seq) = v\ \Rightarrow ((v = null) \vee (v = req\ s.t.\ GetSeq(v) = seq))$

6) Validez de lectura: $\forall\, GetReq(req) = v\ \Rightarrow\ GetReq(v - k) = v',\ 0 \leq k < v,\ v' \neq null$

El pseudocódigo que se nos presentan en los artículos para este componente es el siguiente:

```
CLASS DSS
 1  ARRAY Sequenced := [null, null ...];
 2  INTEGER #LocalSeq := 1;
 3  REQUEST GETSEQ(req)
 4    begin
 5      if (∄#seq : Sequenced[#seq].id = req.id)
 6        then TOCast(req);
 7      wait until (∃#seq : Sequenced[#seq].req_id = req_id);
 8      return (#seq);
 9    end
10  REQUEST GETREQ(j)
11    begin
12      return (Sequenced[j]);
13    end
14  when (TODeliver(req, hε)) do
15    if (∄#seq : Sequenced[#seq] = req)
16      then Sequenced[#LocalSeq] := req;
17           #LocalSeq := #LocalSeq + 1;
```

Tabla 32 : Pseudocódigo de la clase DSS presentado en [14]

5.4.3 Manejador de Replicación Activa (ARH)

Este componente es el núcleo de la lógica de replicación. Haciendo uso del componente secuenciador, ordena todas las peticiones entrantes de los clientes y asegura el envío de una copia de la petición de cada cliente a cada réplica correcta. Una vez que las réplicas devuelven el resultado, el componente ARH lo devuelve a su vez a los clientes. Para conseguir la terminación a pesar de los fallos el componente se implementa con un conjunto de réplicas donde cada una de ellas mantiene un estado interno compuesto por pares formados por el identificador de la petición y la petición misma. El pseudocódigo que se nos presentan en los artículos para este componente es el siguiente:

```
ARH
1  INTEGER LastServedReq := 0;
2  INTEGER #seq;
3  Sequencer := new DSS();
4  when (deliver ["Request", req] from c) do
5    #seq := Sequencer.GETSEQ(req);
6    if (#seq > LastServedReq + 1)
7      then for each j : LastServedReq < j < #seq
8             do req_j := Sequencer.GETREQ(j);
9             for each  r_ℓ ∈ {r_1, ..., r_m} do send ["TORequest", ⟨j, req_j.op⟩] to r_ℓ;
10   for each  r_ℓ ∈ {r_1, ..., r_m} do send ["TORequest", ⟨#seq, req.op⟩] to r_ℓ;
11   LastServedReq := max(LastServedReq, #seq);
12   wait until (deliver ["TOReply", ⟨#seq, res⟩] from r_k ∈ {r_1, ..., r_m})
13   send ["Reply", ⟨req.id, res⟩] to c
```

Tabla 33 : Pseudocódigo del componente ARH presentado en [14]

5.4.4 Filtrado y Ordenación (FO)

Es un manejador de mensajes colocado en cada réplica para evitar ejecuciones repetidas de la misma petición de cliente y para asegurar una ejecución ordenada de las peticiones teniendo en cuenta el número de Secuencia de cada petición. Si sucede algún fallo en el cliente o en la réplica también fallarán sus manejadores de mensajes RR o FO respectivamente. El pseudocódigo que se nos presentan en los artículos para este componente es el siguiente:

```
FO MESSAGE HANDLER
1  ARRAY Executed;
2  INTEGER ExpectedSeq := 1;
3  when (deliver ["TORequest", ⟨seq, request⟩] from h_i) do
4    wait until (seq ≤ ExpectedSeq)
5    if (seq = ExpectedSeq)
6      then result := compute(request);
7           Executed[seq] := result;
8           ExpectedSeq := ExpectedSeq + 1;
9    send ["TOResult", ⟨seq, Executed[seq]⟩] to h_i;
```

Tabla 34 : Pseudocódigo del manejador FO presentado en [15]

5.5 Diseño e Implementación

Se ha optado por realizar dos programas cliente, sus correspondientes programas RR los cuales sirven para aceptar las peticiones, tres programas para los componentes ARH y sus correspondientes módulos DSS, además de un programa TO y tres componentes FO que realiza las llamadas a la función *compute* para realizar las operaciones que se consideren oportunas.

En la imagen siguiente se puede observar una aproximación general del proyecto realizado donde los componentes internos se han desarrollado usando el pseudocódigo de los artículos antes referenciados. En cambio, el programa para los clientes, el programa para el orden total TO y el negocio, no se han realizado mediante ningún pseudocódigo.

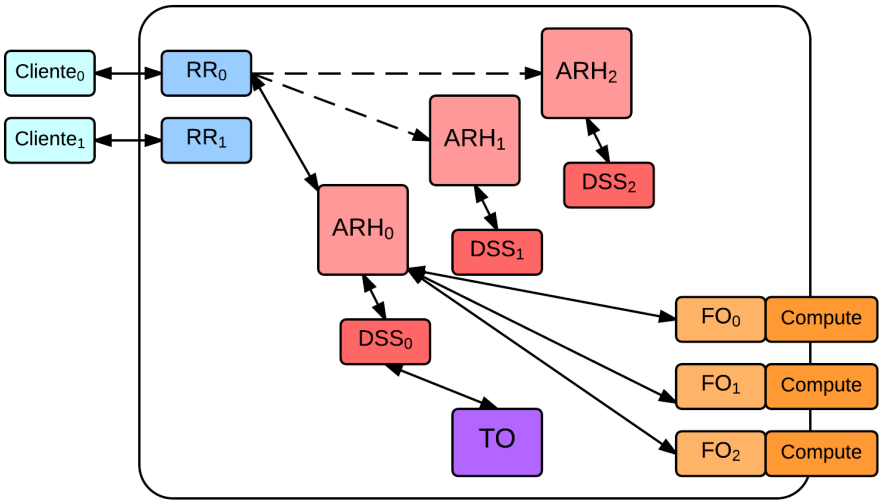


Ilustración 10 : Diseño de replicación activa del proyecto

La única diferencia, es la elección del negocio de empresa, que se ubicaría en el tercer nivel de esta replicación de tres niveles. En este caso, hemos optado por ofrecer a los usuarios del proyecto la capacidad de realizar las operaciones básicas sobre arrays mediante la función *Compute*. Las operaciones que se ofrecen son:

a) `push(x)` añade el elemento x al final del array.
b) `pop( )` elimina el último elemento del array.
c) `shift( )` elimina el primer elemento del array.
d) `unshift(x)` añade el elemento x al principio del array.
e) `indexOf(x)` busca el elemento x en el array.

5.5.1 Diseño de la red

El uso de diferentes tipos de sockets para la comunicación con los patrones de 0MQ hace que se creen muchos puntos de conexión que se detallan a continuación. Los programas Cliente DSS y FO son conectores de las direcciones de RR ARH y TO que son los programas que ofrecen la conexión. Las conexiones están en modo local.

RR0 → `Req/Rep; IP: 127.0.0.1:9010`

RR1 → `Req/Rep; IP: 127.0.0.1:9011`

ARH0 → `Req/Rep; IP: 127.0.0.1:9020`
`Pub/Sub; IP: 127.0.0.1:9021`
`Pull/Push; IP: 127.0.0.1:9022`

ARH2 → `Req/Rep; IP: 127.0.0.1:9026`
`Pub/Sub; IP: 127.0.0.1:9027`
`Pull/Push; IP: 127.0.0.1:9028`

ARH1 → `Req/Rep; IP: 127.0.0.1:9023`
`Pub/Sub; IP: 127.0.0.1:9024`
`Pull/Push; IP: 127.0.0.1:9025`

TO → `Pull/Push; IP: 127.0.0.1:9050`
`Pub/Sub; IP: 127.0.0.1:9051`

Para realizar pruebas distribuidas de la replicación activa, pero en red local, hay que cambiar las direcciones IP en los programas que se deseen ejecutar en otra máquina. En la red interna del laboratorio se comprueba las direcciones de los ordenadores a utilizar y veremos que su dirección es del tipo 1.1.1.x donde x es el número del puesto del ordenador.

5.5.2 Cliente

El programa cliente (Anexo 8.1), se encarga de pedir por teclado al cliente la orden que este quiera realizar. Esto se hace mediante una función llamada *NuevaPeticion,* en la que comprobamos si la entrada es admitida por nuestro sistema. Si no es válida, devolveremos el listado de las acciones disponibles y ofrecemos de nuevo la introducción de una nueva petición, si introduce el *string exit* terminará su ejecución. Si la petición es válida, creamos un objeto JSON llamado *envioCliente_RR.* Para cada cliente el programa genera un s*tring* aleatorio en cada ejecución, esta identificación de cliente se introduce en el objeto antes mencionado así como la petición del cliente. Una vez hecho esto, generamos un *string* a partir del objeto JSON y enviamos al programa RR ese *string.*

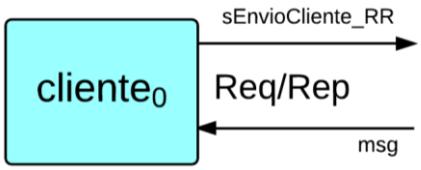
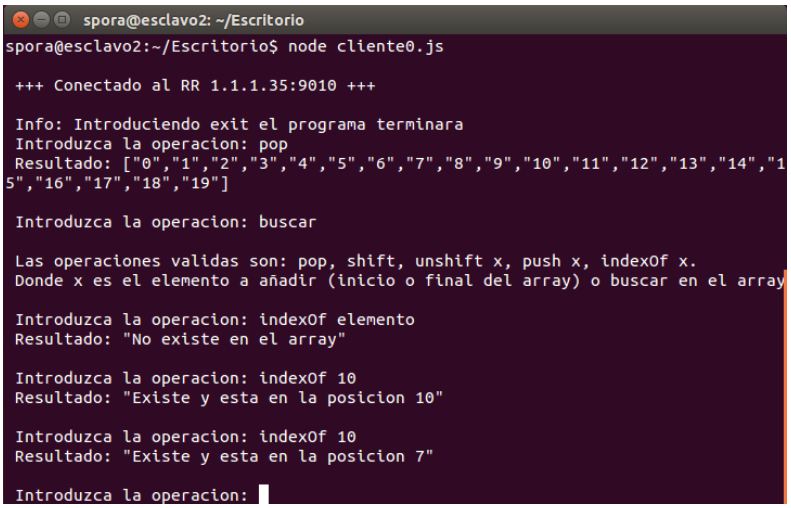


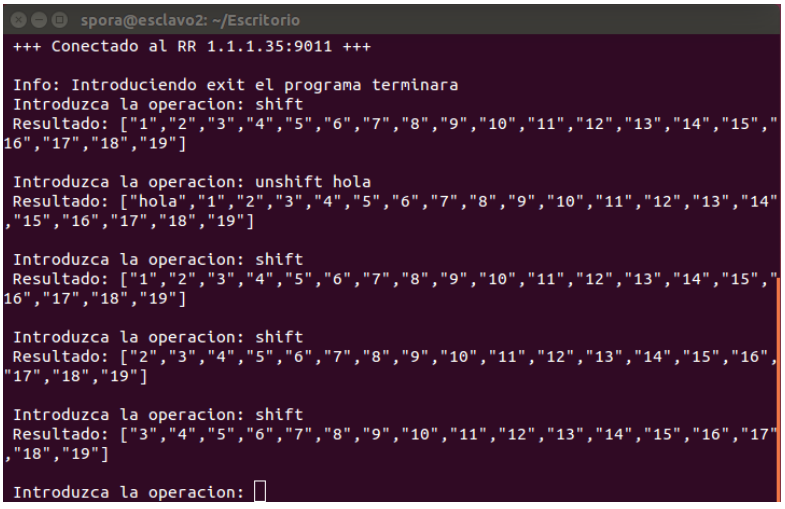
Ilustración 11 : Mensajes de Cliente

Antes de todo esto, el programa se ha tenido que conectar al programa RR de alguna manera. Esto lo realizamos mediante el uso del patrón de Req/Rep de 0MQ. Conectamos el cliente al componente y dejamos un escuchador de mensajes de respuesta escuchando en todo momento. Cuando llega un mensaje, lo primero que se realiza como se ha explicado en apartados anteriores es la convención del error primero, que consiste en el manejo de un posible error en primer lugar y más tarde devolvemos por consola el resultado de esa respuesta al usuario del programa cliente.

A continuación, vemos la ejecución del programa cliente en consola de comandos. Tenemos el cliente0 y el cliente1 que se conectan de manera distribuida a otra máquina en la que están los componentes RR0 y RR1, que hacen de puntos de conexión para cada cliente. Si el cliente introduce una entrada no valida, el programa le indicará cuales son las entradas correctas. Una vez enviada la petición, se queda a la espera de la respuesta que se muestra a continuación. Recibida la petición el cliente tiene la opción de seguir realizando peticiones o salir de la ejecución escribiendo "*exit*".



Resultado 8 : Resultados ejecución cliente0.js



Resultado 9 : Resultados ejecución cliente1.js

5.5.3 RR

El programa RR (Anexo 8.2), es el encargado de dar servicio al cliente, ofreciendole un punto de conexión, recibiendo sus peticiones, mandando estas peticiones a los ARH y devolviendo las soluciones que vienen de las réplicas.

El programa RR realiza un *bind* en un puerto especifico para que los clientes se conecten. A su vez tiene guardada una lista de los ARH existentes, se conecta a todos ellos también mediante el patron Req/Rep de 0MQ. En primer lugar el componente esta a la espera de mensajes, con escuchadores para las peticiones de los clientes y las respuestas de los ARH.

Cuando llega una nueva petición de cliente, se genera un objeto JSON llamado *req_id* con la identificación del cliente que viene en el mensaje y una secuencia para saber el número de petición que lleva realizado el cliente en la sesión. Se crea otro objeto JSON en el que esta incluido el anterior y la petición del cliente que venía en el mensaje. Al igual que el programa cliente, se genera un *string* de este objeto y se envía al ARH mediante una función llamada "*siguiente*".

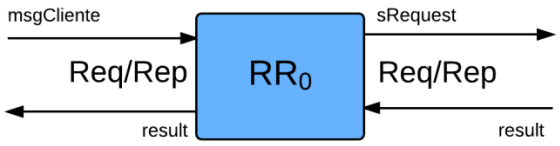


Ilustración 12 : Mensajes de RR

Esta función "*siguiente*" lo que hace es recorrer la lista de ARH que posee internamente el programa RR y enviar la petición del cliente. Se programa una función de *timeout* de un segundo que llamará a la función siguiente con un nuevo ARH como destino si no recibimos respuesta del anterior. El recorrido de la lista de ARH solo se realiza una vez.

Una vez enviado el mensaje de petición al ARH, el programa se queda a la escucha de mensajes que provengan del ARH mediante los escuchadores que tiene para cada ARH. Cuando le llegue un mensaje se realiza una comparación del objeto *req_id* que contiene el identificador del cliente y la secuencia de petición de ese cliente mediante la función *isEqual* de la librería *underScore*. Una de las características de esta librería es que es capaz de comprobar los valores de los objetos JSON y comprobar si son exactamente iguales o si son diferentes, si el resultado es verdadero entonces tenemos la comprobación que la respuesta recibida es de la petición enviada y se manda el resultado al cliente.

La ejecución de los componentes RR se realiza en otra máquina diferente a la de los clientes, por lo que su punto de conexión no tiene dirección local, se establece en la red interna. Una vez ejecutados ambos programas quedarán a la escucha de peticiones de sus clientes correspondientes, RR0 para el cliente0 y RR1 para el cliente1.

```
spora@esclavo1: ~/Escritorio
spora@esclavo1:~/Escritorio$ node RR0.js

RRMessageHandler escuchando en el puerto 9010
Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado
```

Resultado 10 : Resultados ejecución RR0.js

```
spora@esclavo1: ~/Escritorio
spora@esclavo1:~/Escritorio$ node RR1.js

RRMessageHandler escuchando en el puerto 9011
Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado

Enviando a... tcp://127.0.0.1:9026
Result recibido y enviado
```

Resultado 11 : Resultados ejecución RR1.js

5.5.4 ARH

El programa ARH (Anexo 8.3), es el núcleo de la replicación activa, encargado de recibir las peticiones desde los puntos de conexión para los clientes llamados RR, de enviar las peticiones a los puntos finales para la ejecucion de las mismas y comunicarse con el modulo DSS para obtener el número de secuencia único para cada petición teniendo en cuenta los demas ARH.

Este programa realiza 3 tipos de *bind*. El patrón Req/Rep para la comunicación con los RR, el patrón Pub/Sub para publicar las peticiones a todos los FO y el patrón Pull/Push para recibir los resultados de los FO.

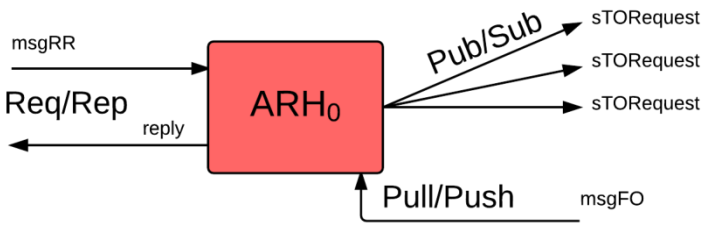


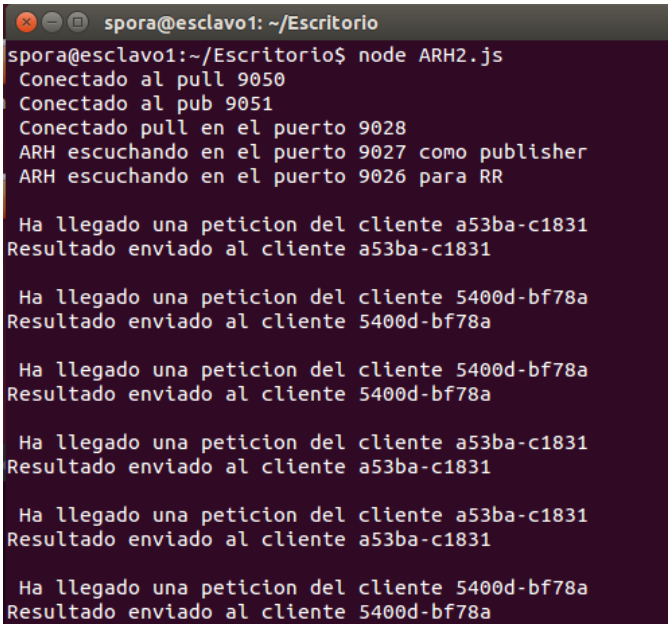
Ilustración 13 : Mensajes de ARH

Cuando el escuchador de mensajes que provienen del programa RR recibe una petición de cliente, el programa ARH hace uso de la función GetSeq del módulo DSS que nos devuelve un número de secuencia único para cada petición.

Una vez tenga el número de la secuencia, el programa comprueba si este número obtenido es mayor que una variable local que se refiere a la última petición servida, si el número es mayor entonces hacemos uso de la función GetReq del módulo DSS para obtener todas las peticiones que nos faltan y se las enviamos a todos los FO's. Una vez realizado esto, o si el número de secuencia no era mayor o igual a esa variable local, enviamos un objeto JSON transformado en *string* con el número de secuencia y la petición del cliente a todos los FO's. Una vez enviadas las peticiones se actualiza el valor local al valor de la secuencia de la última petición servida.

Ahora el programa se queda a la espera de las respuestas de la ejecución de estas peticiones mediante un escuchador de tipo *pull* de todos los FO's. Cuando llegue un mensaje debemos comprobar que tienen el mismo número de secuencia y de si ya lo hemos recibido antes de algún FO. Si pasamos las dos condiciones creamos un objeto *reply* con el objeto *req_id* de la petición y el resultado del mensaje recibido para mandárselo en *string* al RR correspondiente.

Para la ejecución de los ARH debemos tener en cuenta que dependiendo de la cantidad de peticiones que los clientes realicen solo veremos que el trabajo lo realiza un ARH. Podemos comprobar que los demás funcionan correctamente cerrando la ejecución del que está activo y las peticiones pasarán a ser atendidas por otro ARH.



Resultado 12 : Resultados ejecución ARH2.js

5.5.5 DSS

El módulo DSS (Anexo 8.4), es un programa que exporta el uso de las funciones que posee en su código, estas son la función GetSeq y GetReq.

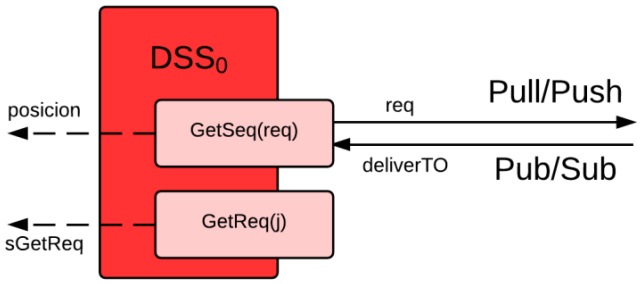


Ilustración 14 : Mensajes de ARH

La función GetSeq del programa DSS necesita del parámetro *req* que es la petición del cliente para que se compruebe si esa petición existe en el array de peticiones ya ejecutadas. Si existe se obtiene la posición y se retorna el valor de la posición al ARH, si en cambio no existe, se envía la petición, haciendo uso del patrón Pull/Push, al programa TO.

Tendremos un escuchador para los mensajes provenientes del programa TO, donde comprobaremos nuevamente la existencia de la petición en el array de peticiones ya ejecutadas y si no existe introduciremos la petición en el array. Si la petición que había llegado de ARH y la petición que ha llegado de TO son iguales entonces emitimos un evento propio que retorna al ARH el valor de la posición.

La función GetReq simplemente realiza una devolución del objeto del array de peticiones ejecutadas ubicado en la posición que se le pasa a la función como parámetro. Este programa es un módulo, no necesita de ejecución.

5.5.6 TO

El programa TO (Anexo 8.5), es una simulación de orden total mediante el uso de una cola. El programa ofrece dos puntos de conexión mediante los patrones Push/Pull y Pub/Sub para recibir las peticiones del módulo DSS y para enviar las peticiones de la cola respectivamente.

El programa tiene un escuchador de mensajes para recibir las peticiones que serán encoladas. Mientras la cola no este vacía, el programa irá mandando el objeto que está en primera posición de la cola para seguidamente desencolarlo de ella.

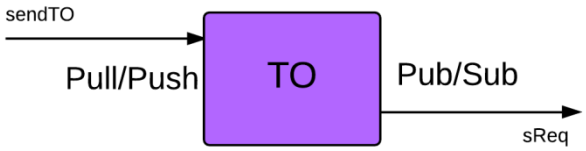


Ilustración 15 : Mensajes de TO

Los resultados que podemos ver en la ejecución del TO son la inserción de las peticiones del cliente en la cola de peticiones. En esta ejecución, donde no realizamos una alta carga de peticiones, no vemos una pila llena sino que vemos como una por una las peticiones entran a la pila y son despachadas a continuación. Se puede observar las peticiones recibidas con su contenido y de que cliente son.

```
spora@esclavo1:~/Escritorio$ node TO.js
 Conectado pull en el puerto 9050
 ARH escuchando en el puerto 9051 como publisher

Recibido: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":1},"request":"pop"}
colaTO[0]: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":1},"request":"pop"}

Recibido: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":1},"request":"shift"}
colaTO[0]: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":1},"request":"shift"}

Recibido: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":2},"request":"unshift hola"}
colaTO[0]: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":2},"request":"unshift hola"
}

Recibido: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":2},"request":"indexOf elemen
to"}
colaTO[0]: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":2},"request":"indexOf eleme
nto"}

Recibido: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":3},"request":"indexOf 10"}
colaTO[0]: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":3},"request":"indexOf 10"}

Recibido: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":3},"request":"shift"}
colaTO[0]: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":3},"request":"shift"}

Recibido: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":4},"request":"shift"}
colaTO[0]: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":4},"request":"shift"}

Recibido: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":5},"request":"shift"}
colaTO[0]: {"req_id":{"cl_id":"5400d-bf78a","cl_seq":5},"request":"shift"}

Recibido: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":4},"request":"indexOf 10"}
colaTO[0]: {"req_id":{"cl_id":"a53ba-c1831","cl_seq":4},"request":"indexOf 10"}
```

Resultado 13 : Resultados ejecución TO.js

5.5.7 FO

El programa FO (Anexo 8.6), se conecta a los puntos de conexión del ARH los cuales utilizan los patrones Pub/Sub y Push/Pull. El programa posee los escuchadores de todos los mensajes de subscripción de todos los ARH. En estos escuchadores es donde se reciben las peticiones y se comprueba si el número de secuencia de la petición es mayor que el número de secuencia esperado. Si es menor o igual, se realiza otra comprobación de si el número de secuencia esperado es igual al número de secuencia de la petición. Si es así, se llama a la función de compute, se guarda el resultado en la tabla de peticiones ejecutadas en la posición secuencia de la petición y se aumenta el número de secuencia esperado.

Se crea un objeto JSON con el número de secuencia y el resultado obtenido de la función compute y se envía mediante el socket push al ARH correspondiente.

La función compute sustituye a la lógica de negocio de un sistema de replicación activa que podría estar en otros servidores. En este proyecto en cambio se ha optado por realizar una función que ofrezca las operaciones propias para realizar en un array. Asi pues dependiendo de la petición del cliente, la función compute ejecutará una u otra función en el array y devolverá el array que ha resultado de esa petición como resultado al FO para que tras todos los pasos llegue a manos del cliente.

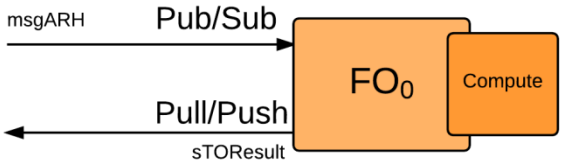


Ilustración 16 : Mensajes de FO

Los resultados de los FO son todos identicos debido a que todos los FO reciben todas las peticiones de los clientes, para ver sus resultados se muestra la ejecución de FO2.js. Podemos observar como llegan las peticiones y una vez procesada la petición mediante la función *compute* se devuelve al cliente. Todos los FO tienen las mismas peticiones recibidas y ejecutadas, es decir, tiene el mismo estado.

```
spora@esclavo1: ~/Escritorio
spora@esclavo1:~$ cd Escritorio/
spora@esclavo1:~/Escritorio$ node FO0.js

 Conectado al publicador: tcp://127.0.0.1:9021
 Conectado al pull: tcp://127.0.0.1:9022
 Conectado al publicador: tcp://127.0.0.1:9024
 Conectado al pull: tcp://127.0.0.1:9025
 Conectado al publicador: tcp://127.0.0.1:9027
 Conectado al pull: tcp://127.0.0.1:9028

Recibido: {"seq":1,"request":"pop","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":2,"request":"shift","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":3,"request":"unshift hola","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":4,"request":"indexOf elemento","ARH":2}
Ejecucion
Enviando resultado
```

```
spora@esclavo1: ~/Escritorio
spora@esclavo1:~$ cd Escritorio/
spora@esclavo1:~/Escritorio$ node FO1.js

 Conectado al publicador: tcp://127.0.0.1:9021
 Conectado al pull: tcp://127.0.0.1:9022
 Conectado al publicador: tcp://127.0.0.1:9024
 Conectado al pull: tcp://127.0.0.1:9025
 Conectado al publicador: tcp://127.0.0.1:9027
 Conectado al pull: tcp://127.0.0.1:9028

Recibido: {"seq":1,"request":"pop","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":2,"request":"shift","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":3,"request":"unshift hola","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":4,"request":"indexOf elemento","ARH":2}
Ejecucion
Enviando resultado
```

```
spora@esclavo1: ~/Escritorio
spora@esclavo1:~$ cd Escritorio/
spora@esclavo1:~/Escritorio$ node FO2.js

 Conectado al publicador: tcp://127.0.0.1:9021
 Conectado al pull: tcp://127.0.0.1:9022
 Conectado al publicador: tcp://127.0.0.1:9024
 Conectado al pull: tcp://127.0.0.1:9025
 Conectado al publicador: tcp://127.0.0.1:9027
 Conectado al pull: tcp://127.0.0.1:9028

Recibido: {"seq":1,"request":"pop","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":2,"request":"shift","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":3,"request":"unshift hola","ARH":2}
Ejecucion
Enviando resultado

Recibido: {"seq":4,"request":"indexOf elemento","ARH":2}
Ejecucion
Enviando resultado
```

Resultado 14 : Resultados ejecución FO0.js FO1.js y FO2.js

6 Conclusiones y trabajos futuros

• Conclusiones

En primer lugar, la curva del aprendizaje sobre NodeJS ha sido algo lenta al principio ya que no la conocía con anterioridad. Conforme avanzaba el proyecto ya no había problemas con el lenguaje. No me considero conocedor de todas las posibilidades que NodeJS tiene, pero si conozco la gran comunidad de desarrolladores que lo utilizan y la gran cantidad de recursos que ha generado en tan poco tiempo de vida.

La decisión de escoger las librerías 0MQ para la comunicación de los mensajes ha sido la más acertada debido a la facilidad del uso de sus diferentes tipos de sockets. Una vez programado el primer tipo de patrón en unos programas de ejemplo, el desarrollo de los demás patrones varía poco. De lo que hay que darse cuenta es, qué patrón es el más adecuado para cada caso.

La principal dificultad en el desarrollo del proyecto ha sido la de cambiar el modo de desarrollar aplicaciones con la programación orientada a eventos. No había realizado todavía ninguna aplicación de este modo, lo que ha hecho que las primeras iteraciones del desarrollo fuesen para entender este tipo de programación. Una vez finalizado puedo decir que, aunque sea difícil de seguir al principio, se hace muy cómoda de desarrollar y ofrece muchas posibilidades.

Todo el proyecto ha sido desarrollado mediante la metodología extrema, que procura que el desarrollador haga pruebas y obtenga resultados de cada cambio que se hace en el desarrollo del proyecto, Esta metodología hace que el avance del proyecto en sus primeras fases sea lento, pero a cambio nos da la seguridad de que más adelante no habrá problemas inesperados debido a nuestro desarrollo.

• Trabajos futuros

a) Desplegar y probar el proyecto de replicación activa en modo distribuido.

b) Implantar el uso de promesas en el código del proyecto.

c) Probar los sockets que ofrece la librería nanomsg.

7 Apéndice

7.1 Programación Asíncrona y sus Modelos

Es importante exponer en primer lugar como se entiende y se desarrolla la programación secuencial y en contraposición que diferencias existen entre esta y la programación asíncrona.

7.1.1 Programación Secuencial

La programación secuencial se basa en la ejecución de todos los procesos u operaciones contenidos en el sistema hasta que estas finalicen. Es decir, el programa ejecutara una serie de instrucciones de manera ordenada donde cada instrucción debe esperar a que la anterior se ejecute para comenzar.

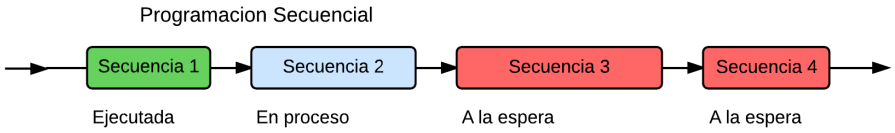


Ilustración 17 : Diagrama del flujo de un programa secuencial.

Los programas desarrollados con este tipo de programación son más sencillos y más rápidos de confeccionar. Pueden ser leídos de forma secuencial y dado que la estructura es clara por las relaciones entre las instrucciones su entendimiento también se hace sencillo.

7.1.2 Programación Asíncrona

Gran parte del tiempo de las operaciones que realiza un programa en su ejecución se pierde esperando. La solución clásica a este problema es proporcionarle ese tiempo de espera a otro proceso. Así aparece la programación asíncrona.

En la programación asíncrona la ejecución de algunas tareas se puede realizar en segundo plano mientras el programa continúa con la ejecución de otras. Estas tareas en segundo plano son las llamadas no bloqueantes. Al contrario que con la programación secuencial, la asincronia en la programación hace que el entendimiento del programa sea más complicado pero aumenta la escalabilidad mediante una ejecución no secuencial.

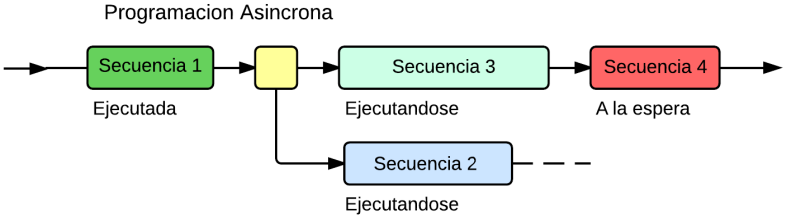


Ilustración 18 : Diagrama del flujo de un programa asíncrono.

Una de las ventajas de la programación asíncrona es la cantidad de ejecuciones en segundo plano que se pueden realizar al mismo tiempo. Este tipo de programación está cogiendo peso actualmente debido a la demanda de aplicaciones altamente escalables requeridas para operar en internet.

El inconveniente principal trata sobre cómo realizar la continuación de estas operaciones ejecutadas en segundo plano una vez que han finalizado. Para tratar este problema se han desarrollado varios modelos de programación asíncrona como el de eventos y el de promesas que se exponen a continuación.

7.1.3 Modelo de Eventos

En el modelo de eventos, un evento es definido como un cambio significativo del estado. El sistema debe tratar ese cambio de estado como un evento. Lo que realmente se produce no es el evento en sí, sino la notificación de ese evento, el evento simplemente ocurre.

El sistema desarrollado con el modelo de eventos está compuesto de emisores de eventos y consumidores de eventos, donde los consumidores deben llevar a cabo una reacción tan pronto como el evento este presente. Existen dos tipos de arquitecturas en el modelo de eventos, la centralizada y la distribuida.

7.1.3.1 Arquitecturas

En el modelo de eventos está presente una arquitectura llamada centralizada. La arquitectura centralizada es la que posee un bus central que asume el papel de mediador que se encarga de hacer efectivo el proceso de registro de los clientes escuchadores y de lanzar las notificaciones bajo demanda de los proveedores a los mismos. Este esquema se conoce como publicador y subscriptor (*PUB/SUB*)[11].

Otro tipo de arquitectura en el modelo de eventos de la programación asíncrona es el de la arquitectura distribuida. Al contrario que la arquitectura centralizada, en esta, cada proveedor es responsable de gestionar la suscripción de sus clientes y de enviar las notificaciones cuando se produce un evento. Este esquema se conoce como observador y observable o también llamado *event emitters*[12] en NodeJS.

[11] Es un patrón de comunicación para envío de mensajes.

[12] Emisor de eventos del modelo de eventos de la programación asíncrona.

7.1.3.2 Control de flujo

❖ **Secuenciamiento**

El flujo llamado secuenciamiento es el que se define dentro de este modelo cuando se realiza el programa a través del encadenamiento de escuchadores. La salida de un escuchador será la entrada del siguiente. El resultado final es el cúmulo de los resultados de cada escuchador de la cadena, el cual será recogido y procesado por el cliente [16].

```
01: var EventEmitter = require('events').EventEmitter;
02: var addEmitter = new EventEmitter();
03: var mulEmitter = new EventEmitter();
04:
05: function add(x, y) {
06:   addEmitter.emit('result', x+y);
07: }
08:
09: function mul(z) {
10:   addEmitter.on('result', function (data){
11:     mulEmitter.emit('result', data * z);
12:   });
13: }
14:
15: mul(5);
16: mulEmitter.on('result', function (data){
17:   console.log(data);
18: });
19: add (2,3);
```

Tabla 35 : Código emisión de eventos secuencial.

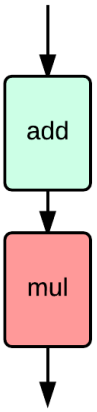


Ilustración 19 : Diagrama de eventos secuenciales.

❖ **Paralelización**

En el modelo de eventos lo único que hace falta para paralelizar acciones que queremos que se ejecuten en nuestro sistema es hacer que cada una opere de manera independiente y lance eventos cuando genere los resultados. De este modo cada evento se ejecuta de manera paralela e independiente.

```
01: var EventEmitter = require('events').EventEmitter;
02: var addEmitter = new EventEmitter();
03: var subEmitter = new EventEmitter();
04:
05: function add(data) {
06:   addEmitter.emit('result', x+y);
07: }
08:
09: function sub(x, y) {
10:   subEmitter.emit('result', x-y);
11: }
```

```
01: var EventEmitter = require('events').EventEmitter;
02: var emitter = new EventEmitter();
03:
04: function add(x, y) {
05:   emitter.emit('add', x+y);
06: }
07:
08: function sub(x, y) {
09:   emitter.emit('sub', x-y);
10: }
```

Tabla 36 : Dos opciones de código para la paralelización de eventos.

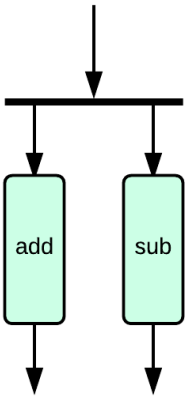


Ilustración 20 : Diagrama de eventos paralelos.

❖ **Sincronización**

Para sincronizar eventos que son paralelos hace falta de un proceso que esté a la escucha de todas las partes del sistema que emitan eventos. Las funciones manejadoras deben incluir lógica de tratamiento que emita nuevos eventos más generales con datos resultantes. A esto se le llama correlación de eventos. Existe una gran colección de patrones de diseño propios de las arquitecturas de correlación entre los que se cuentan las transformaciones, los agregados, los filtros o las abstracciones temporales.

```
01: var EventEmitter = require('events').EventEmitter;
02: var addEmitter = new EventEmitter();
03: var subEmitter = new EventEmitter();
04: //Adjuntar aqui: primer ejemplo de paralelizacion
05: function mul() {
06:   var r;
07:   function h(data) {
08:     if (r) {console.log (r * data)}
09:     else r = data;
10:   };
11:   addEmitter.on ('result', h);
12:   subEmitter.on ('result', h);
13: }
14: mul ();
```

Tabla 37 : Código de continuación para la sincronización de eventos.

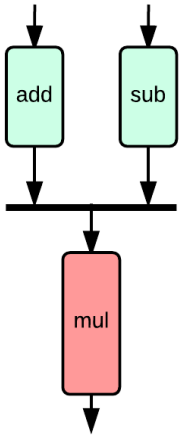


Ilustración 21 : Diagrama de sincronización de eventos

7.1.3.3 Librerías

Dado que se ha realizado una diferenciación dentro del modelo de eventos entre arquitecturas centralizadas y arquitecturas distribuidas hay que exponer las librerías que existen para cada una de ellas.

❖ **En Arquitecturas Distribuidas**

Algunas de las librerías presentes para las arquitecturas distribuidas de la programación de eventos son:

- ***Events:***

  Events es un módulo estándar de NodeJS utilizado para trabajar con eventos. Dispone de un constructor que genera emisores de eventos con todos los métodos necesarios para la creación, la eliminación y la propagación de eventos.

- ***Util:***

  Algunos autores utilizan el método *inherits* de la librería estándar útil con el fin de explotar las capacidades de la librería *events* por herencia en lugar de por delegación.

❖ **En Arquitecturas Centralizadas**

- ***Postal:***

  La librería Postal, en las arquitecturas centralizadas, es un bus de comunicaciones que implementa el patrón PUB/SUB tanto para cliente como para servidor. Usa expresiones regulares sobre el tipo de eventos para gestionar familias y goza de una buena comunidad.

7.1.3.4 Ventajas e inconvenientes

Las ventajas del modelo de eventos en la programación asíncrona son la gran cantidad de respuestas que ofrece este método de programación a nuevos problemas y las grandes posibilidades que ofrece, entre las cuales se encuentran el desacoplamiento nominal, la facilidad de extender el sistema mediante nuevos manejadores y el razonamiento en local de problemas desacoplados.

Los inconvenientes de este modelo, que mantienen aquellos que no apoyan su uso, son las incomodidades de utilizar los eventos, que la coordinación de los procesos sea complicada, que la lógica de secuenciamiento queda diluida entre los manejadores, el carácter invasivo de esta solución y la dificultad de seguir, mantener y depurar a medida que crece el tamaño del problema.

7.1.4 Modelo de Promesas

Este modelo es una manera alternativa de modelar asincronía. Es una construcción explícita del flujo de ejecución donde el manejo de errores está más controlado en la combinación de diferentes flujos asíncronos.

7.1.4.1 Qué es una promesa

Una promesa es una abstracción computacional que representa un compromiso por parte de la operación no bloqueante invocada para entregar una respuesta al programa que la llama cuando obtiene un resultado tras su finalización. La promesa es un objeto que expone dos métodos *then* y *fail* para incluir la lógica de tratamiento en caso de éxito o fracaso una vez la operación no bloqueante ha terminado.

El ciclo de vida de una promesa es sencillo, su valor reside en dos principios. El primero es que la lógica del tratamiento en caso de éxito o fracaso se ejecuta una sola vez. Y la segunda es que se garantiza la ejecución de la lógica de éxito o fracaso aunque la promesa se resuelva antes de haber inyectado sus manejadores. La promesa espera, si es necesario, a disponer de sus manejadores.

Existen diversas maneras de obtener promesas que pueden identificarse como patrones de construcción que aparecen recurrentemente al utilizar este modelo. A continuación comentamos los más relevantes con ejemplos en la librería Q.

Instalar el modulo Q con el comando: `npm install q`

`var promise = getPromise();`	Obtención directa
`var promise = Q.resolve(value);`	Resolución directa
`var promise = Q.reject(error);`	Rechazo directo

`var f = Q.denodeify(fs.readFile);`
`var f = Q.nfbind (fs.readFile);`
`Q.nfcall (fs.readFile (...));`
	Denodificación

`return Q.fcall (function () {`
`  return value;`
`});`
	Invocación funcional

`var deferred = Q.defer();`
`if (error) deferred.reject (error);`
`if (data) deferred.resolve (data);`
`return deferred.promise;`
	Factoría de diferidos

Tabla 38 : Ejemplos de obtención de promesas de la librería Q

7.1.4.2 Control de Flujo

❖ **Secuenciamiento**

El control de flujo en secuencia utilizando promesas se consigue mediante el encadenamiento sucesivo de varios métodos *then*. Esto provoca el esperado comportamiento secuencial ya que cada *then* genera una promesa que encapsula el valor devuelto, obligando a mantener el orden.

```
Var Q = require('q');
Q.resolve (2)
.then (function (value) {
  return value - 1;
})
.then (function (value) {
  return value - 1;
})
.then (function (value) {
  if (value===0) throw Error ();
  return (8 / value);
})
.then (function (value) {
  return value + 1;
})
.fail (function (error) {
  console.log (error);
});
```

Tabla 39 : Código de secuenciamiento de promesas.

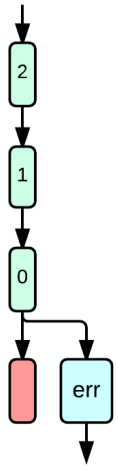


Ilustración 22 : Diagrama de promesas en secuencia.

❖ **Paralelización**

Para conseguir que dos procesos sean paralelos gracias a la paralelización se debe invocar directamente las operaciones debido a que estas tienen un comportamiento no bloqueante. En este modelo la programación es más natural y comprensible ya que cada operación devuelve un valor en forma de promesa.

```
01: var r1 = add(2, 3);   | 01: var r1 = Q.fcall (add, 2, 3);
02: var r2 = sub(3, 4);   | 02: var r2 = Q.fcall (sub, 3, 4);
```

```
01: function doParallel(fns, params) {
02:   var promises = [];
03:   for (var index = 0; index < fns.length; index ++) {
04:     var p = Q.fapply (fns[index], params[index]);
05:     promises.push (p);
06:   }
07:   return promises;
08: }
```

```
01: var promises = doParallel ([add, sub], [[2,3],[4,5]]);
```

Tabla 40 : Código de paralelización de promesas.

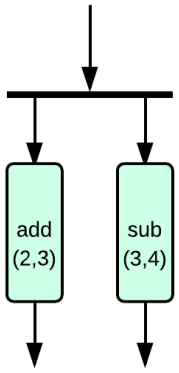


Ilustración 23 : Diagrama promesas paralelas

❖ **Sincronización**

Es sencillo realizar políticas de sincronización teniendo los resultados de las promesas. El método *all* genera una promesa que toma un array de promesas y se resuelve a un array de valores cuando todas las promesas se han resuelto. Si hay fallo se devuelve el de la primera promesa del array en fallo. El método *allSettled* por su parte resuelve cuando todas las promesas están en *settled*. Y el método spread es la versión en array del método *then*.

```
01: var promises = doParallel ([add, sub], [[2,3],[4,5]]);
02: Q.all (promises).then (function (values) {
03:   console.log (values);
04: });
```

```
01: var promises = doParallel ([mul, div], [[2,3],[4,0]]);
02: Q.allSettled (promises).spread (function (vMul, vDiv) {
03:   if (vMul.state === "fulfilled")
04:     console.log (vMul.value);
05:   if (vDiv.state === "fulfilled")
06:     console.log (vDiv.value);
07: });
```

Tabla 41 : Dos opciones de código para la sincronización de promesas.

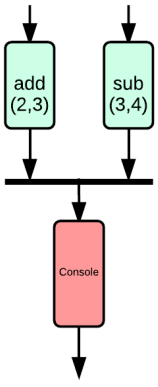


Ilustración 24 : Diagrama de sincronización de promesas

7.1.4.3 Librerías

Existen varias librerías que implementan el modelo de promesas. Tal vez la de mayor comunidad es Q además de ser muy potente, aunque otras como When o RSVP también gozan de bastante empuje [17].

Entre todas ellas Bluebird, Q y When son *toolkits* que, además de proveer al desarrollador la utilización de promesas, también te ofrecen muchas funciones de ayuda para manejar las promesas bajo las librerías asíncronas.

Lie, RSVP y Then-Promise son librerías de promesas básicas que tienen las funciones estáticas *resolve*, *reject* y todos los métodos *.then* y *.catch*.

En la siguiente tabla se observan las diferentes librerías para realizar promesas fijándose solamente en las características directamente ligadas a las promesas.

	Promises/A+	Progression	Delayed promise	Parallel synchronizaction
Bluebird	✓	✓ (+389 B)	✓ (+615 B)	✓ +(272 B)
Catiline	✓	-	-	✓
ES6 Promise polyfill	✓	-	-	✓
Jquery	-	✓	-	✓
kew	✓	-	✓	✓
lie	✓	-	-	-
MyDeferred	✓	-	-	✓
MyPromise	-	-	✓	✓
Q	✓	✓	✓	✓
RSVP	✓	-	-	✓
when	✓	✓	✓	✓
Yui	✓	✓	-	✓

Tabla 42 : Tabla comparativa sobre las librerías para la implementación de promesas [17].

7.1.4.4 Ventajas e inconvenientes

La ventaja principal, haciendo uso de las promesas, es que hemos recuperado el estilo secuencial en el desarrollo de aplicaciones asíncronas manteniendo su carácter no bloqueante. Recuperamos el *return* y la asignación. Vemos el programa de tal manera que las promesas son valores de futuro.

A pesar de las ventajas que aporta este modelo usando las promesas, al igual que en el de modelo de eventos, hay gente que no lo apoya puesto que el uso de esas promesas es incómodo de tratar. Resulta difícil de depurar hasta que las promesas no se han resuelto y es imprescindible usar librerías de terceros para gestionarlas.

8 Anexos

8.1 Código del programa CLIENTE.js

```
//REQUIRES DE NODE.JS
var readline = require('readline');  //Para leer input de teclado
var zmq = require('zmq');                   //Sockets tipo ZMQ
//VARIABLES
var rq = zmq.socket('req');                 //Socket de Request
var id = randString();                      //Identificacion mediante string aleatorio
var rl = readline.createInterface(process.stdin, process.stdout); //Para leer input
//================================CODIGO=======================================
//CONECTAMOS A UN RR:
rq.connect('tcp://127.0.0.1:9010');
console.log('\n +++ Conectado al RR 127.0.0.1:9010 +++\n');
console.log(' Info: Introduciendo exit el programa terminara');
NuevaPeticion();                    //Llamamos a la funcion NuevaPeticion
//==============================LISTENERS=====================================
//Listener para los mensajes de vuelta al cliente con el resultado del request
rq.on('message',function(msg,err){
        if( err ){
                throw err;
                console.log(err);
        }
        var recibido = JSON.parse(msg);      //Pasar a JSON el string recibido
        console.log(' Resultado: ' + JSON.stringify(recibido.res)+'\n');
        NuevaPeticion();                     //Llamamos de nuevo a la funcion
});
//==============================FUNCIONES=====================================
//FUNCION en la que creamos el objeto JSON con el identificador y el request
//para enviarselo a nuestro RR
function NuevaPeticion(){
        rl.question(" Introduzca la operacion: ", function LeerTeclado(re) {
                var args = re.trim().split(' ');
                if( (args[0]=='pop' || args[0]=='shift') && args[1]==null){
                       //JSON de la identificacion de user y el request
                        var envioCliente_RR ={
                                ide: id,
                                request: args[0]
                        }
                        //String del JSON para enviar
                        sEnvioCliente_RR = JSON.stringify(envioCliente_RR);
                        rq.send(sEnvioCliente_RR);    //Enviamos al RR
                }
                else if((args[0]=='push' || args[0]=='unshift' || args[0]=='indexOf')
                                                                  && args[1]!=null){
                        //JSON de la identificacion de user y el request
                        var envioCliente_RR ={
                                ide: id,
                                request: args[0]+' '+args[1]
                        }
                        //String del JSON para enviar
                        sEnvioCliente_RR = JSON.stringify(envioCliente_RR);
                        rq.send(sEnvioCliente_RR);    //Enviamos al RR
                }
                else if(args[0]=='exit'){
                        rq.close();
                        process.exit();
                }
                else{
                        console.log('\n Las operaciones validas son: pop, shift,
                                                        unshift x, push x, indexOf x.');
                        console.log(' Donde x es el elemento a añadir (inicio o final
                                                         del array) o buscar en el array\n');
                        NuevaPeticion();
                }
        });
};
//_____________________________________________________________________________
//FUNCION de string aleatorio
function randString () {
        var len = 10
        , charSet = '0123456789abcdef'
```

```
        , result = [];
        for (var i = 0; i < len; ++i) {
                result.push(charSet[Math.floor(Math.random() * charSet.length)]);
        }
        result.splice(len / 2, 0, ['-']);
        return result.join('');
}
```

Tabla 43 : Código del programa cliente

8.2 Código del programa RR.js

```
//REQUIRES DE NODE.JS
var zmq = require('zmq');            //Sockets tipo ZMQ
var underScore = require('underscore');//Underscore
//VARIABLES
var rp = zmq.socket('rep');          //Socket de Reply
var rq = zmq.socket('req');          //Socket de Request
var cl_seq = 0;                      //Secuencia del cliente
var req_id;                          //JSON de identificacion de cliente
var request;                         //JSON del request
var recibido;                        //Booleano para ver si hemos recibido mensaje
var sRequest;                        //String de JSON
var resultJSON;                      //JSON del resultado
var i;                               //Posicion array servers hlist
var rqs = [];                        //Array para almacenar los ARH
var hlist = ['tcp://127.0.0.1:9020',
             'tcp://127.0.0.1:9023',
              'tcp://127.0.0.1:9026'];
//================================CODIGO=====================================
for(var k=0; k<hlist.length; k++){                  //Conectamos todos los h(i)
        rqs[k] = zmq.socket('req');                 //Socket de Request
        rqs[k].connect(hlist[k]);                   //Conectarlo
}
rp.bind('tcp://127.0.0.1:9010', function(err){      //Bind para conexiones
        if(err) console.log(err)
        else console.log("\n RRMessageHandler escuchando en el puerto 9010")
});
//============================LISTENERS=====================================
//Listener para cuando recibimos la operacion de cliente.js
rp.on('message',function(msgCliente,err){
        if( err ) {
                throw err;
                console.log(err);
        }
        var msgClienteJSON = JSON.parse(msgCliente);
        i = hlist.length-1;          //Inicializamos i a la longitud del hlist menos 1
        cl_seq = cl_seq + 1;         //Aumentamos la secuencia de cliente
        req_id = {                   //Objeto JSON para req_id
                cl_id: msgClienteJSON.ide,
                cl_seq: cl_seq
        };
        request = {                  //Objeto JSON para la peticion
                req_id: req_id,
                request: msgClienteJSON.request
        };
        sRequest = JSON.stringify(request);  //String del JSON request
        siguiente(i);         //Funcion para enviar el request al server o servers
});
//_____________________________________________________________________________
//Listener para cuando recibimos el resultado de ARH.js
for(var j=0;j<hlist.length;j++){
        rqs[j].on('message',function(result,err){
                if( err ) {
                        throw err;
                        console.log(err);
                }
                recibido = true;             //Hemos recibido mensaje = true
                resultJSON = JSON.parse(result);
                //Si los req_id son iguales
                if( underScore.isEqual(resultJSON.req_id, request.req_id) ){
                        rp.send(result);      //Enviamos el resultado al cliente
                        console.log(' Result recibido y enviado\n');
                }
                else{
```

```
                        console.log(' Req_id distinto entre resultado y envio\n')
                }
        });
}
//_____________________________________________________________________________
//LISTENER para Ctrl + C -> salir
process.on('SIGINT', function() {           //Cerrar adecuadamente cada socket
        rp.close();
        for(var k=0; k<hlist.length; k++){
                rqs[k].close();
        }
        process.exit();
});

//==============================FUNCIONES=======================================
//FUNCION Siguiente para pasar el request al siguiente servidor
function siguiente(i){
        if(i>=0){
                rqs[i].send(sRequest);       //Enviamos al servidor i el string
                console.log(' Enviando a... '+ hlist[i].toString());
                recibido = false;
                setTimeout(function TDeEspera(){     //Cada 1 segundo repetir la funcion
                        if(!recibido){               //Si no hemos recibido mensaje
                          console.log(' |-El servido parece estar caido, reenvio: ');
                          siguiente(i-1);     //Llamar a la funcion con i-1
                        }
                },1000);
        }
}
```

Tabla 44 : Código del programa RR

8.3 Código del programa ARH.js

```
//REQUIRES DE NODE.JS
var zmq = require('zmq');                    //Sockets tipo ZMQ
var underScore = require('underscore');      //Underscore
var Sequencer = require('./DSS0.js');        //Utilizamos el modulo DSS
//VARIABLES
var pullSocket = zmq.socket('pull');        //Socket de pull
var pubsocket = zmq.socket('pub');           //Publisher
var rp = zmq.socket('rep');                  //Socket de Reply
var msgRRJSON;                               //JSON del cliente
var TORequest;                               //JSON para envio de TO
var sTORequest;                              //String del anterio JSON
var lastServedReq = 0;                       //Ultima peticion atendida
var seq;                                     //Numero de secuencia
var seqaux;
var recibido = false;                        //Si hemos recibido un mensaje
//================================CODIGO========================================
rp.bind('tcp://127.0.0.1:9020', function(err){             //Bind para reply
        if(err) console.log(err)
        else console.log(" ARH escuchando en el puerto 9020 para RR")
});
//_____________________________________________________________________________
pubsocket.bind('tcp://127.0.0.1:9021', function(err){      //Bind para publicar
        if(err) console.log(err)
        else console.log(' ARH escuchando en el puerto 9021 como publisher')
});
//_____________________________________________________________________________
pullSocket.bind('tcp://127.0.0.1:9022')                    //Bind para pull
console.log(' Conectado pull en el puerto 9022')
//==============================LISTENERS=======================================
//Listener para cuando recibimos la peticion de algun RR
rp.on('message',function(msgRR,err){
        if( err ) {
                throw err;
                console.log(err);
        }
        recibido = false;     //Control de mensaje recibido, admitir solo el primero
        msgRRJSON = JSON.parse(msgRR);              //Pasamos el String a JSON
        console.log('\n Ha llegado una peticion del cliente '+msgRRJSON.req_id.cl_id);
        Sequencer.GetSeq(msgRR,function callback(seq){ //Obtenemos la secuencia del TO
                if(seq > lastServedReq + 1){
                        for(var j=lastServedReq+1;j<seq;j++){
```

```
                                //String recibido del JSON en la posicion j
                                var reqj = Sequencer.GetReq(j);
                                //Pasamos el String a JSON
                                 var reqjJSON = JSON.parse(reqj);
                                 TORequest = {          //Objeto JSON hacia FO's
                                        seq: j,
                                        request: reqjJSON.request,
                                        ARH: 0
                                 };
                                //Pasamos el JSON del TORequest a string
                                 sTORequest=JSON.stringify(TORequest);
                                //Publicamos a todos los sub el objeto TORequest
                                 pubsocket.send(''+sTORequest);
                        }
                }
                TORequest = {                           //Objeto JSON hacia FO's
                        seq: seq,
                        request: msgRRJSON.request,
                        ARH: 0
                };
               //Pasamos el JSON del TORequest a string
                sTORequest=JSON.stringify(TORequest);
               //Actualizamos el valor de lastServerRequest
                lastServedReq = Math.max(lastServedReq,seq);
               //Publicamos a todos los sub el objeto TORequest
                pubsocket.send(''+sTORequest);
                seqaux = seq;
        });
});
//_____________________________________________________________________________________
//Listener para el mensaje de resultado desde FO
pullSocket.on('message', function(msgFO,err){
        if( err ) {
                throw err;
                console.log(err);
        }
        var msgFOJSON = JSON.parse(msgFO);   //Pasamos el String a JSON
       //Si lo que nos llega del FO es de la peticion actual
       if(seqaux == msgFOJSON.seq){
       //Si todavia no hemos recibido mensaje alguno de los FO's
                if (!recibido){
                        recibido = true;
                        var reply = {                   //Objeto JSON hacia RR
                                req_id: msgRRJSON.req_id,
                                res: msgFOJSON.result
                        };
                        //Pasamos el JSON del reply a string
                        sReply = JSON.stringify(reply);
                        console.log('Resultado enviado al cliente '
                                                +msgRRJSON.req_id.cl_id);
                        msgRRJSON=null;
                        rp.send(sReply);    //Enviamos a RR conectado al socket rp
                }
        }
});
//_____________________________________________________________________________________
//LISTENER para Ctrl + C -> salir
process.on('SIGINT', function() {           //Cerrar adecuadamente cada socket
        pubsocket.close();
        pullSocket.close();
        rp.close();
        process.exit();
});
```

Tabla 45 : Código del programa ARH

8.4 Código del programa DSS.js

```
//REQUIRES DE NODE.JS
var events = require('events');                             //Eventos
var zmq = require('zmq');                                   //Sockets tipo ZMQ
var underScore = require('underscore');             //Underscore
//VARIABLES
var pushSocket = zmq.socket('push');        //Socket de pull
var subSocket = zmq.socket('sub');                  //Socket de pull
```

```
var emision = new events.EventEmitter();     //Variable para emision de evento propio
var sequenced = [];                          //Array de secuenciados
var localSeq = 1;                            //Secuencia local
var deliverTOJSON, reqJSON;                  //JSON
var primero=false;                           //Variable
//================================CODIGO=====================================
pushSocket.connect('tcp://127.0.0.1:9050');         //conectarlo al pull
console.log(' Conectado al pull 9050');
subSocket.connect('tcp://127.0.0.1:9051');          //conectarlo al pub
subSocket.subscribe('');                            //Subscibirse a la "cadena vacia"
console.log(' Conectado al pub 9051')
//================================FUNCIONES==================================
//FUNCION GETSEQ(req)
exports.GetSeq = function(req,callback){
        reqJSON = JSON.parse(req);
        if( sequenced.indexOf(reqJSON)==-1 ){       //Si no existe el objeto en el array
                pushSocket.send(req);               //Mandamos a TO el request
        }
        else{
        //Si existe guardamos en la variable su posicion
                var posicion = sequenced.indexOf(reqJSON);
                return posicion;                    //Retornamos la posicion
        }
        if(!primero){                               //Si primero es false
               //Lo asignamos a true para que solo entre el primer emit cada vez
                primero = true
               //Evento dssmsg con parametro msgposicion
                emision.on('dssmsg', function(msgposicion){
                        //callback de la posicion
                        callback(msgposicion);
                        return msgposicion;
                });
        }
}
//FUNCION GETREQ(j)
exports.GetReq = function (j){
        sGetReq = JSON.stringify(sequenced[j]);     //Obtenemos el objeto de la posicion
        return sGetReq;                             //Lo retornamos
}
//================================LISTENERS==================================
//Listener para los mensajes que llegan de TO
subSocket.on('message',function(deliverTO,err){
        if( err ) {
                throw err;
                console.log(err);
        }
        deliverTOJSON = JSON.parse(deliverTO);
        if( sequenced.indexOf(deliverTOJSON)==-1 ){ //Si no existe el objeto en el array
               //Introducimos el objeto en la posicion del array localseq
                sequenced[localSeq] = deliverTOJSON;
                localSeq = localSeq + 1;
                var bool = underScore.isEqual(deliverTOJSON, reqJSON);
                reqJSON=null;
                if( bool ){                                 //Si los objetos son iguales
                        emision.emit('dssmsg', localSeq-1);  //Emitimos el evento dssmsg
                }
        }
});
//____________________________________________________________________________
//LISTENER para Ctrl + C -> salir
process.on('SIGINT', function() {           //Cerrar adecuadamente cada socket
        subSocket.close();
        pushSocket.close();
        process.exit();
});
```

Tabla 46 : Código del programa DSS

8.5 Código del programa TO.js

```
//REQUIRES DE NODE.JS
var zmq = require('zmq');                   //Sockets tipo ZMQ
//VARIABLES
var pullSocket = zmq.socket('pull');        //Socket de pull
var pubsocket = zmq.socket('pub');          //Socket de publisher
```

```
var colaTO = [];
//================================CODIGO=====================================
pullSocket.bind('tcp://127.0.0.1:9050');                    //Bind para pull
console.log(' Conectado pull en el puerto 9050');
//___________________________________________________________________________
pubsocket.bind('tcp://127.0.0.1:9051', function(err){       //Bind para publicar
        if(err) console.log(err);
        else console.log(' ARH escuchando en el puerto 9051 como publisher');
});
//==============================FUNCIONES====================================
//Listener para el mensaje de DSS
pullSocket.on('message', function(sendTO,err){
        if( err ) {
                throw err;
                console.log(err);
        }
        var sendTOJSON = JSON.parse(sendTO);
        console.log('\nRecibido: '+sendTO);
        colaTO.push(sendTOJSON);              //Encolamos el objeto
        var i = 0;
        while(colaTO[i] != null){             //Mientras la cola no este vacia
                var sReq = JSON.stringify(colaTO[i]);
                console.log('colaTO['+i+']: ' + sReq);
                pubsocket.send(''+sReq);      //Enviamos el objeto de la primera posicion
                colaTO.shift();               //Eliminamos de la cola ese objeto
        }
});
```

Tabla 47 : Código del programa TO

8.6 Código del programa FO0.js

```
//REQUIRES DE NODE.JS
var zmq = require('zmq');             //Sockets de tipo ZMQ
//VARIABLES
var executed = [];                    //Array de operaciones ejecutadas
var expectedSeq = 1;                  //Secuencia esperada
var result, seq;                      //result y seq del sequencer
var TOResult;                         //JSON
var sTOResult;                        //string de JSON
var subSocket = [];                   //Array de conexiones y listeners de tipo sub
var pushSocket = [];                  //Array de conexiones y listeners de tipo push
var publist = ['tcp://127.0.0.1:9021',      //Array con las direcciones de tipo pub
                        'tcp://127.0.0.1:9024',
                        'tcp://127.0.0.1:9027'];

var pulllist = ['tcp://127.0.0.1:9022',     //Array con las direcciones de tipo pull
                         'tcp://127.0.0.1:9025',
                         'tcp://127.0.0.1:9028'];
var arrayPrueba = [];                       //Array para el supuesto
for(k=0; k<21; k++){
        arrayPrueba[k]=k.toString();
}
//================================CODIGO=====================================
console.log('');
for(var k=0;k<publist.length;k++){
        subSocket[k] = zmq.socket('sub');           //crear el sub para cada hi
        subSocket[k].connect(publist[k]);           //conectarlo
        console.log(' Conectado al publicador: ' + publist[k]);
        subSocket[k].subscribe('');                 //Subscibirse a la "cadena vacia"
//___________________________________________________________________________
        pushSocket[k] = zmq.socket('push');         //crear el push para cada pull
        pushSocket[k].connect(pulllist[k]);         //conectar a todos los hi con push
        console.log(' Conectado al pull: ' + pulllist[k]);
}
//==============================LISTENERS====================================
//Listeners para todos los sub's cuando reciben el request del ARH pub
for(var i=0; i<publist.length; i++){
        subSocket[i].on('message', function(msgARH,err) {
                if( err ) {
                        throw err;
                        console.log(err);
                }
                var msgARHJSON = JSON.parse(msgARH); //Pasar a JSON el string recibido
                console.log('\nRecibido: '+msgARH);
```

```
                seq = msgARHJSON.seq;     //Asignamos a seq el valor de seq del objeto
                if(seq>expectedSeq){
                        //DO NOTHING
                }
                else{
                        if(seq == expectedSeq){
                               //Computar el request y almacenar en resultado
                                result = compute(msgARHJSON.request);
                               //Almacenar el resultado en el array de ejecutados
                                executed[seq] = result;
                               //Iterar una posicion
                                expectedSeq = expectedSeq + 1;
                        }
                        TOResult = {          //Objeto JSON del resultado computado
                                seq: seq,
                                result: executed[seq]
                        };
                        //Pasamos el JSON del resultado al hi
                        sTOResult=JSON.stringify(TOResult);
                        pushSocket[msgARHJSON.ARH].send(sTOResult); //Envio de TOResult
                        console.log('Enviando resultado');
                }
        });
}
//_____________________________________________________________________________
//LISTENER para Ctrl + C -> salir
process.on('SIGINT', function() {                    //Cerrar adecuadamente cada socket
        for(var k=0;k<publist.length;k++){
                pushSocket[k].close();
                subSocket[k].close();
        }
        process.exit();
});
//==============================FUNCIONES====================================
//FUNCION compute para ejecutar el request y obtener el "resultado"
function compute(request){
        console.log('Ejecucion');
        var args = request.split(' ');
        if(args[0]=='pop'){
                arrayPrueba.pop();
        }
        else if(args[0]=='shift'){
                arrayPrueba.shift();
        }
        else if(args[0]=='push'){
                arrayPrueba.push(args[1]);
        }
        else if(args[0]=='unshift'){
                arrayPrueba.unshift(args[1]);
        }
        else if(args[0]=='indexOf'){
                var pos = arrayPrueba.indexOf(args[1]);
                if(pos!=-1){
                        return 'Existe y esta en la posicion '+pos;
                }
                else{
                        return 'No existe en el array'
                }
        }
        return arrayPrueba;
}
```

Tabla 48 : Código del programa FO

8.7 Código del módulo auxfunctions.js

```
module.exports = {

// *** getLoad function

getLoad : function() {
var fs = require('fs')
, data = fs.readFileSync("/proc/loadavg") // version sincrona
, tokens = data.toString().split(' ')
```

```
, min1 = parseFloat(tokens[0])+0.01
, min5 = parseFloat(tokens[1])+0.01
, min15 = parseFloat(tokens[2])+0.01
, m = min1*10 + min5*2 + min15;
return m;
},

// *** randNumber function

randNumber : function(upper, extra) {
 var num = Math.abs(Math.round(Math.random() * upper));
 return num + (extra || 0);
},

// *** randTime function

randTime : function(n) {
return Math.abs(Math.round(Math.random() * n)) + 1;
},

// *** showArguments function

showArguments : function(a) {
for (var k in a)
console.log('\tPart', k, ':', a[k].toString());
}
}
```

Tabla 49 : Código del módulo auxfunctions

8.8 GIT del laboratorio

Tabla 2	cluster.js
Tabla 7	hwclient.js
Tabla 10	hwserver.js
Tabla 12	pullpushclient.js
Tabla 13	pullpushserver.js
Tabla 14	subscriber.js
Tabla 15	publisher.js
Tabla 17	randnum.js
Tabla 18	rrclient.js
Tabla 19	rrbroker.js
Tabla 20	rrworker.js
Tabla 25	lbclient.js
Tabla 26	lbworker.js
Tabla 27	lbbroker.js
Tabla 43	cliente0.js
Tabla 44	RR0.js
Tabla 45	ARH0.js
Tabla 46	DSS0.js
Tabla 47	TO.js
Tabla 48	FO0.js
Tabla 49	auxfunctions.js

Tabla 50 : Tabla de relaciones entre códigos de la memoria y programas de GIT

Estos son los nombres de los programas que están el GIT del laboratorio y se han mostrado en las tablas de la memoria.

No se añaden los programas restantes de la replicación activa ya que son prácticamente idénticos y simplemente cambian el número del nombre como en el caso de cliente0.js y cliente1.js y las direcciones IP correspondientes en su código.

9 Bibliografía

[1] Joyent Inc. (2014) NodeJS. [Online]. http://www.nodejs.org/

[2] Todd Eichel. Node.js: Building for Scalability with Server-Side JavaScript. Articulo Web.

[3] Tom Hugbes-Croucher & Mike Wilson, *Node: Up and Running*, Andy Oram and Simon St.Laurent, Ed. Sebastopol: O'Reilly Media, May 2012.

[4] Express: Fast, unopinionated, minimalist web framework for Node.js. [Online]. www.expressjs.com

[5] Socket.IO. [Online]. www.socket.io

[6] MIT. (2011) Mongoose: elegan mongodb object modeling for node.js. [Online]. www.mongoosejs.com

[7] (2009) npm is the package manager for javascript. [Online]. www.npmjs.com

[8] Felix. (2011) Node.js Style Guide. [Online]. www.github.com/felixge/node-style-guide

[9] Edmond Meinfelder, NodeJS in 20 Minutes, 2014, https://www.youtube.com/watch?v=mq6uCM4ynpc.

[10] Wikipedia. (2010, July) 0MQ. [Online]. http://en.wikipedia.org/wiki/%C3%98MQ

[11] Pieter Hintjens, *ZeroMQ.*: O'Reilly, 2013.

[12] Tyler Treat. (2014, June) Introspections of a software engineer. [Online]. http://www.bravenewgeek.com

[13] André Schiper and Nicole Sergent Xavier Défago, Semi-Passive Replication.

[14] Roberto Baldoni, Sara Tucci-Piergiovanni and Antonino Virgillito Carlo Marchetti, "Fully Distributed Three-Tier Active Software Replication," vol. 17, no. 7, July 2006.

[15] Carlo Marchetti and Sara Tucci Piergiovanni Roberto Baldoni, Asynchronous Active Replication in Three-tier Distributed Systems.

[16] Javier Vélez Reyes, Programacion Asíncrona en Node JS, Mayo 2014.

[17] Poul Foged & Therese Hansen. (2014, Marzo) Complexity Maze. [Online]. http://complexitymaze.com/2014/03/03/javascript-promises-a-comparison-of-libraries/

[18] Fernando Pedone, From Object Replication to Database Replication.

[19] Marc Harter, T.J. Holowaychuk and Nathan Rajlich Mike Cantelon, *Node.js in Action.*: Manning, 2014.

[20] Pedro Teixeira, *Professional Node.js: Building Javascript Based Scalable Software.* Indianapolis: John Wiley & Sons, 2013.

[21] J.R. Gonzalez de Mendivil, Fully Distributed Three-Tier Active Software Replication, 13/11/2008.

[22] J.R. Gonzalez de Mendivil, Replicacion de Bases de Datos.