ZeroMQ

¿Qué es ZeroMq?

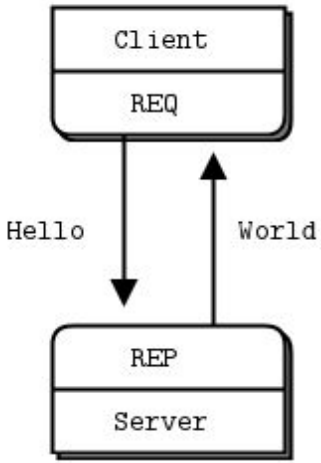
- Middleware de comunicaciones
- Biblioteca que permite gestionar las comunicaciones en un proceso separado que funciona en el mismo nodo (*brokerless*)
- Gestiona las conexiones y los mensajes a través de sockets TCP y el paso de mensajes entre colas (locales y remotas)
- Ofrece *bindings* para múltiples lenguajes y plataformas
- Utiliza conceptos similares a los sockets TCP (socket, bind, send, connect)

replies with "World" to each request.

hwserver: Hello World server in C

C++ | C# | Clojure | CL | Delphi | Erlang | F# | Felix | Go | Haskell | Haxe | Java | Lua | Node.js | Objective-C | Perl | PHP | Python | Q | Racket | Ruby | Scala | Tcl | Ada | Basic | ooc

Figure 2 - Request-Reply



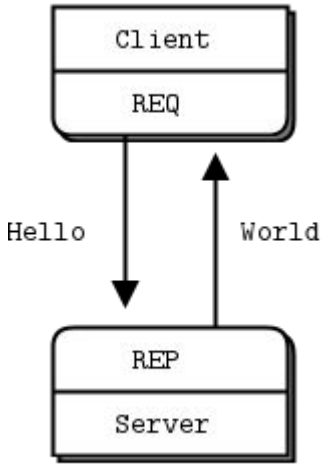
The REQ-REP socket pair is in lockstep. The client issues `zmq_send()` and then `zmq_recv()`, in a loop (or once if that's all it needs). Doing any other sequence (e.g., sending two

Ofrece

- ★ Modelo asincrónico de comunicación
- ★ Funcionamiento rápido y ligero
- ★ Independiente del orden de conexión o reconexiones (*endpoint*)
- ★ Transparencia en el transporte (*ipc*, *inproc*, *tcp*, ….)
- ★ Patrones de conexión con modelos simples y avanzados (conexiones sockets múltiples N-N, topología para las comunicaciones)

Patrones

REQ - REP



Comunicación síncrona

Node.js

Client.js

```
var zmq = require('zmq');
console.log("Connecting to hello world server…");
var requester = zmq.socket('req');

var x = 0;
requester.on("message", function(reply) {
  console.log("Received reply", x, ": [", reply.toString(), ']');
  x += 1;
  if (x === 10) {
    requester.close();
    process.exit(0);
  }
});

requester.connect("tcp://localhost:5555");

for (var i = 0; i < 10; i++) {
  console.log("Sending request", i, '…');
  requester.send("Hello");
}
```

Server.js

```
var zmq = require('zmq');
var responder = zmq.socket('rep');

responder.on('message', function(request) {
  console.log("Received request: [", request.toString(), "]");
  setTimeout(function() {
    responder.send("World");
  }, 1000);
});

responder.bind('tcp://*:5555', function(err) {
  if (err) {
    console.log(err);
  } else {
    console.log("Listening on 5555…");
  }
});
```

Node.js

Client.js

```
var zmq = require('zmq');
console.log("Connecting to hello world server…");
var requester = zmq.socket('req');

var x = 0;
requester.on("message", function(reply) {
  console.log("Received reply", x, ": [", reply.toString(), ']');
  x += 1;
  if (x === 10) {
    requester.close();
    process.exit(0);
  }
});

requester.connect("tcp://localhost:5555");

for (var i = 0; i < 10; i++) {
  console.log("Sending request", i, '…');
  requester.send("Hello");
}
```

socket

conexión

Server.js

```
var zmq = require('zmq');
var responder = zmq.socket('rep');

responder.on('message', function(request) {
  console.log("Received request: [", request.toString(), "]");
  setTimeout(function() {
    responder.send("World");
  }, 1000);
});

responder.bind('tcp://*:5555', function(err) {
  if (err) {
    console.log(err);
  } else {
    console.log("Listening on 5555…");
  }
});
```

Node.js

Client.js

```
var zmq = require('zmq');
console.log("Connecting to hello world server…");
var requester = zmq.socket('req');

var x = 0;
requester.on("message", function(reply) {
  console.log("Received reply", x, ": [", reply.toString(), ']');
  x += 1;
  if (x === 10) {
    requester.close();
    process.exit(0);
  }
});

requester.connect("tcp://localhost:5555");

for (var i = 0; i < 10; i++) {
  console.log("Sending request", i, '…');
  requester.send("Hello");
}
```

Recepción mensajes

Server.js

```
var zmq = require('zmq');
var responder = zmq.socket('rep');

responder.on('message', function(request) {
  console.log("Received request: [", request.toString(), "]");
  setTimeout(function() {
    responder.send("World");
  }, 1000);
});

responder.bind('tcp://*:5555', function(err) {
  if (err) {
    console.log(err);
  } else {
    console.log("Listening on 5555…");
  }
});
```

Envío mensajes

Patrones

REQ - REP

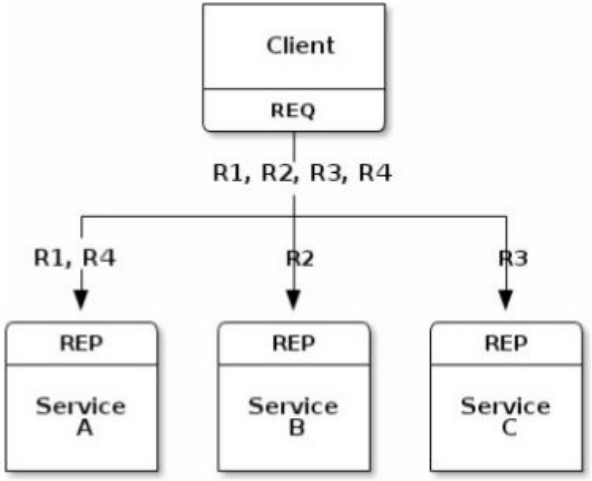
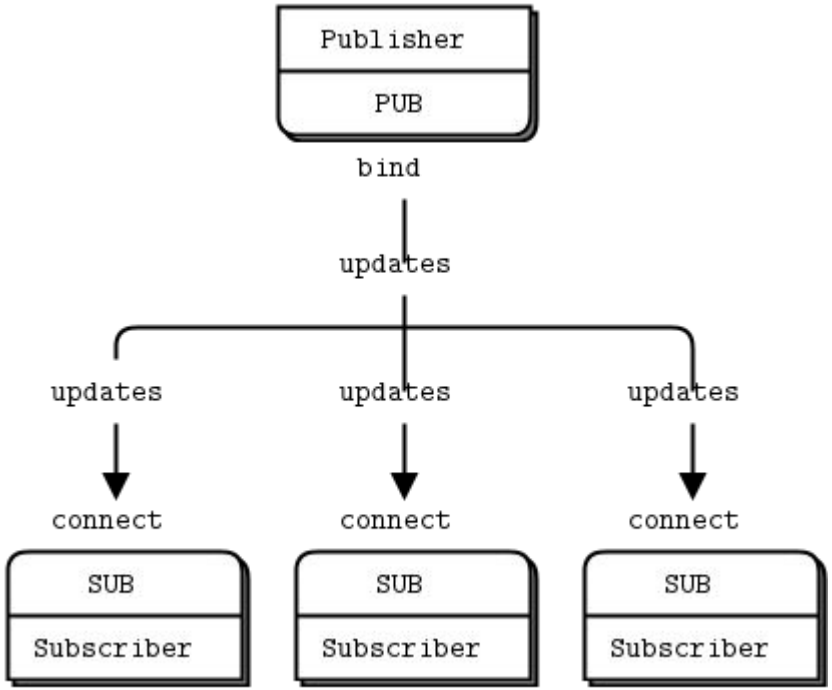


Figure 18 – Load balancing of requests

Balanceado de carga de peticiones

Patrones

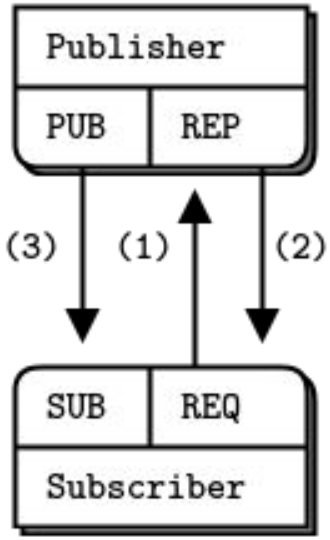
PUB - SUB



Distribución de información en un único sentido

Patrones

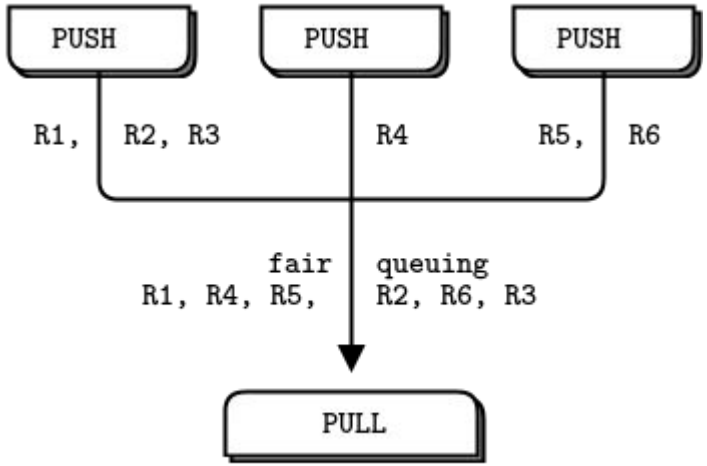
PUB - SUB



Sincronización inicial
El nodo publicador espera a que estén todos los subscriber listos (por ejemplo los cuenta)

Patrones

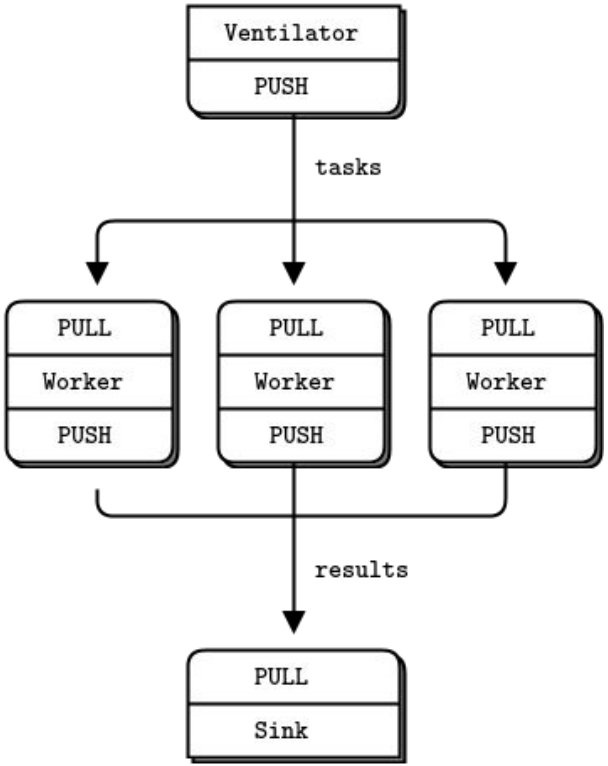
PUSH - PULL



Modelo productor/consumidor
Se dan por defecto políticas de *fair queuing*
En la práctica puede incumplirse aparentemente

Patrones

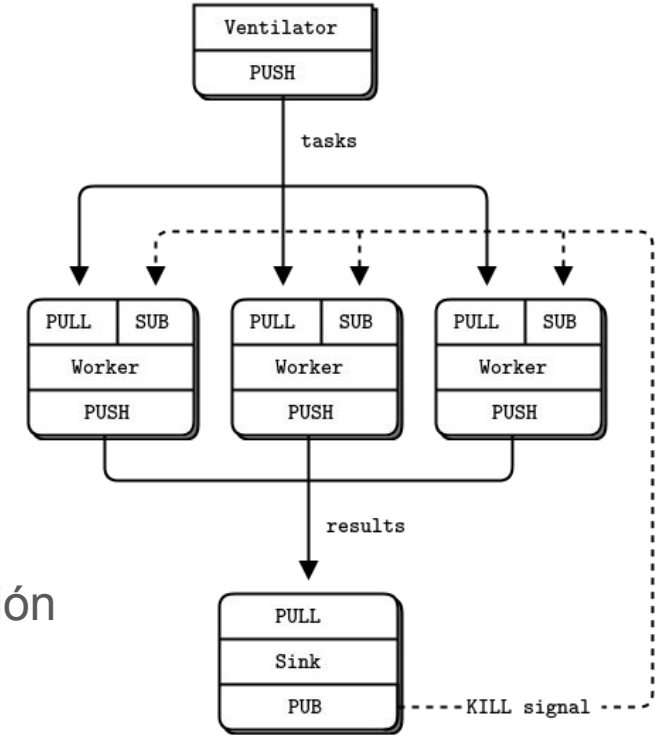
PUSH - PULL



Distribución *fanout*, paralelización y recolección de información

Patrones

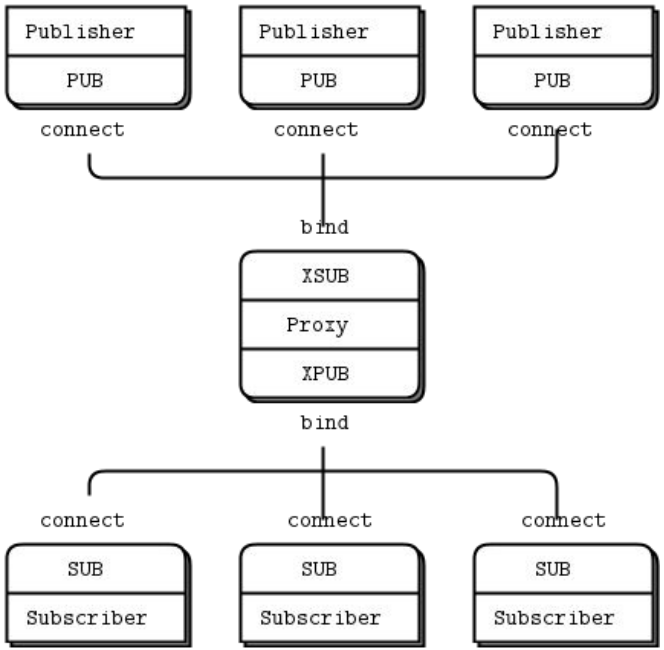
PUSH - PULL



Distribución *fanout*, paralelización
y recolección de información, con gestión
de cierre de aplicación organizado

Patrones (con intermediarios brokers/proxies)

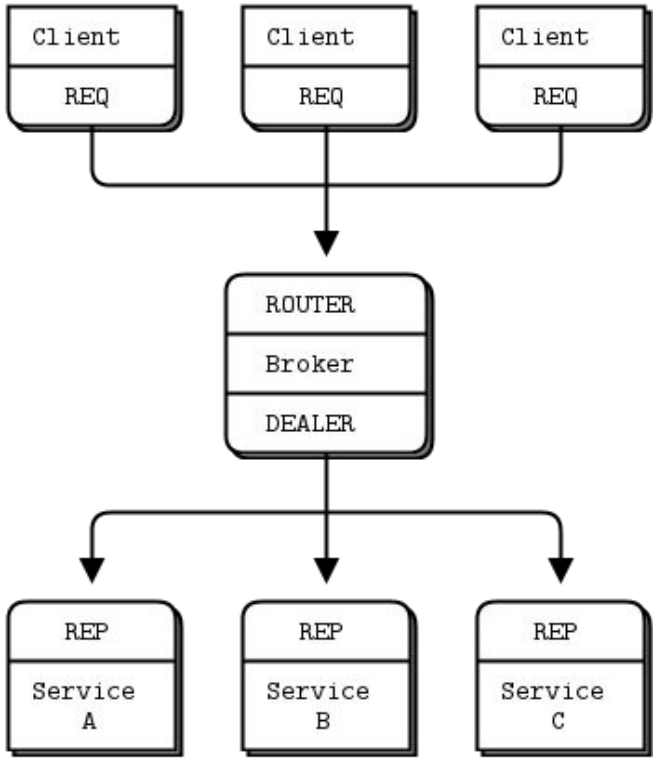
XPUB - XSUB



Permite que las suscripciones lleguen a los publicadores. XPUB, XSUB son switches

Patrones (con intermediarios brokers/proxies)

ROUTER-DEALER

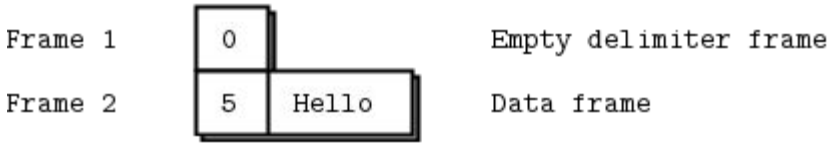


De esta manera un servicio puede recibir varias peticiones de forma asincrónica
ROUTER identifica las peticiones, para contestar al cliente adecuado

Patrones (formato mensajes)

Según la combinación de sockets, los mensajes tienen cierto formato o estructura

REQ - REP

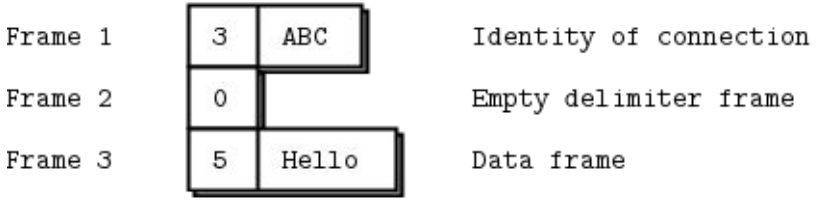


En este patrón, de manera transparente, REQ incluye un separador y REP lo consume

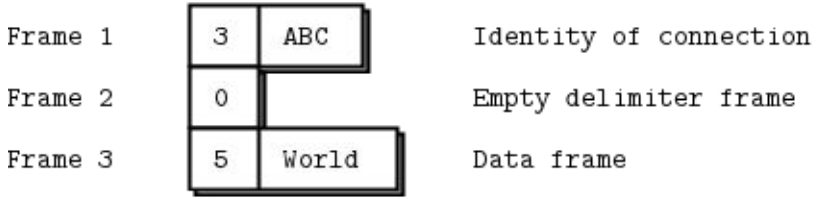
Patrones (formato mensajes)

Según la combinación de sockets, los mensajes tienen cierto formato o estructura

REQ - ROUTER



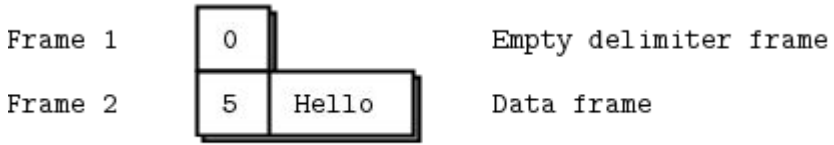
En este patrón, de manera transparente, REQ incluye un separador y ROUTER recibe el mensaje añadiendo el identificador del cliente. Para la respuesta ROUTER necesita una estructura similar



Patrones (formato mensajes)

Según la combinación de sockets, los mensajes tienen cierto formato o estructura

DEALER - REP



DEALER no afecta al mensaje de ninguna manera, pero REP se espera un separador, por lo que se debe incluir

Otras combinaciones

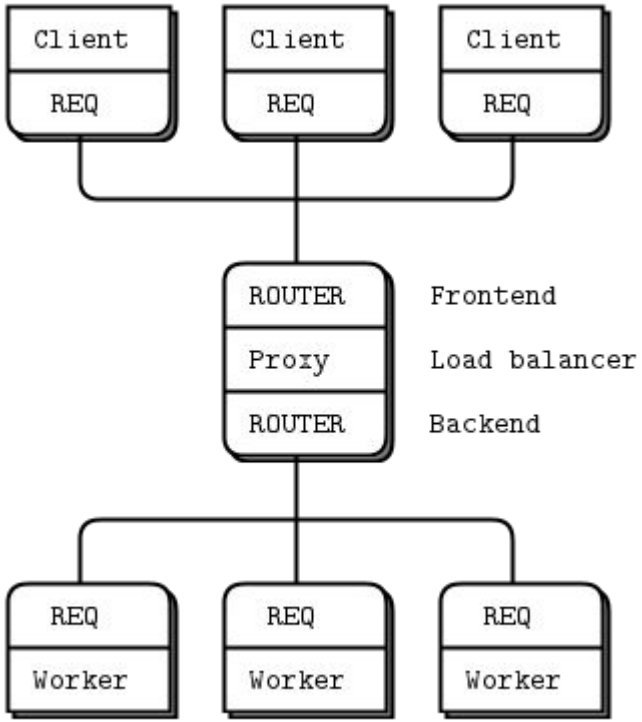
DEALER to ROUTER. Se pueden enviar mensajes a múltiples servidores sin esperar respuesta

DEALER to DEALER, similar a dos pull push en dos sentidos. Sólo funciona entre dos pares

ROUTER to ROUTER. El más versátil, pero complicado, permite conexiones N-N

Patrones (con intermediarios brokers/proxies)

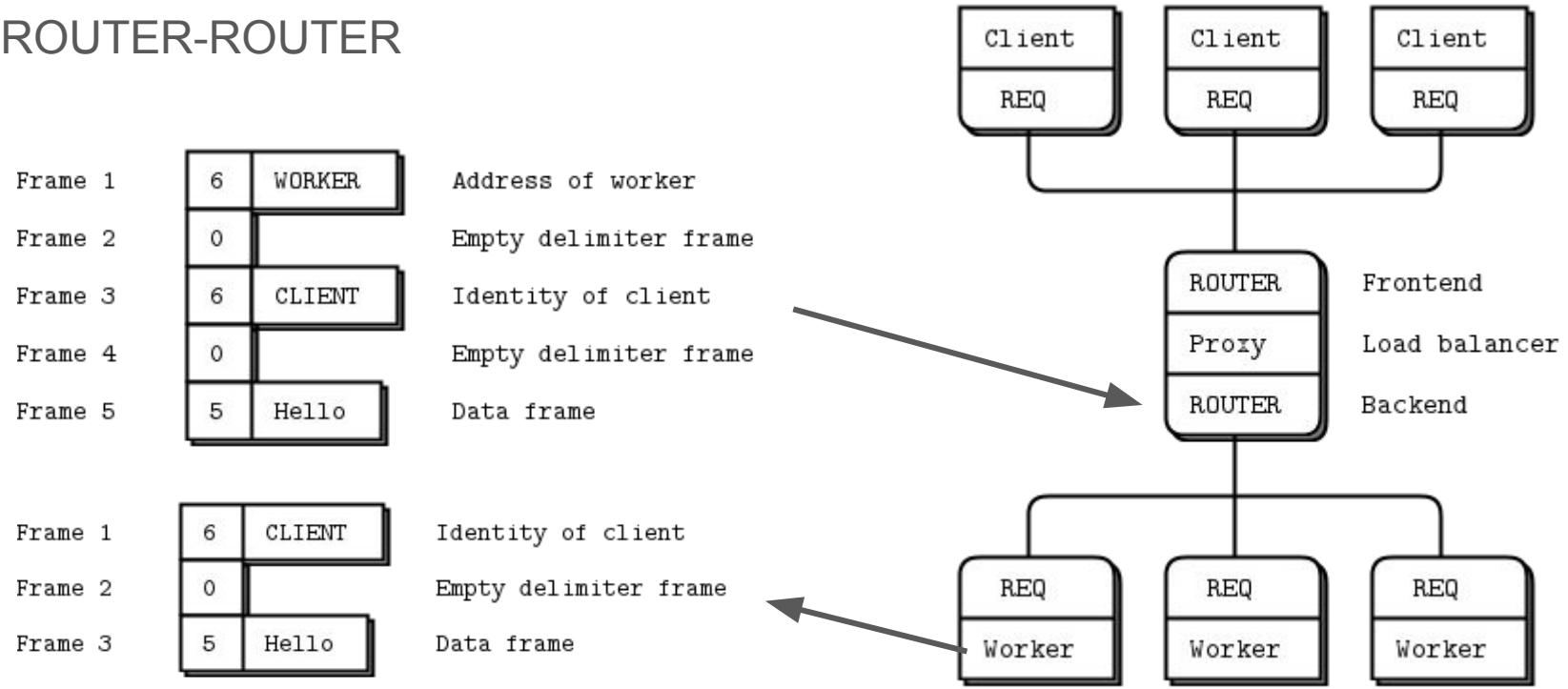
ROUTER-ROUTER



De esta manera un servicio puede recibir varias peticiones de forma asincrónica
ROUTER identifica las peticiones, para contestar al cliente adecuado

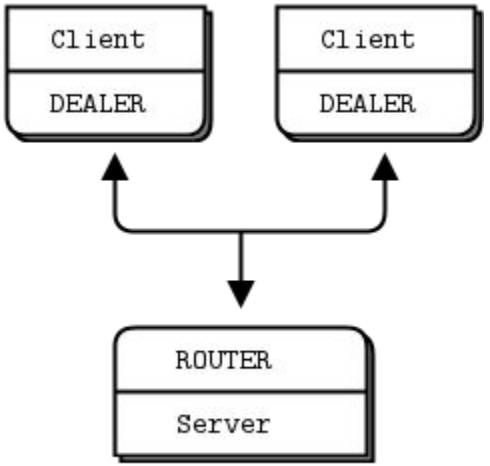
Patrones (con intermediarios brokers/proxies)

ROUTER-ROUTER



Patrones (cliente asincrónico/servidor)

DEALER-ROUTER



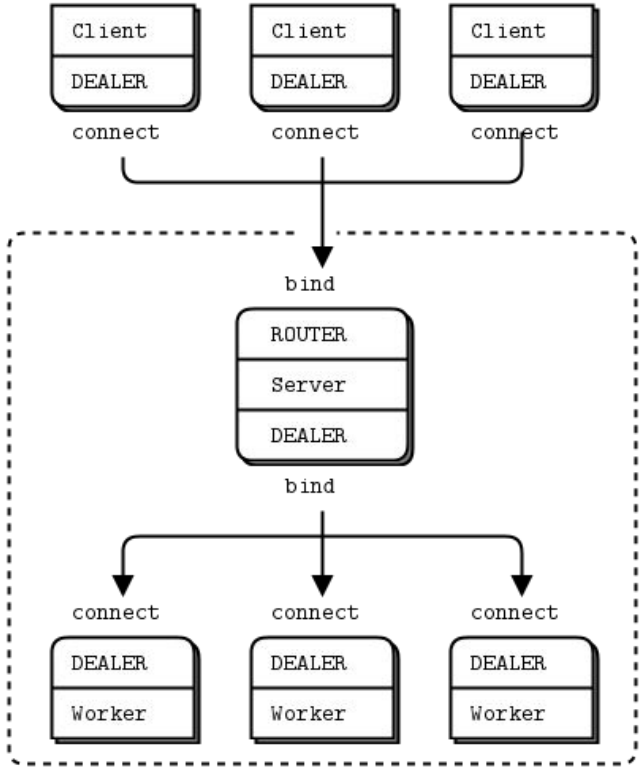
Los clientes lanzan peticiones asincrónicamente
a un servidor, que puede contestar 0, 1 o más respuestas, sin esperar peticiones.

Patrones (cliente asincrónico/servidor)

DEALER-ROUTER

DEALER-DEALER

Nos permite enviar múltiples respuestas, sin esperar peticiones. El servidor también es asincrónico sin estado



Ejemplo: planteamiento...

- Tenemos workers, en diferentes plataformas, varios cientos por cluster y una docena de clusters.
- Las peticiones de los clientes consisten en tareas para los workers. Se debe asignar a un worker disponible (preferentemente local). Los clientes vienen y van arbitrariamente (son dinámicos).
- Los clusters se pueden añadir o quitar en cualquier momento, esto supone que desaparecen los workers y clientes asociados a el mismo
- Los clientes sólo envían una tarea y esperan respuesta, si no la obtienen en un cierto tiempo, la vuelven a enviar.
- Los workers sólo hacen una tarea, si se produce algún problema, se reinician por un script.

Diseño para un cluster

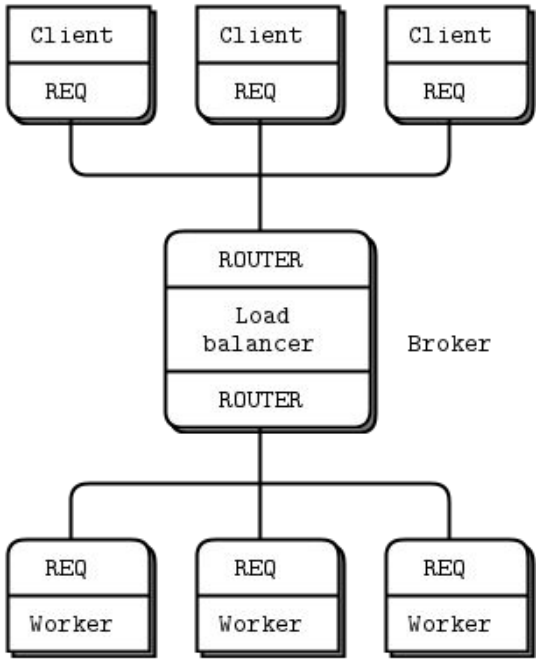
Los workers y clientes son síncronos.

Los workers son iguales (no hay servicios diferentes) y anónimos (los clientes no se dirigen a uno en concreto).

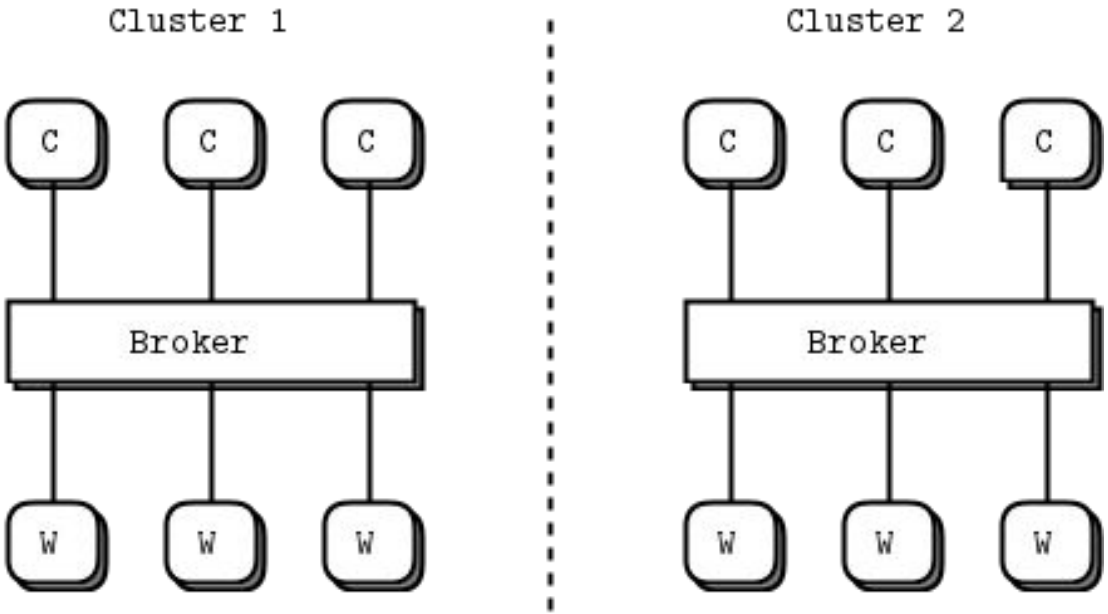
Los clientes y los workers no necesitan hablar directamente.

Diseño para un cluster

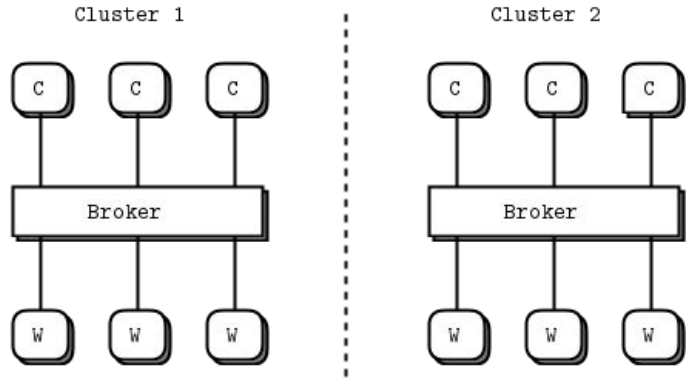
La solución viene dada por el balanceador de carga visto anteriormente

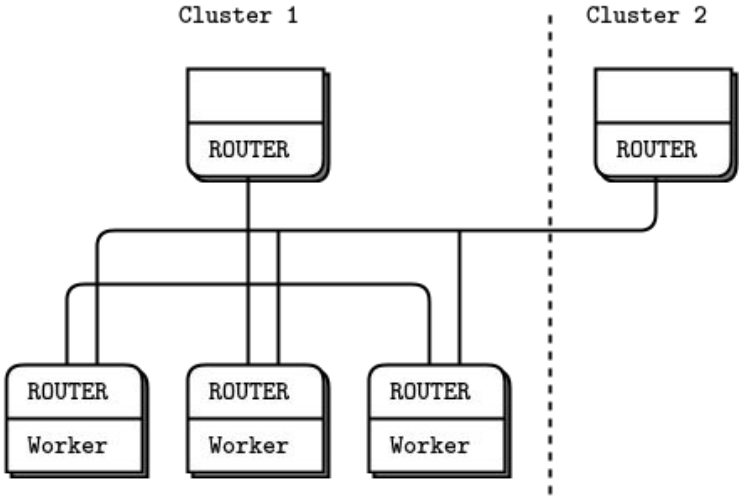


Múltiples clusters



Múltiples clusters

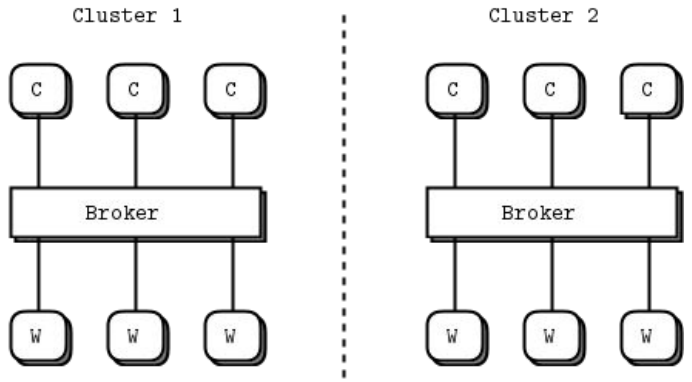


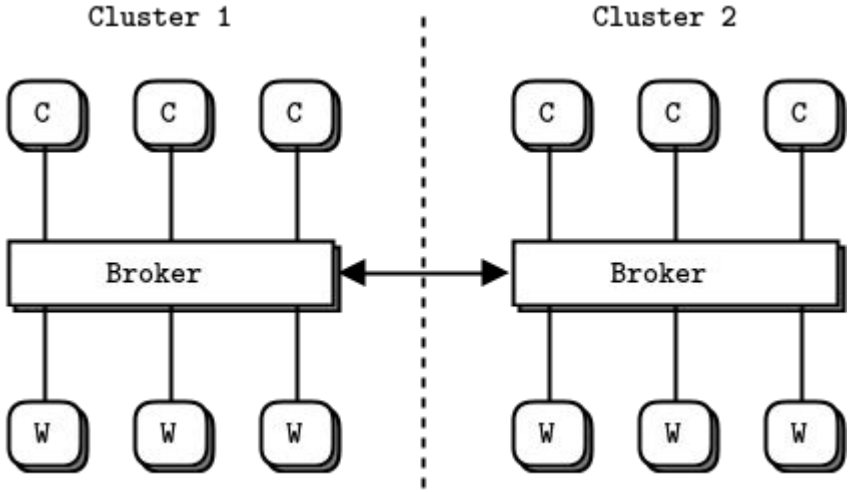


Un mismo worker podría recibir tareas de dos broker

Los workers serían seleccionados independientemente de si son locales o de otro cluster

Múltiples clusters

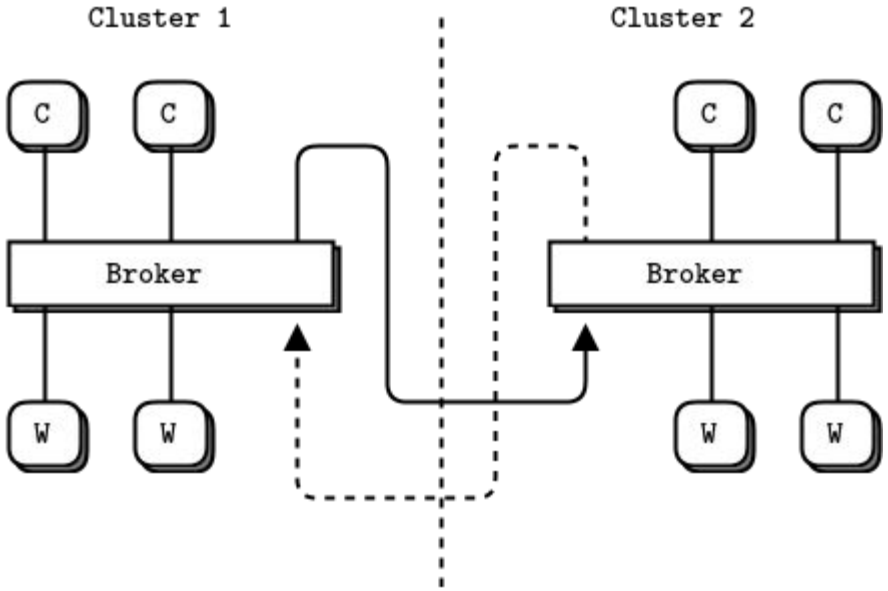




Los brokers se comunican y se subcontratan si es necesario

Permite realizar un enrutado selectivo, solo en caso necesario

Múltiples clusters



Debemos comunicar entre brokers la capacidad, y también enviar la tarea, y recibir la respuesta
Los brokers pueden actuar en forma de cliente y de worker.
Pero el problema que sucede es que los clientes actúan de manera síncrona, y sólo aceptaría una tarea de esta manera.

Múltiples clusters

Vamos a conectar a los brokers en una conexión P2P. Todos conocen a todos.

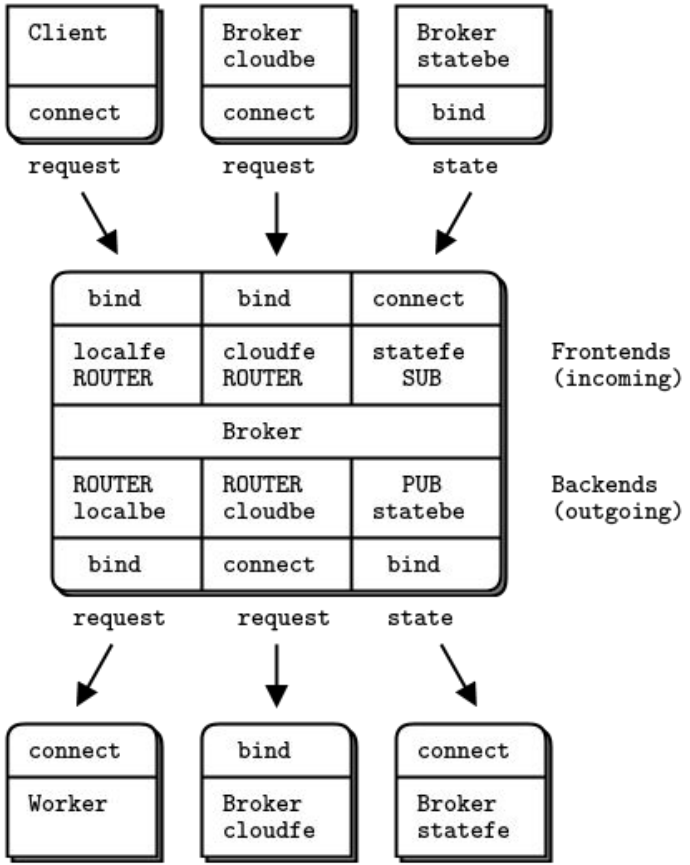
Se usa un patrón PUB-SUB para hacer conocer al resto de brokers cuantos workers quedan disponibles

Se usa un socket ROUTER para hacer la petición a otro broker.

Se usa un socket ROUTER para recibir las peticiones de otro broker.

Múltiples clusters

localfe -> recibe peticiones clientes locales
localbe -> conecta las peticiones con workers locales
cloudfe -> recibe peticiones del resto de brokers
cloudbe -> realiza peticiones al resto de brokers
statefe -> recibe la capacidad del resto de brokers
statebe -> emite la capacidad al resto de brokers



Múltiples clusters

Capacidad de los brokers (PUB-SUB)

Emitimos un mensaje cada vez que cambia (mucho tráfico) o cada cierto tiempo. Un suscriptor solo mantiene la capacidad durante un tiempo, y después asume que el publicador no está disponible (poner capacidad 0).

Múltiples clusters

Peticiones y respuestas (localfe - cloudfe)

En la petición (cloudfe o localfe) se identifica el cliente o el router que la hace que se pasa al worker. Si al responder un worker (por localbe) no está en la lista de clientes, entonces comprobar si es de un router. Enrutarla hacía el cliente o el broker.

Si llega una petición (localfe) y no hay workers, elegir el broker con mayor capacidad y enviar la petición por cloudbe.

Reliability

El código puede terminar abruptamente o bloquearse

La colas de mensajes se pueden saturar (clientes lentos)

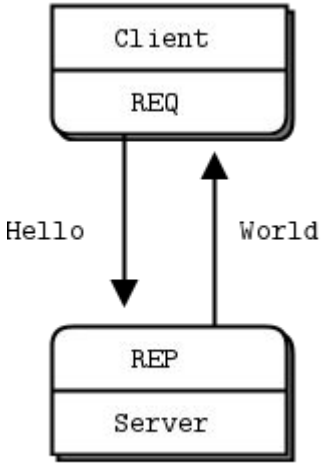
Las redes pueden fallar (¿qué sucede con los mensajes enviados, se pierden o se reenvían?)

Fiabilidad

REQ - REP

Si el servidor muere procesando una petición o si se pierde la petición o la respuesta por la red. El cliente se queda esperando la respuesta.

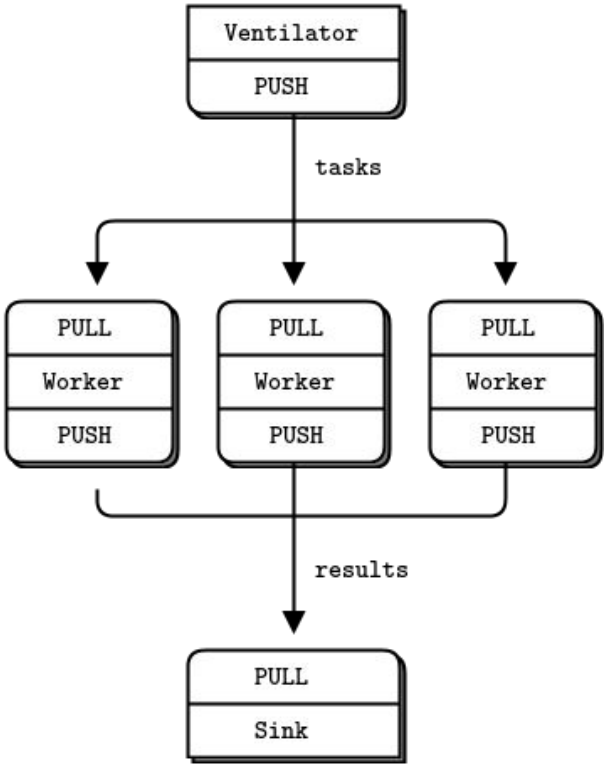
Una solución es poner un timeout, y si no se ha recibido una respuesta, cerrar y reabrir el socket REQ y reenviar la petición.



Fiabilidad

PUSH - PULL

Si muere un worker, el productor no se da cuenta. Pero el recolector de datos puede darse cuenta de que una tarea no ha sido terminada en un tiempo dado y pedirla de nuevo al productor.
Si muere el productor, el recolector puede detectar que no se ha completado y solicitar el lote completo o sólo las tareas restantes

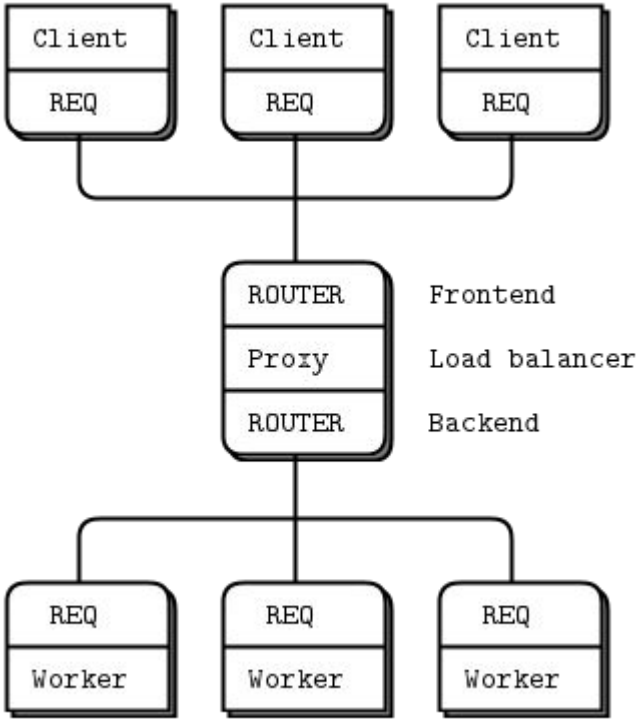


Fiabilidad

Con intermediario (gestión de colas de petición)

Las fuentes de fallos suelen originarse en los servidores. Ponemos a un intermediario que es capaz de redirigir las nuevas peticiones a otros servidores.

El proxy puede convertirse en un punto de fallo
Si se viene abajo el proxy y se reinicia, los workers no existirán para el proxy.
Si fallan los workers el proxy sólo se dará cuenta cuando envíe una petición.

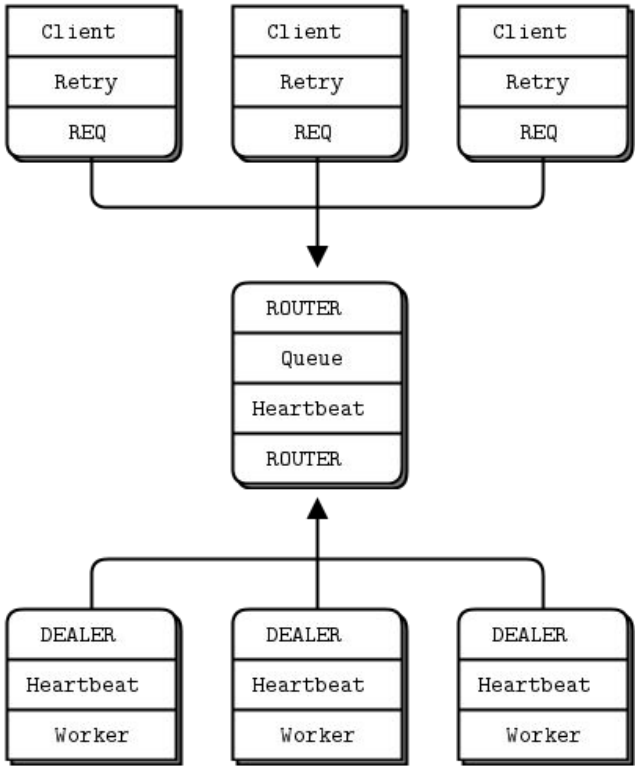


Fiabilidad

Con intermediario (gestión de colas de petición) y *heartbeat*.
El socket DEALER debe gestionar el sobre. Este socket permite enviar un hearbeat desde el Worker al proxy, y también a la inversa.

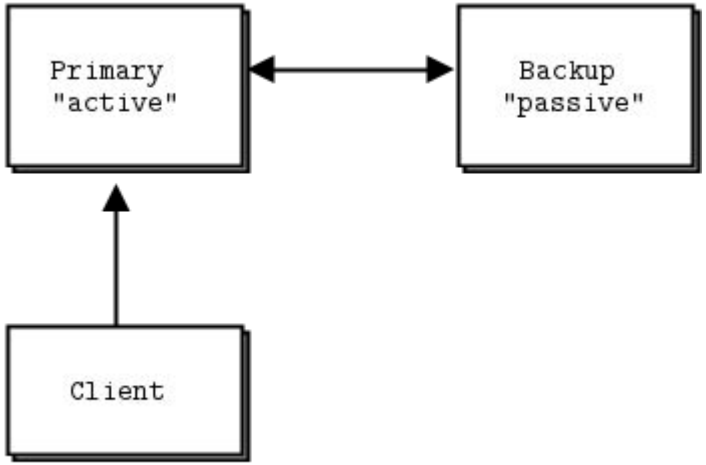
El worker podrá presentarse ante un reiniciado del proxy.

El proxy podré eliminar de su lista de workers disponibles los que no ha recibido el heartbeat



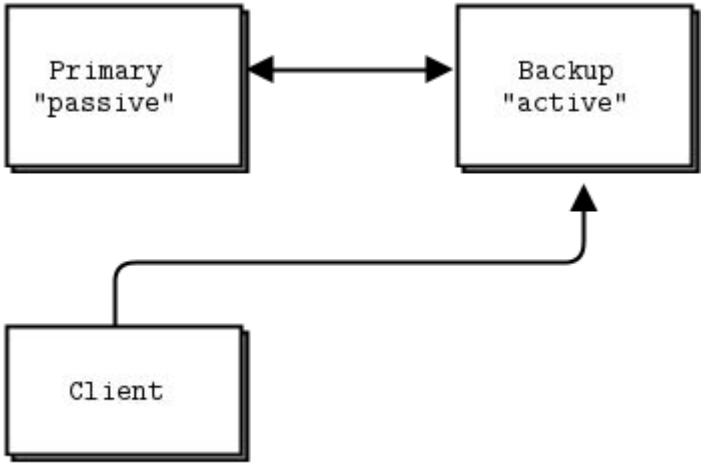
Alta disponibilidad

Patrón estrella en binario



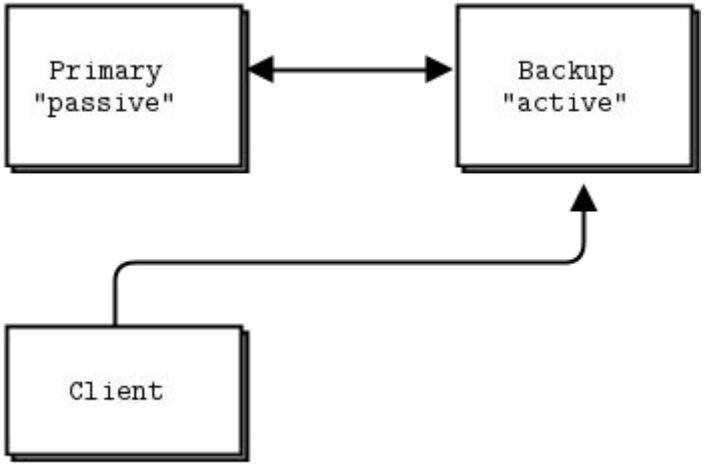
Alta disponibilidad

Patrón estrella en binario durante fallo



Alta disponibilidad

Patrón estrella en binario (sindrome *split-brain*)



Alta disponibilidad

Máquina finita de estados para un patrón en estrella binaria